# Coordinate-Independent Computations on Differential Equations 

by<br>Kevin K. Lin<br>Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of<br>Master of Engineering in Electrical Engineering and Computer Science<br>at the<br>MASSACHUSETTS INSTITUTE OF TECHNOLOGY<br>August 11, 1997<br>(C) Massachusetts Institute of Technology 1997. All rights reserved.



Author $\qquad$ August 11, 1997

## $n$



Certified by $\qquad$ Gerald Jay Sussman Matsushita Professor of Electrical Engineering Thesis Supervisor


Accepted by $\qquad$
$\qquad$

# Coordinate-Independent Computations on Differential Equations 

by

Kevin K. Lin

Submitted to the Department of Electrical Engineering and Computer Science on August 11, 1997, in partial fulfillment of the requirements for the degree of Master of Engineering in Electrical Engineering and Computer Science


#### Abstract

This project investigates the computational representation of differentiable manifolds and the use of such abstractions to help manage multiple coordinate systems on $n$-dimensional spaces. This idea is applied to the accurate integration of ordinary differential equations using multiple coordinate systems. An attempt is also made to apply this idea to the numerical solution of linear partial differential equations, although in this case unexpected difficulties arise even with the simplest equations.


Thesis Supervisor: Gerald Jay Sussman
Title: Matsushita Professor of Electrical Engineering

## Acknowledgments

As is the case with any undertaking, there are far too many people to thank in relation to this project. If I neglected anyone, it is purely out of failure of memory and I beg for their forgiveness.

To begin, I would like to thank Norman K. Yeh and Mariya Minkova for proof-reading my thesis proposal, and Tim McNerney for suggestions on how to present manifolds. Because of them, the presentation is much better than the rambling mess that was my first draft.

To the folk at Switzerland, especially Daniel Coore and Jim McBride: Thanks for the stimulating (and sometimes late-night) discussions on life, the universe, and everything. Thanks are also due to Eric Grimson, Thanos Siapas, and Ken Yip for crucial references, to Hardy Mayer and Michael Chechelnitsky for many helpful conversations on PDEs, and to Rebecca Bisbee and Anne Hunter for making sure that all the important things got done.

Portions of this work was supported by the Barry M. Goldwater Foundation, without whose help it would have been difficult to get this far.

I am forever indebted to my teachers throughout the years: To Mr. Reyerson at BCP, and Professors Flanigan and Morris at SJSU, for introducing me to mathematics; to Professors Munkres, Guillemin, and Wisdom for being such great teachers; and finally, to Gerald Jay Sussman, who has helped shape my interests, guided my work, and above all else encouraged me onward as I took the first faltering steps into a world of breathtaking wonders.

To my family and all my friends: You have made a great difference, and I could not have made it without you.

Still round the corner there may wait
A new road or a secret gate;
And though I oft have passed them by, A day will come at last when I
Shall take the hidden paths that run
West of the Moon, East of the Sun.
-J. R. R. Tolkien

To Grandpa.
1908-1994

## Contents

1 Introduction ..... 12
2 Ordinary Differential Equations and Manifolds ..... 14
2.1 A brief introduction to manifolds ..... 14
2.1.1 The spherical pendulum ..... 14
2.1.2 Differentiable manifolds ..... 16
2.1.3 Some examples ..... 18
2.1.4 Tangent vectors ..... 21
2.1.5 Smooth maps and differentials ..... 24
2.1.6 Tangent bundles ..... 27
2.1.7 Making new manifolds ..... 29
2.1.8 Boundaries ..... 30
2.2 Vector fields and differential equations ..... 31
2.2.1 Smooth vector fields ..... 31
2.2.2 Flows generated by smooth vector fields ..... 32
2.2.3 Manifolds and classical mechanics ..... 34
2.3 Numerical experiments ..... 38
2.3.1 The circle field ..... 38
2.3.2 The spherical pendulum ..... 40
2.3.3 Rigid body motion and coordinate singularities ..... 45
2.4 Directions for future work ..... 55
3 Linear partial differential equations ..... 56
3.1 Partial differential operators on manifolds ..... 57
3.2 Approaches to discretization ..... 58
3.3 Finite differences on manifolds ..... 60
3.3.1 Generating coefficients for irregular sample points ..... 63
3.3.2 Solving linear algebraic equations ..... 65
3.3.3 Numerical examples ..... 66
3.4 Finite elements on manifolds ..... 74
3.4.1 Integration on manifolds ..... 75
3.4.2 More about boundaries ..... 80
3.4.3 Computing with finite elements on manifolds ..... 83
3.4.4 Local finite-elements ..... 83
3.4.5 Basic FEM algorithm on manifolds ..... 89
3.4.6 Interpolation between charts ..... 92
3.4.7 Some numerical results. ..... 98
3.4.8 The problem with interpolation ..... 101
3.4.9 Other approaches to FEM on manifolds ..... 106
3.5 Some comments on mesh generation ..... 117
3.6 Directions for future work ..... 118
3.6.1 Improvements to finite differences ..... 118
3.6.2 Improvements to finite elements ..... 119
3.6.3 Other methods ..... 119
4 Problems with Time ..... 120
4.1 The linear wave equation ..... 121
4.2 Initial value problems and characteristics ..... 122
4.2.1 Characteristic curves for a first-order equation ..... 123
4.2.2 Characteristics for general equations ..... 124
4.2.3 Variational principles revisited ..... 125
4.2.4 Galerkin's method and the initial value problem ..... 127
4.3 Variations on a theme of Lagrange ..... 128
4.3.1 Modifying the action principle ..... 128
4.3.2 Modifying the domain ..... 135
4.4 Difficulties with the spacetime approach ..... 140
4.4.1 Why the variations failed ..... 140
4.4.2 Other problems ..... 141
4.5 Directions for future work ..... 142
A Background Material on Partial Differential Equations ..... 143
A. 1 Matrix inversion ..... 143
A.1.1 Iterative methods and relaxation ..... 143
A.1.2 Jacobi iteration ..... 144
A.1.3 Gauss-Seidel iteration ..... 145
A.1.4 Overrelaxation ..... 145
A. 2 A brief introduction to finite elements ..... 145
A.2.1 Introduction ..... 145
A.2.2 Partial differential equations ..... 146
A.2.3 The Rayleigh-Ritz Method ..... 148
A.2.4 Galerkin's method ..... 153
B Odds \& Ends ..... 155
B. 1 Charts without a manifold ..... 155
B. 2 Integration of differential forms on oriented manifolds ..... 156
B.2.1 Stokes's theorem ..... 157
C Complete Program Listings ..... 158
C. 1 Computational manifolds ..... 158
C.1.1 basis-imb.scm ..... 158
C.1.2 basis-poly.scm ..... 160
C.1.3 basis-real.scm ..... 162
C.1.4 boundary.scm ..... 163
C.1.5 bug.scm ..... 167
C.1.6 charts.scm ..... 167
C.1.7 cotangent.scm ..... 169
C.1.8 fields.scm ..... 173
C.1.9 hamilton.scm ..... 179
C.1.10 imbedding.scm ..... 179
C.1.11 job-ode.scm ..... 180
C.1.12 job-pde.scm ..... 181
C.1.13 lagrange.scm ..... 182
C.1.14 linear.scm ..... 184
C.1.15 load-main.scm ..... 187
C.1.16 load-ode.scm ..... 187
C.1.17 load-pde.scm ..... 188
C.1.18 lshared.scm ..... 189
C.1.19 manifold.scm ..... 192
C.1.20 misc-math.scm ..... 197
C.1.21 misc.scm ..... 197
C.1.22 ode-examples.scm ..... 199
C.1.23 ode-fast.scm ..... 203
C.1.24 ode.scm ..... 204
C.1.25 pde-aux.scm ..... 209
C.1.26 pde-charts.scm ..... 210
C.1.27 pde-cmpgrd.scm ..... 213
C. 1.28 pde-collect.scm ..... 219
C.1.29 pde-collect1.scm ..... 220
C.1.30 pde-collect2.scm ..... 220
C.1.31 pde-collect3.scm ..... 221
C.1.32 pde-collect4.scm ..... 221
C.1.33 pde-config.scm ..... 222
C.1.34 pde-debug.scm ..... 222
C.1.35 pde-elements.scm ..... 232
C.1.36 pde-examples.scm ..... 233
C.1.37 pde-gentest.scm ..... 235
C.1.38 pde-main.scm ..... 236
C.1.39 pde-mergers.scm ..... 244
C.1.40 pde-nodes.scm ..... 249
C.1.41 pde-ops.scm ..... 251
C.1.42 pde-test.scm ..... 252
C.1.43 pde-thesis.scm ..... 255
C.1.44 pde-thesis1.scm ..... 256
C.1.45 pde-thesis2.scm ..... 257
C.1.46 pde-thesis3.scm ..... 258
C.1.47 pde-thesis4.scm ..... 258
C.1.48 pde-tools.scm ..... 259
C.1.49 pde-works.scm ..... 263
C.1.50 product.scm ..... 265
C.1.51 ranges.scm ..... 267
C.1.52 richardson.scm ..... 269
C.1.53 rigid-body.scm ..... 271
C.1.54 rigid-compute.scm ..... 274
C.1.55 rigid-errors.scm ..... 274
C.1.56 rigid-fields.scm ..... 275
C.1.57 rigid-read.scm ..... 282
C.1.58 rigid-test.scm ..... 283
C.1.59 rigid-too.scm ..... 284
C.1.60 rigid-tops.scm ..... 286
C.1.61 rigid.scm ..... 287
C.1.62 smooth.scm ..... 293
C.1.63 spaces.scm ..... 298
C.1.64 stubs.scm ..... 306
C.1.65 tangent.scm ..... 307
C.1.66 vbundle.scm ..... 312
C. 2 Finite elements ..... 314
C.2.1 2d-domains.scm ..... 314
C.2.2 2d-examples.scm ..... 323
C.2.3 2d-operators.scm ..... 326
C.2.4 2d-poly-basis.scm ..... 328
C.2.5 2d-real-basis.scm ..... 333
C.2.6 2d-real-diff.scm ..... 334
C.2.7 2d-trapezoid.scm ..... 335
C.2.8 basis.scm ..... 336
C. 2.9 bent.scm ..... 338
C.2.10 collect.scm ..... 340
C.2.11 debug.scm ..... 341
C.2.12 delaunay.scm ..... 346
C.2.13 delaux.scm ..... 349
C.2.14 dyntable.scm ..... 351
C.2.15 edge.scm ..... 352
C.2.16 fem.scm ..... 356
C.2.17 job.scm ..... 362
C.2.18 job1.scm ..... 362
C.2.19 load.scm ..... 363
C.2.20 matlib.scm ..... 363
C.2.21 nodes.scm ..... 371
C.2.22 opalg.scm ..... 372
C.2.23 operators.scm ..... 373
C.2.24 relax.scm ..... 374
C.2.25 sparse.scm ..... 375
C.2.26 thesis.scm ..... 378
C.2.27 thesis1.scm ..... 379
C.2.28 util-too.scm ..... 379
C.2.29 util.scm ..... 381
C. 3 Finite differences ..... 382
C.3.1 farray.h ..... 382
C.3.2 gunk.h ..... 382
C.3.3 matrix.h ..... 384
C.3.4 random.h ..... 384
C.3.5 stat.h ..... 385
C.3.6 accum.c ..... 385
C.3.7 approx.c ..... 388
C.3.8 band.c ..... 395
C.3.9 blud.c ..... 396
C.3.10 check.c ..... 402
C.3.11 circle.c ..... 404
C.3.12 dot.c ..... 406
C.3.13 estimate.c ..... 411
C.3.14 farray.c ..... 413
C.3.15 femcompact.c ..... 414
C.3.16 femstats.c ..... 415
C.3.17 fill.c ..... 416
C.3.18 filt.c ..... 417
C.3.19 grad.c ..... 419
C.3.20 gs_mat.c ..... 420
C.3.21 gsort.c ..... 423
C.3.22 gunk.c ..... 425
C.3.23 gunk2mat.c ..... 432
C.3.24 hilbert.c ..... 434
C.3.25 improve.c ..... 435
C.3.26 jac_mat.c ..... 439
C.3.27 list2grid.c ..... 442
C.3.28 lud.c ..... 443
C.3.29 maplegrid.c ..... 446
C.3.30 matrix.c ..... 447
C.3.31 migrate.c ..... 453
C.3.32 mkbd.c ..... 456
C.3.33 mkgrid.c ..... 460
C.3.34 mksq.c ..... 461
C.3.35 new2old.c ..... 465
C.3.36 norm_eqs.c ..... 466
C.3.37 nstat.c ..... 475
C.3.38 old2new.c ..... 475
C.3.39 order.c ..... 476
C.3.40 peek.c ..... 477
C.3.41 peek_coeff.c ..... 480
C.3.42 poly.c ..... 483
C.3.43 pot.c ..... 489
C.3.44 random.c ..... 489
C.3.45 relax.c ..... 490
C.3.46 repos.c ..... 493
C.3.47 reset_val.c ..... 495
C.3.48 rref.c ..... 496
C.3.49 sample.c ..... 497
C.3.50 scramble.c ..... 501
C.3.51 slice.c ..... 502
C. 3.52 spect.c ..... 504
C.3.53 sqstat.c ..... 506
C.3.54 stat.c ..... 507
C.3.55 walk.c ..... 509

## Chapter 1

## Introduction

Partial differential equations ${ }^{1}$ arise naturally in a large variety of physical problems. Like ordinary differential equations, the majority of partial differential equations cannot be solved analytically save in special cases. Thus, efficient and accurate numerical solutions of partial differential equations are essential in many applications. However, unlike ordinary differential equations, the solution of even linear partial differential equations can be a non-trivial task. There are no general methods that apply to all types of partial differential equations, and it is often necessary to exploit special structures in the problem at hand.

This project explores the the numerical solution of partial differential equations using coordinate-independent representations. This approach makes possible the use of whatever coordinate system that happens to simplify the problem locally. As a result, we can exploit the structure and locality of interaction inherent in many physical systems, which can provide more accurate solutions as well as insights into the physical and mathematical structure of problems. In addition, it may also help us reformulate such problems for distributed computers. This project focuses on low-order linear equations in low dimensions for which analytical solutions are available, so that we can check our numerical solutions.

The rest of this document is divided into three chapters and three appendices. The first chapter develops the idea of differentiable manifolds and other basic concepts from modern differential geometry, and applies these concepts directly to the representation and solution of ordinary differential equations, particularly those arising from classical mechanics. Next, partial differential equations are discussed; for simplicity, the discussion is restricted to simple scalar linear equations, such as Laplace's equation over regions in the plane. Finally, the coordinate-independent solution of equations involving time, such as the linear wave equation, is investigated; in this context, the spacetime representation of equations (rather than the traditional "space + time") seems most natural.

[^0]While the manifold abstraction works beautifully with ordinary differential equations, some unexpected difficulties arise when dealing with even the simplest partial differential equations. Thus, most of the methods described herein, with one notable exception, actually do not work all that well, and in some cases completely fail. Thus, there is much work to be done. However, given the limited scope and time scale of this thesis project, not all possible solutions to these problems can be adequately explored. It is hoped that these ideas can be explored more fully in the future.

Appendix A includes relevant background material on partial differential equations. In particular, it presents the numerical methods that form a basis for this project, as well as some important geometric and analytical properties of partial differential equations. Appendix B contains some material on the theory of manifolds that was not directly needed in the thesis, while Appendix C contains complete program listings.

The entire document, including the material on abstract manifolds, suppose only a strong background in linear algebra and advanced calculus; little familiarity with more advanced mathematics is assumed. Also, it is helpful, though not necessary, to be acquainted with classical mechanics in Chapter 2, and $\S 4.3 .1$ presumes some acquaintance with the basic concepts of relativity.

Finally, a note about the presentation: Throughout this document, programs implementing the main ideas will be presented alongside the mathematics. This serves a few different purposes: First, because this project is fundamentally about computational techniques, it would not be complete without actual programs. Second, it is often the case that seeing something presented in different ways aids in understanding, especially in subjects involving a significant amount of abstraction. Furthermore, programming languages, by their very nature, force one to be as careful with the details as with the main concepts, something that math and physics texts sometimes neglect. The language chosen for this project is Scheme, a dialect of Lisp. The choice is primarily based on the exceptional ease and flexibility with which Scheme expresses mathematical concepts; good references for the language are [9] and [2].

## Chapter 2

## Ordinary Differential Equations and Manifolds

This chapter describes the computational representation of manifolds, as well as their use in the formulation and numerical solution of ordinary differential equations. As motivating examples for the main definitions, problems in classical mechanics are presented using the manifold formalism. In following chapters, some ideas for integrating linear partial differential equations using multiple coordinate systems are treated.

For a good reference on advanced calculus as well as an elementary introduction to manifolds, Munkres [21] is excellent. Also, Guillemin and Pollack give a beautifully lucid exposition on the topology of manifolds [14]. A more technical and abstract treatment is given in Warner [28], and the classic by Arnol'd [4] presents manifolds in the context of classical mechanics-An approach followed closely in spirit (but only in spirit) in this chapter.

### 2.1 A brief introduction to manifolds

This section introduces the basic notions using a physical example, which will be revisited from time to time as new concepts are developed.

### 2.1.1 The spherical pendulum

A good starting point for the study of manifolds is a variation on the classical pendulum, the spherical pendulum (see Figure 2-1): Suppose a point mass of mass $m$ is connected to a fixed point by a massless rod of length $l$. Furthermore, suppose that the point mass is allowed to move freely in any angle (not simply constrained to a vertical plane, as in the usual pendulum), and that it is subject only to a uniform gravitational field of constant


Figure 2-1: The spherical pendulum.
magnitude $g$.
The equations of motion for this problem are easy to derive. However, instead of deriving the equations to analyze properties of the motion, let us focus on some of the more fundamental issues in a complete mathematical description of the problem. As we will see, this problem illustrates most of the basic ideas in the theory of manifolds. ${ }^{1}$

First, consider the problem of specifying the configuration of the system. What information do we need to specify the position of the pendulum? Since the point mass is constrained to move at a constant distance $l$ from the fulcrum, the problem of specifying configurations of the spherical pendulum is equivalent to the problem of locating points on a sphere.

In order to specify points on a sphere, there are a couple of alternatives. One natural idea is to use the fact that the two-dimensional sphere sits inside three-dimensional Euclidean space, and to use the coordinates of $R^{3}$ to parametrize the sphere. Unfortunately, this approach is natural only for the sphere, and there are many important abstract spaces that cannot be easily visualized as subspaces of Euclidean space, such as the space of all orientations of a rigid body (which will be discussed later). Furthermore, in numerical integrations of ODEs, it will often happen that the trajectory "veers off" the sphere due to round-off error, and the constraint that the point mass lies at a constant distance $l$ from the origin would no longer hold.

Another approach is to put coordinate systems on the sphere that require only two parameters. Formally, these are differentiable one-to-one mappings that map subsets of the sphere onto subsets of the plane and for which a differentiable inverse exists. This turns out

[^1]to be a well-studied problem, since cartographers must deal with the fact that the surface of the Earth is spherical (approximately) but maps are flat (Euclidean). The usual examples of map-making projections, such as the Mercator projection (cylindrical coordinates) or the system of longitudes and latitudes (spherical coordinates) are all examples of coordinate systems on the sphere. Note that there exists no two-parameter coordinate system that covers all of the sphere in a continuous fashion, but for every point on the sphere, we can always find a coordinate system that parametrizes a neighborhood of the point using a pair of parameters, so that the parametrization matches the dimension of the space.

In addition, in cartography, there is a natural solution to the problem that no coordinate system covers all of the Earth: We can simply use more than one map. We can simply switch to another map when one map becomes nearly useless. All that is required is some systematic way of figuring out when coordinates in two different maps are in fact the same point on the sphere, so that one could switch between maps without getting lost. This idea has been generalized beyond recognition to form the foundation of modern differential geometry, and spaces covered by maps (usually called charts) that make the space look "locally Euclidean" are called manifolds.

### 2.1.2 Differentiable manifolds

These ideas can be formulated mathematically as follows: Let $M$ be a non-empty set of points, ${ }^{2}$ and let $n$ be some fixed positive integer. An $n$-dimensional chart on $M$ is a triple ( $U, V, \phi$ ), where $U$ is a subset of $M, V$ an open subset of $R^{n}$, and $\phi$ a one-to-one map of $U$ onto $V$ (see Figure 2-2). $\phi$ is a coordinate map, and a chart $(U, V, \phi)$ is said to contain a point $p \in M$ if $U$ contains $p$. Given two charts $C_{1}=\left(U_{1}, V_{1}, \phi_{1}\right)$ and $C_{2}=\left(U_{2}, V_{2}, \phi_{2}\right)$, suppose the intersection $U_{1} \cap U_{2}$ is non-empty, and let $W_{i}$ be the image $\phi_{i}\left(U_{1} \cap U_{2}\right)$ for $i=1,2$. We can then form a transition map, $\phi_{2} \circ \phi_{1}^{-1}$, which is a bijective mapping from $W_{1}$ to $W_{2}$. Note that the inverse of this transition map is $\phi_{1} \circ \phi_{2}^{-1}$, which is represented by the same set of lines in Figure 2-2 with the arrows reversed. Now, if $W_{1}$ and $W_{2}$ are both open subsets of Euclidean $n$-space, then it makes sense to talk about the differentiability of the transition map $\phi_{2} \circ \phi_{1}^{-1}$, and the charts $C_{1}$ and $C_{2}$ are said to be compatible if $W_{1}$ and $W_{2}$ are open and the corresponding transition map is smooth, i.e. has all orders of partial derivatives. ${ }^{3}$ A collection $\mathcal{A}$ of charts on $M$ is called an atlas if all its charts are mutually compatible, and if every point of $M$ is contained in some chart in $\mathcal{A} .^{4} M$, together with an

[^2]

Figure 2-2: The sphere with generic charts and a transition map.
atlas $\mathcal{A}$, is called a differentiable manifold of dimension n. ${ }^{5}$
These formal definitions may take some time to absorb, but after some thought one should see that all that this says is that $M$ is completely covered by a collection of maps, so that given any point $p \in M$, we can find a chart that makes a neighborhood of $p$ look like an open subset of Euclidean $n$-space. Thus, the definition formalizes the idea of coordinate systems on abstract spaces, and transition maps allow one to switch between coordinate systems in a consistent way. This would, for example, allow us to define spheres in a way that solves the problem of locating points: One simply specifies a chart and a point, provided the appropriate charts have been constructed.

## Implementation of manifolds in Scheme

The implementation of charts as computational objects is straightforward; it is accomplished through the procedure make-simple-chart. Make-simple-chart expects five arguments:
and $C_{2}$ and $C_{3}$ are also compatible, then so are $C_{1}$ and $C_{3}$.
${ }^{5}$ Technically, manifolds are often required to be "second-countable Hausdorff spaces." This is a rather technical condition and has no direct consequences for the material here.

Dim, the dimension of the chart; in-domain?, a procedure that takes a point as argument and returns \#t or \#f depending on whether the given point is in the chart; in-range?, the analogous procedure for coordinate vectors in the range of the coordinate map; coord-map, a computational representation of the function $\phi$ that maps points in the manifold to coordinate vectors in $R^{n}$; and its inverse, inverse-map. The constructor simply packages up these procedures and provides auxiliary procedures for accessing these methods.

Similarly, the procedure make-manifold constructs manifolds. It takes four arguments: Dim, the dimension of the manifold; general-find-chart, a procedure that takes a point $p$ and a list of predicates, and returns a chart $C$ containing $p$ such that every predicate in the list returns \#t when called with $C$; find-minimizing-chart, which takes a point $p$ and a real-valued function $f$ on charts, and returns the chart $C$ that contains $p$ and minimizes $f ;{ }^{6}$ and get-local-atlas, a function that takes a point $p$ and returns the list of all charts containing $p$. Note that, since lists in Scheme must necessarily be finite, this means any atlas constructed this way is locally finite; that is, every point $p$ is contained in only finitely many charts. ${ }^{7}$ However, the fact that everything is implemented procedurally allows for the possibility that the atlas itself is potentially infinite. For convenience, there is also a constructor charts->manifold which takes a finite list of charts and constructs the procedures general-find-chart, etc., by searching through this finite list.

### 2.1.3 Some examples

One obvious class of examples of differentiable manifolds is the Euclidean space $R^{n}$. Here, the atlas consists of a single chart, $\left(R^{n}, R^{n}, i d_{R^{n}}\right)$, where $i d_{R^{n}}$ is the identity map on $R^{n}$.

We can express this example in Scheme as follows:

```
(define (make-euclidean-space dim)
    ;; Just need one big happy chart:
    ;; (test v) = #t iff v is a real vector of length dim:
    (let* ((test (make-euclidean-test dim))
        (chart (make-simple-chart dim test test identity identity)))
        (charts->manifold (list chart))))
```

Another example is the circle, where two charts are now required (see Figure 2-3): Removing the point $(1,0)$ from the circle gives a smooth bijection between the rest of the

[^3]

Figure 2-3: The circle as a manifold.
circle and the interval $(0,2 \pi)$, using the usual angular parametrization. Similarly, removing the point $(-1,0)$ gives a correspondence between the rest of the circle and the interval $(-\pi, \pi)$. These two charts suffice to cover the circle.

We can, in fact, generalize such coordinate systems to higher-dimensional spheres. In dimensions higher than 1 , though, no single choice of charts is completely natural. We could use cylindrical coordinates or spherical coordinates or some other coordinate system. Each choice has its advantage. However, it is not hard to see that we can always choose enough charts to cover all of the sphere. Rather than implementing the circle described above as a special case, here is some code that implements the $n$-dimensional sphere using stereographic projection (see Figure 2-4):

```
;;; Make a chart for the sphere using stereographic projection:
(define (make-stereographic-chart dim pole-dim pole-dir)
    (let* ((ubound 5)
        (dim+1 (+ dim 1))
        (pole (vector:basis dim+1 pole-dim pole-dir)))
    (letrec
        ((in-domain?
            (let ((sphere? (make-imbedded-sphere-test dim)))
```



Figure 2-4: Stereographic projection.

```
            (lambda (v)
            (and (sphere? v)
            (not (almost-equal? (vector:distance^2 v pole) 0))
            (< (- (/ 4 (vector:magnitude^2 (vector:- v pole))) 1)
                ubound)))))
            (in-range?
                (let ((euclidean? (make-euclidean-test dim)))
            (lambda (v)
                    (and (euclidean? v)
                            (< (vector:magnitude^2 v) ubound)))))
                (map
                (lambda (x)
            (let* ((d (vector:- x pole))
                    (y (vector:* (/ 2 (vector:magnitude^2 d)) d)))
                    (vector:drop-coord (vector:+ y pole) pole-dim))))
    (inverse
        (lambda (x)
            (let* ((d (vector:- (vector:add-coord x pole-dim) pole))
                    (y (vector:* (/ 2 (vector:magnitude^2 d)) d)))
            (vector:+ y pole)))))
                (let ((chart (make-simple-chart dim in-domain? in-range? map inverse)))
    (make-spherical-range chart (make-vector dim 0) (sqrt ubound))
    chart))))
;;; Construct the sphere:
(define (make-sphere dim)
    (charts->manifold (list (make-stereographic-chart dim 0 1.)
                            (make-stereographic-chart dim 0-1.))))
```

Stereographic projection works as follows: Let $i$ be an integer between 1 and $n$, and let $p$ be a vector of the form $\pm e_{i}$, where $e_{i}$ is the $i$ th canonical basis vector of $R^{n}$. Then each point $q$ on the sphere is mapped to the plane $\left\{x_{i}=0\right\}$ by defining $x$ to be the point where the straight line joining $p$ and $q$ intersects the plane. This creates a bijection between the set $S^{n}-\{p\}$ and the plane $\left\{x_{i}=0\right\}$, which can be identified with $R^{n-1}$ by dropping the $i$ th coordinate. This defines a chart. The relevant formulae are easy to derive and are left as an exercise for the reader. In the program above, the variable pole-dim represents the


Figure 2-5: A local tangent vector.
index $i$; it is the dimension singled out for defining the point $p$ (which is the vector pole). Pole-dir specifies whether $p$ is $+e_{i}$ or $-e_{i}$.

Notice that it took quite a bit of work to define such a simple manifold; the implementation of spherical coordinates is even more involved. However, the manifold abstraction lets us separate the definition of the actual space from operations we would like to perform on the abstract space, such as integrating a differential equation. It makes these tasks completely independent of each other.

### 2.1.4 Tangent vectors

The manifold construction described above only provides a way for specifying positions of the pendulum. In order to completely capture the dynamical state of the problem, we also need a way to describe the velocity of the point mass.

Consider the evolution of the pendulum: As time goes on, the point mass traces a path on its configuration space, the 2 -sphere. We can describe the position at each instant $t$ by a 3 -vector $\gamma(t)$ whose distance from the origin is the constant $l$. The velocity is then the derivative $\dot{\gamma}$. Since the path is imbedded in the 2 -sphere, $\dot{\gamma}(t)$ must be tangent to the sphere itself. Conversely, if a vector $v$ is tangent to the 2 -sphere at some point $p$, then there exists a smooth path $\gamma$ lying entirely in the sphere such that $\gamma(t)=p$ and $\dot{\gamma}(t)=v$ for some $t$, so that every vector tangent to the 2 -sphere describes the velocity of some possible path of the pendulum. Velocities, then, are naturally described by vectors tangent to the configuration manifold, and we can define velocities for arbitrary configuration spaces by generalizing tangent vectors to manifolds.

We can arrive at a general definition of tangent vectors on manifolds as follows: First,
note that if we are given a chart $C=(U, V, \phi)$, then locally a tangent vector at $p \in U$ can be represented by a vector $v$ "anchored" to the coordinate vector $x=\phi(p)$ in the chart (see Figure 2-5). We call the object ( $C, p, v$ ) a local tangent vector. Now, in order for tangent vectors to be coordinate-independent, there must be a consistent way of transforming them between charts, and locally they must always behave like the derivatives of paths. That is, if $C_{1}, C_{2}$, and $C_{3}$ are overlapping charts, and $\gamma$ is a path on $M$, then there should be locally-defined transformations $T_{i j}$ such that $\phi_{j} \circ \gamma=T_{i j} \circ \phi_{i} \circ \gamma$, and such that the derivative of $\phi_{i} \circ \gamma$ in $C_{i}$ is carried to the derivative of $\phi_{j} \circ \gamma$ in $C_{j}$. This requires that applying $T_{12}$ to some vector $v$, followed by $T_{23}$, yields the same result as applying $T_{13}$ directly. In view of the chain rule, the transformation $T_{i j}$ must be the transformation represented by the Jacobian matrix $D\left(\phi_{j} \circ \phi_{i}^{-1}\right)$ of the transition map. Thus, we can say two local tangent vectors $\left(C_{1}, p, v_{1}\right)$ and $\left(C_{2}, p, v_{2}\right)$ at $p$ are equivalent if $D\left(\phi_{2} \circ \phi_{1}^{-1}\right)(x) \cdot v_{1}=v_{2}$, where $x=\phi_{1}(p)$. The tangent vector corresponding to a given local tangent vector ( $C, p, v$ ) can then be defined as the set of all local tangent vectors equivalent to $(C, p, v)$. The space of all tangent vectors at a given point $p$ is the tangent space of $M$ at $p$, denoted by $T_{p} M$. The union of all tangent spaces is denoted by $T M$ and is called the tangent bundle.

This construction defines tangent vectors as equivalence classes. Now, each of these equivalence classes, and hence each tangent vector, can be in fact a rather large set of local tangent vectors. ${ }^{8}$ While this may seem too abstract to be useful, one should realize here that any local tangent vector in the equivalence class can be used to represent the tangent vector, and the important thing is that there is a consistent rule for transforming local tangent vectors between charts. Similarly, the intrinsic structure of the manifold arises from the way charts overlap, and whether or not the manifold happens to be a subspace of Euclidean space is of secondary importance. In fact, as stated before, there are many important examples of manifolds that are most naturally defined in ways that make them hard to describe as subsets of Euclidean spaces, although in principle this can always be done. ${ }^{9}$

One last remark: A manifold as we have defined it has an intrinsic notion of smoothness, but has no intrinsic notions of distance or size. The property of smoothness is stronger than that of continuity, but not as strong as having a metric for measuring the distance between points. Thus, our constructions in this section have shown that the idea of tangent spaces

[^4]is really a property of the differentiable structure of the manifold (i.e. its atlas), and not a metric property. A manifold where a particular metric is defined is called a Riemannian manifold; the definition of such a metric relies on defining inner products in a smooth way on the tangent spaces of a manifold, using the same methods that we have been using. They are important in applications of differential geometry to physics, but will not be needed in this chapter.

## Tangent vectors in Scheme

The implementation of tangent vectors is easy. The constructor make-tangent simply packages up the structures for defining a local tangent vector into a convenient Scheme object:

```
(define (make-tangent chart p v)
    ;; p is the (abstract) point to which v is tangent, and v is the *coordinate
    ;; representation* of the tangent vector in the coordinates provided by the
    ;; given chart.
    (vector 'tangent chart p v))
```

Though it is not necessary for later work, it is instructive to consider the tangent space as a vector space. For example, how does one define addition on the tangent space $T_{p} M$ ? One can define vector addition for tangent vectors as follows:

```
;;; Add two tangent vectors:
(define (tangent:+ v w)
    (let ((p (tangent:get-anchor v))
            (q (tangent:get-anchor w)))
        (if (equal? p q)
            (let ((chart (tangent:get-chart v)))
                (make-tangent chart
                            p
                                    (vector:+ (tangent:get-coords v)
                                    (chart:push-forward w chart))))
            (error "Cannot add vectors tangent to different points."))))
;;; Push a tangent vector along a chart:
(define (chart:push-forward tv chart)
    (let ((other (tangent:get-chart tv))
            (v (tangent:get-coords tv)))
        (if (eq? chart other)
            v
            (push-forward-in-coords
                (chart:make-transition-map other chart)
                (chart:point->coords (tangent:get-anchor tv) other)
                    v))))
(define (push-forward-in-coords f x v)
    (((diff f) x) v))
```

The expression (chart:push-forward w chart) computes the image of the tangent vector w under the transition map $\phi_{2} \circ \phi_{1}^{-1}$, and (( (diff f) $x$ ) v) applies the Jacobian matrix of $f$ at $x$ to the vector $v$. The procedure tangent:get-anchor extracts the point $p$, which we call the anchor of the tangent vector, from the local tangent vector ( $C, p, v$ ). Other operations on tangent vectors can be defined in a similar fashion, and scalar multiplication is even simpler:

```
(define (tangent* a v)
    (make-tangent (tangent:get-chart v)
                            (tangent:get-anchor v)
                            (vector:* a (tangent:get-coords v))))
```


### 2.1.5 Smooth maps and differentials

Having defined differentiable manifolds, the next natural step is to see how of the usual notions of the calculus carry over. For the sake of simplicity, only the concepts of differential calculus are discussed in this section; a discussion of integration on manifolds would take us too far afield and is thus postponed until the next chapter, where integration becomes a necessary tool.

Recall that in the definition of tangent vectors, charts were used to make the manifold look locally like Euclidean space, where tangent vectors are well-defined. We can define differentiable functions analogously. Let $M$ and $N$ be two differentiable manifolds, and let $f$ be a function from $M$ to $N$. Let $p$ be any point of $M$ and $q=f(p) \in N$. Then $f$ is smooth or differentiable if for every chart ( $U, V, \phi$ ) containing $p$ and every chart ( $U^{\prime}, V^{\prime}, \phi^{\prime}$ ) containing $q$, the function $\phi^{\prime} \circ f \circ \phi^{-1}$ mapping $V$ to $V^{\prime}$ is smooth; that is, if $\phi^{\prime} \circ f \circ \phi^{-1}$, as a mapping from one subset of an Euclidean space into another, has all orders of partial derivatives. By extension, $f$ is a real-valued smooth function on $M$ if it is smooth as a map from $M$ into the manifold $R$, where $R$ is given the canonical atlas $\left\{\left(R, R, i d_{R}\right)\right\}$, and smoothness is defined as above. It is easy to verify that when $M$ is $R^{n}$, this definition of smoothness is equivalent to the usual one.

Let us now consider the idea of derivatives. As we saw in the discussion of tangent vectors in 2.1.4, derivatives of transition maps provide a natural way to transform tangent vectors from one coordinate system to another. Generalizing this observation, we can say that derivatives of smooth maps between manifolds should transport tangent vectors from one tangent space to another. This is, in fact, not that different from the use of gradients in vector calculus: The directional derivative of a real-valued function is the dot product of its gradient and a unit vector in some given direction. Furthermore, if $v$ is the value of the gradient of a function at some point, and $w$ is a vector, then mapping $w$ by the linear transformation $A$ to the vector $A w$ while keeping $v \cdot w$ an invariant quantity requires that $v$ be mapped to $v A^{-1}$. Thus, coordinate representations of gradients actually change by a
transformation opposite that of vectors. This shows that derivatives evaluated at a given point are not vectors, but are linear functionals. This is exactly the kind of duality captured by the use of row and column vectors in elementary calculus.

More formally, let $M$ and $N$ be differentiable manifolds, and let $f$ be a smooth function from $M$ to $N$. Let $p$ be any point in $M$, and let $q=f(p)$. Consider the map that takes a local tangent vector $(C, p, v), C=(U, V, \phi)$, to $\left(C^{\prime}, q, w\right), C^{\prime}=\left(U^{\prime}, V^{\prime}, \phi^{\prime}\right)$, where

$$
\begin{equation*}
w=D\left(\phi^{\prime} \circ f \circ \phi^{-1}\right)(\phi(p)) \cdot v \tag{2.1}
\end{equation*}
$$

One can easily check that if two local tangent vectors represent the same tangent vector in $T_{p} M$, then their images under this map also represent the same tangent vector in $T_{q} N$. Thus, the mapping can be used to define a map $d f_{p}$ from the tangent space $T_{p} M$ to $T_{q} N$. Furthermore, one could see from the definition that the map is linear on local tangent vectors in the same chart, and hence $d f_{p}$ is a linear transformation between tangent spaces as well. The function $d f$ that assigns to each point $p$ the linear transformation $d f_{p}$ is the differential of $f$.

Note that, in this notation, the chain rule can be stated very simply:

$$
\begin{equation*}
d(g \circ f)_{p}=d g_{q} \circ d f_{p}, \tag{2.2}
\end{equation*}
$$

where $q=f(p)$. This simply restates the usual chain rule while making the role of the differential as a mapping between tangent spaces explicit.

## Computing differentials of smooth maps

The implementation of smooth maps is complicated by the implementation of differentiation in Scheme. ${ }^{10}$ As a result, the constructor make-smooth-map takes four arguments: A

[^5]manifold, domain; another manifold, range; a procedure that actually computes the function, point-function; and a procedure that constructs transition maps, make-transition. However, for most purposes, smooth maps can be constructed using make-simple-map, which only needs the first three arguments and requires that point-function is a composition of primitive Scheme functions. Another procedure, make-real-map, is also provided for convenience; it packages a real-valued function on a manifold into a smooth-map structure. Smooth maps cannot be called directly as functions, but may be applied using apply-smooth-map.

Here are some examples that will become useful when we discuss Lagrangian mechanics:

```
;;; Euclidean 3-space...
(define R^3 (make-euclidean-space 3))
;;; And its tangent bundle.
(define TR^3 (make-tangent-bundle R^3))
;;; The Lagrangian for a particle traveling in a uniform graviational field.
;;; It's just the difference between the kinetic energy, 1/2*|v/~
;;; potential energy, z, where v is the velocity and p = ( }x,y,z)\mathrm{ is the
;; position (in 3-space) of the point mass (assume m = l = 1).
(define falling-lagrangian
    (make-real-map
        TR^3 (lambda (p)
            (- (* 1/2 (vector:magnitude^2 (tangent:get-coords p)))
                            (vector-third (tangent:get-anchor p))))))
```

;; This restricts the Lagrangian above to the unit sphere, effectively forming
;;; a Lagrangian for the spherical pendulum.
(define S~2 (make-sphere 2))
;;; Define the identity map from the 2-sphere into 3-space, then differentiate
;; it to extend the function to the tangent bundle.
(define spherical-inclusion
(smooth-map:diff (make-simple-map $S^{\sim} 2 R^{\wedge} 3$ identity)))
;; This composition restricts the domain of the Lagrangian to the 2-sphere.
(define spherical-lagrangian
(smooth-map:compose falling-lagrangian spherical-inclusion))

Here's an example of how the function can be used:
; ; ; The tangent bundle of the sphere is the state space of the spherical
;;; pendulum:
(define TS~2 (make-tangent-bundle S^2))

```
;;; Define the south pole of the sphere.
(define p (vector 0 0 -1))
;Value: p
;;; Find a chart.
(define chart (manifold:find-chart S^2 p))
;Value: chart
;;; Make a tangent vector.
(define v (make-tangent chart p (vector 0 1)))
;Value: v
;;; Compute the Lagrangian. Note that, because Euclidean spaces are all
;;; constructed using a single procedure, elements of R^1 are actually vectors
;;; containing a single element, *not* real numbers (as is customary).
(apply-smooth-map spherical-lagrangian v)
;Value 61: #(1.5)
;;; Find a chart for the tangent vector itself in the tangent bundle.
(define another-chart (manifold:find-chart TS^2 v))
;Value: another-chart
;;; Make a tangent vector (this object lives in T(TS~2)).
(define w (make-tangent another-chart v (vector 0 0 1 0)))
;Value: w
;;; Apply the differential of the Lagrangian:
(define u (apply-smooth-map (smooth-map:diff spherical-lagrangian) w))
;Value: u
;; u should be an object in TR. Its anchor is the value of the Lagrangian at
;;; v.
(tangent:get-anchor u)
;Value 67: #(1.5)
(tangent:get-coords u)
;Value 68: #(0.)
```


### 2.1.6 Tangent bundles

A useful thing to notice, at this point, is that the tangent bundle is itself a differentiable manifold. More precisely, if $M$ is an $n$-manifold, then $T M$ is an $2 n$-dimensional manifold. To see this, suppose $C=(U, V, \phi)$ is a chart on $M$. Then we can define the chart $T C=$ ( $T U, V \times R^{n}, \psi$ ), where $T U$ (by an abuse of notation) denotes the union of the tangent spaces $T_{p} M$ for which $p \in U$, and is hence an open subset of $T M$, and $\psi$ is the map defined
by:

$$
\begin{equation*}
\psi(C, p, v)=\left(\phi(p), d \phi_{p}(v)\right) \tag{2.4}
\end{equation*}
$$

where ( $C, p, v$ ) is a local tangent vector in $C$. The expression $d \phi_{p}(v)$ makes sense because $V$ is an open subset of $R^{n}$, and we can thus treat $\phi$ as a smooth map between manifolds and compute its differential. Furthermore, the tangent space of $V$ is trivially equal to $R^{n}$ at each point, so the dimension of $T M$ is twice the dimension of $M$. The chart $T C$ is called a tangent chart, and the tangent bundle $T M$ is given the atlas consisting of the set of all tangent charts.

## Implementation in Scheme

The construction of tangent bundles builds on tangent vectors, and the most important part is the construction of tangent charts:

```
;;; Construct a tangent chart:
(define (make-new-tangent-chart chart)
    ;; First, extract some useful information from CHART:
    (let* ((dim (chart:dimension chart))
            (2*dim (* 2 dim))
                (in-M-domain? (chart:get-membership-test chart))
                (in-M-range? (chart:get-range-test chart))
                (M-map (chart:get-coord-map chart))
                (M-inverse (chart:get-inverse-map chart))
                (dim-vector? (make-euclidean-test dim))
                (2*dim-vector? (make-euclidean-test 2*dim)))
        (letrec
            ((in-domain?
                (lambda (v)
                        (and (in-M-domain? (tangent:get-anchor v))
                            (dim-vector? (tangent:get-coords v)))))
                (in-range?
                    (lambda (v)
                        (and (2*dim-vector? v)
                            (in-M-range? (vector-head v dim)))))
                (coord-map
                (lambda (v)
                        (vector-append (M-map (tangent:get-anchor v))
                                    (chart:push-forward v chart))))
        (inverse-map
            (lambda (x)
```

```
        (make-tangent chart
            (M-inverse (vector-head x dim))
                            (vector-end x dim))))
(transition
    (lambda (Tother)
        (let* ((other (chart:get-base-chart Tother))
                (f (chart:make-transition-map chart other)))
            (lambda (x)
                (let ((anchor (vector-head x dim))
                    (tangent (vector-end x dim)))
                    (vector-append (f anchor)
                            (push-forward-in-coords
                            f anchor tangent))))))))
(let ((new-chart (make-chart 2*dim in-domain? in-range?
                    coord-map inverse-map transition)))
    ;; Some auxiliary information:
    (chart:install-extra new-chart 'base-chart (delay chart))
    (chart:install-extra chart 'tangent-chart (delay new-chart))
    new-chart))))
```

This procedure can then be used to construct tangent bundles.

### 2.1.7 Making new manifolds

As noted in the previous section, the tangent bundle of a manifold is also manifold. This gives us a way to construct new manifolds out of old ones. In this section, we will take a look at a few other ways of constructing new manifolds out of existing ones.

Product manifolds. First, consider two manifolds $M$ and $N$. Let $(U, V, \phi)$ be a chart on $M$, and let $\left(U^{\prime}, V^{\prime}, \phi^{\prime}\right)$ be a chart on $N$. The product chart associated with the two charts is the chart $\left(U \times U^{\prime}, V \times V^{\prime}, \phi \times \phi^{\prime}\right)$, where $U \times U^{\prime}$ is the Cartesian product $\{(x, y): x \in$ $\left.U, y \in U^{\prime}\right\}, V \times V^{\prime}$ is similarly defined, and $\phi \times \phi^{\prime}$ is the map taking $(x, y) \in U \times U^{\prime}$ to $\left(\phi(x), \phi^{\prime}(y)\right) \in V \times V^{\prime}$. The product manifold $M \times N$, then, is the manifold whose space is the Cartesian product of the spaces $M$ and $N$, and whose atlas is given by the set of all product charts. If the dimension of $M$ is $m$ and that of $N$ is $n$, then the dimension of $M \times N$ is $m+n$. For example, the Euclidean space $R^{n}, n>1$, may be constructed as a product manifold $R^{n-1} \times R$, and the torus can be thought of as the product manifold $S^{1} \times S^{1}$ (where $S^{n}$ denotes the $n$-dimensional sphere, and hence $S^{1}$ is the circle).

Cotangent bundles. Recall now that every vector space has a dual space of linear functionals. Thus, to every tangent space $T_{p} M$, we can find its dual vector space $T_{p}^{*} M$. It turns out that the union $T^{*} M$ of all dual spaces $T_{p}^{*} M$ is also a differentiable manifold, by using a construction similar to that of the tangent bundle. The space $T^{*} M$ is called the cotangent bundle of the manifold, and is just as important geometrically as the tangent bundle, if not
more so. In classical mechanics, the cotangent bundle of a configuration space is called its phase space. Whereas the state space describes a system by its position and velocity, the phase space describes a system by its position and generalized momentum.

The inverse function theorem. Finally, there is a method of constructing manifolds that is very useful theoretically, but practically useless for computation: The inverse function theorem. Briefly, it states that if $f$ is a smooth map from $M$ into $N$, the dimension of $N$ is less than the dimension of $M$, and for some point $q \in N$, every $p \in M$ such that $f(p)=q$ has a surjective differential $d f_{p}$, then the inverse image $f^{-1}(q)=\{p \in M: f(p)=q\}$ is a smooth manifold. Furthermore, if the dimension of $M$ is $m$ and that of $N$ is $n$, then the dimension of this new manifold is $m-n$. For theoretical purposes, this is a very useful way of constructing manifolds, especially for describing constraints in mechanical systems. For example, the configuration space for a free particle is $R^{3}$, and if one were to enforce the constraint that the particle must stay at a constant distance $l$ from the origin, this theorem immediately tells us that the resulting space (the sphere, in this case) is a differentiable manifold. However, the proof of this theorem involves some non-constructive arguments, and hence it cannot be used directly for computation. The efficient computation of general inverses of functions is, at the present, not possible.

Since most of these constructions (except the cotangent bundle) will not be used directly in later sections, their implementation will not be discussed here. The cotangent bundle will appear again when we discuss the Hamiltonian approach to mechanics.

### 2.1.8 Boundaries

Our definition of manifolds does not allow for spaces with boundaries. For example, notice that the unit disc

$$
\begin{equation*}
\left\{(x, y) \in R^{2}: x^{2}+y^{2} \leq 1\right\} \tag{2.5}
\end{equation*}
$$

is not a manifold by our definition, because the points $(x, y)$ for which $x^{2}+y^{2}=1$ (that is, those lying on the boundary of the disc) do not have neighborhoods that "look like" open subsets of $R^{2}$. However, it locally have the structure of an Euclidean half-space (see Figure 2-6). Since boundaries are often useful in applying partial differential equations to model physical systems, this section takes a closer look at this concept.

In order to describe manifolds with boundaries, a new type of chart is necessary. First, some definitions: Given an Euclidean space $R^{n}$, let $R_{+}^{n}$ be the half-space

$$
\begin{equation*}
R_{+}^{n}=\left\{x \in R^{n}: x_{n} \geq 0\right\} \tag{2.6}
\end{equation*}
$$



Figure 2-6: A boundary chart.
where $x_{n}$ denotes the $n$th component of the $n$-vector $x$. A boundary chart, depicted in Figure 2-6, is then a triple ( $U, V, \phi$ ), where $U$ is a subset of $M, V$ is the (non-empty) intersection of some open subset $V^{\prime}$ of $R^{n}$ with the half-space $R_{+}^{n}$, and $\phi$ is a bijection between $U$ and $V$. The usual definition of compatibility between charts still applies to boundary charts, although what it means to be differentiable at the boundary $\left\{x_{n}=0\right\}$ requires more careful analysis (omitted here).

Now suppose $M$ is an arbitrary set, and extend the definition of atlases to allow boundary charts. A set $M$ is a manifold with boundary if it has an atlas $\mathcal{A}$ with mutually compatible charts and boundary charts. If a point $p$ has the property that for some boundary chart $(U, V, \phi), x_{n}=0$, where $x=\phi(p)$, then $p$ is said to lie on the boundary of the manifold $M .{ }^{11}$ The boundary of a manifold $M$ is usually denoted by $\partial M$, and consists of the set of points that lie on the boundary of some boundary chart. It is easy to verify that the boundary of a manifold is itself a manifold without boundary. ${ }^{12}$

The computational implementation of boundaries is not used in the rest of this chapter, so its discussion is postponed until it is needed in the next chapter on partial differential equations and boundary value problems.

### 2.2 Vector fields and differential equations

### 2.2.1 Smooth vector fields

The tangent bundle construction actually facilitates the definition of smooth vector fields: Let $\pi$ denote the projection map from $T M$ into $M$, defined by:

[^6]\[

$$
\begin{equation*}
\pi(p, v)=p \tag{2.7}
\end{equation*}
$$

\]

That is, the projection map $\pi$ extracts the "anchor" of the tangent vector, much like the procedure tangent:get-anchor. A smooth vector field on $M$ is then a smooth map $v$ from $M$ into $T M$, such that for every point $p \in M$, the equation $\pi(v(p))=p$ holds.

It is easy to verify that, over each chart, a smooth vector field as defined here corresponds to what one usually means by a smooth vector field. Thus, the usual local existence and uniquness theorems apply. This abstraction lets one define systems of first-order ordinary differential equations, and higher-order equations are typically handled by using the tangent bundle construction. A second-order equation, for example, can be thought of as a vector field on the tangent bundle, and so on. This is why mathematical descriptions of mechanics problems involve vector fields (first-order equations) on tangent or cotangent bundles of manifolds.

### 2.2.2 Flows generated by smooth vector fields

How can we integrate ODEs on manifolds? Since within each chart ( $U, V, \phi$ ), the manifold "looks like" Euclidean space, the obvious thing to try is to use the coordinate map $\phi$ to "push" vector field onto the Euclidean subspace $V$. More precisely, suppose we are given a tangent vector that is represented by the local tangent vector ( $C^{\prime}, p, v^{\prime}$ ), and wish to map this local tangent vector over to the chart in which we are integrating the equations, $C=(U, V, \phi)$. Then we can simply apply the Jacobian of the transition map, to obtain $(C, p, v)$, where $v$ is defined by:

$$
\begin{equation*}
v=D\left(\phi \circ \phi^{\prime-1}\right)\left(\phi^{\prime}(p)\right) \cdot v^{\prime} \tag{2.8}
\end{equation*}
$$

This consistently transforms the local tangent vector to the other chart. Thus, a smooth vector field on $M$ can always be turned into a local vector field on the open subset $V$ of Euclidean $n$-space, for which there exist numerous methods of integration.

The computational implementation of ODE integrators on manifolds, however, requires that we consider a few more issues. For example, for the sake of flexibility and efficiency, it is easier to implement vector fields directly as procedures which return local vector fields when given a chart, rather than a procedure that actually returns a local tangent vector representation of some tangent vector every time whenever it is given a point on the manifold. This is because, in some situations, it may be easier for procedures that compute vector fields to use internal representations that are not in the form of local tangent vectors. If the procedure must convert its internal representation to a local tangent vector,


Figure 2-7: When should the ODE integrator switch charts?
as the integrator requires, it might as well directly convert it to the current chart. This structure gives procedures this flexibility, as will be demonstrated in later examples.

A more serious issue is that of switching between charts (see Figure 2-7): As the integrator moves along in one chart, taking discrete steps forward in time, it will eventually step off the chart. One solution is to always watch where the next step "lands" before actually committing to it, and to switch charts if the next step is outside the current chart. This approach has the problem that when switching charts, one needs to keep track of which charts have already been visited so that the integrator does not enter an infinite loop, idly switching from one chart to another without making progress. However, this introduces quite a bit of complexity into the integrator, and did not seem to be the best design for a first attempt.

There is, in fact, a more elegant solution to the problem of switching charts: Simply evolve the trajectory in all possible charts! This solution requires that the atlas be locally finite-For every point $p$, there must be only finitely many charts in $\mathcal{A}$ that contain $p$. This is not an overly restrictive requirement, and in general it is easy for the user to control the amount of overlap between charts when constructing them, so that there is not too much overhead in the multiple evaluation. This is the strategy finally chosen, and the main idea is expressed in the code below: ${ }^{13}$

```
;;; This is a simple description of the integration algorithm for ODEs on
;;; manifolds:
(define (v.field->flow manifold make-local-field next-step error-est)
    ;; Integrate the ODE starting at pO, with time index running from t0 to t1:
```

[^7]```
(lambda (p0 t0 t1)
    (let loop ((p p0) (t t0))
        (if (<= t t1)
            ;; Compute the possible next steps, then choose the one that
            ;; minimizes the error estimator, ERROR-EST.
            (let* ((charts (manifold:get-local-atlas manifold p))
                (p1 (minimize-function-over-list
                (compose error-est integrator:get-new-x)
                        (map (lambda (chart)
                                    (next-step (chart:point->coords p chart)
                                    (make-local-field chart)))
                    charts)
                charts)))
                    ;; If the local integrator can step forward in at least one chart,
                    ;; then we can continue:
                    (if p1
                            (loop (integrator:get-new-x p1) (+ t (integrator:get-dt p1)))
                            (error "Ran out of charts!")))))))
```

Notice a few things about this code: First, it takes four arguments: Manifold is just the domain of the ODE; make-local-field is the local vector field constructor, as described before; next-step is a local ODE integrator, a procedure that knows nothing about the manifold but can numerically solve a given ODE in Euclidean coordinates to produce a new coordinate vector; and error-est, a function for estimating the local numerical error.

Notice, first, that the integrator has no built-in notions of step size. It simply relies on the local integrator to supply both a new step and a step size. This facilitates the use of variable-step-size integrators, which can be more efficient and numerically robust. Second, it requires an error estimator that helps it choose from among the guesses supplied by the different charts. This is an advantage of this method: Because of truncation and round-off errors, numerical computations are not actually coordinate-independent. Thus, this integrator allows local error analysis, which improves accuracy greatly, especially in the presence of coordinate singularities (discussed in §2.3.3).

### 2.2.3 Manifolds and classical mechanics

There are several reasons why the manifold abstraction is especially suited to dealing with ordinary differential equations. First, notice that a classical $n$-particle system is described by the configuration space $R^{3 n}$, since each particle has three coordinates. Nontrivial manifolds arise in classical mechanics from constraints, such as the constraint that a point mass lies at a constant distance $l$ from the origin (which yields the spherical pendulum). Now, as noted in §2.1.1, in the traditional approach of modeling the manifold as a subset of a larger Euclidean space and integrating the ODEs in the larger space, trajectories can
sometimes go off the manifold because of the accumulation of round-off and truncation errors. Thus, physical constraints are not enforced faithfully in this classical approach, whereas the manifold abstraction helps minimize this kind of error. Second, in generating local vector fields, it is useful to have explicit formulas. It is rather tedious, in general, to derive differential equations that describe complex physical systems in different coordinate systems. However, in classical mechanics, one could always use variational methods to derive the equations of motion in different coordinate systems with the aid of computer algebra, which is often easier than transforming second-order equations between coordinate systems. ${ }^{14}$

Furthermore, in classical mechanical systems, the error-est function above can be implemented rather easily: Instead of checking the local numerical properties of the chart, one can exploit the existence of conserved quantities, such as energy and momentum. This has the advantage that these quantities are often easy to compute, and systems in classical mechanics usually have a sufficient number of such conserved quantities that one could simply check their deviations from initial values as time marches forward to determine how well the integrator is doing. ${ }^{15}$

[^8]\[

$$
\begin{equation*}
D\left(\partial_{\dot{x}} L \circ \gamma\right)=\partial_{x} L \circ \gamma, \tag{2.9}
\end{equation*}
$$

\]

where $\partial_{\dot{x}}$ denotes differentiation with respect to the velocity part of the Lagrangian, $\partial_{x}$ denotes differentiation with respect to the position part of the Lagrangian, and $D$ is the operator that differentiates real-valued functions of one real variable (in this case time). Equation (2.9) is known as the Euler-Lagrange equation, and gives a system of second-order equations that determine the stationary path. It can be deduced by using the same technique as in the derivation of Equation (4.19).

This provides an easy way to change coordinate systems: Simply substitute the new coordinates into the Lagrangian, simplify the resulting expression, and derive the Euler-Lagrange equations for the new coordinate system. For more information on this topic, see Arnol'd [4].
${ }^{15}$ It is also possible to enforce the conservation laws as constraints, so that one integrates the equations of motion on submanifolds of the state space. While this would ensure that the conservation laws are satisfied exactly (up to round-off error), it also makes checking the accuracy of solutions a little harder-Because the conservation laws were "used up" as constraints, one would now need to perform numerical error analysis to estimate the accuracy of the numerical integration.

## Lagrangian mechanics

Although it is extremely inefficient, one can in fact implement Lagrangian mechanics directly using our Scheme system:

```
;;; The Lagrangian should be a smooth map from the tangent bundle of some
;;; manifold into the real line.
;;; This is very slow, as every evaluation of the field involves a matrix
;;; inversion. Which is why Hamiltonians are *better*, even for comuptational
;;; purposes!
(define (lagrangian->v.field L)
    (let ((TM (smooth-map:get-domain L))
            (R (smooth-map:get-range L)))
        (lambda (p)
            (let ((U
                (if (tangent? p)
                            (make-tangent-chart (tangent:get-chart p))
                            (manifold:find-best-chart TM p))))
                (let ((f (smooth-map:make-transition
                    L U (car (manifold:get-finite-atlas R))))
                            (x (chart:point->coords p U)))
                    (let ((v (vector-tail x (/ (vector-length x) 2))))
                        (let ((E-L (euler-lagrange-in-coords f x)))
                        (let ((A (car E-L))
                        (B (cadr E-L))
                            (c (caddr E-L)))
                            (let ((accel (matrix:solve-linear-system
                                    A
                                    (vector:+ (apply-linear-transformation B v) c))))
                                    (make-tangent U p (vector-append v accel)))))))))))
;;; Derive the Euler-Lagrange equations for fat x (in coordinates) in the form
```

```
        ;; Next, compute the rest of the terms involving the partials of the
        ;; Lagrangian with respect to the positions (note the minus sign):
        (do ((j O (+ j 1)))
        ((>= j n))
    (let ((val (- (vector-first (((pdiff j) ((pdiff i) f)) x)))))
        (matrix-set! B p j val)))
        ;; And then there's the term due to the derivative of the Lagrangian with
;; respect to the position variables:
(vector-set! c p (vector-first (((pdiff p) f) x))))
```

(list A B c)))

```
;;; In many mechanics problems, it's natural to check conservation laws:
(define (check-vector-conservation-law quantity ref-point)
    (let ((ref (quantity ref-point)))
        (lambda (chart tangent)
            (vector:distance (quantity (tangent:get-anchor tangent)) ref))))
```

The cost of inverting the matrix (when matrix:solve-linear-system is called) makes this a prohibitively slow way to compute vector fields, but it does work.

## Hamiltonian mechanics

A slightly more efficient form of automatically generating vector fields is provided by the Hamiltonian point of view. ${ }^{16}$ It can be implemented much more directly:

```
;;; The Hamiltonian should be a smooth map from the cotangent bundle of some
;;; manifold into the real line.
(define (hamiltonian->v.field H)
    (let ((T*M (smooth-map:get-domain H))
            (R (smooth-map:get-range H)))
        (lambda (p)
            (let ((U (manifold:find-best-chart T*M p)))
                (make-tangent U p
                            (hamilton-in-coords
                            (smooth-map:make-transition
```

[^9]H U (car (manifold:get-finite-atlas R)))
(chart:point->coords p U)))))))

```
;;; Derive Hamilton's equations for f at x (in coordinates):
(define (hamilton-in-coords f x)
    (let* ((2n (vector-length x))
            (v (make-vector 2n))
            (n (/ 2n 2)))
        (do ((i n (+ i 1))
            (j O (+ j 1)))
            ((>= j n) v)
        (vector-set! v i (- (vector-first (((pdiff j) f) x))))
        (vector-set! v j (vector-first (((pdiff i) f) x))))))
```

However, this is still rather inefficient due to the evaluation of the partial derivatives. In the numerical experiments that follow, the appropriate vector fields are pre-computed for each chart in the relevant manifold.

### 2.3 Numerical experiments

Finally, this section presents the results of three numerical experiments.

### 2.3.1 The circle field

The first example is a simple integration around a circle. The vector field simply consists of unit vectors going counter-clockwise around the circle, and the trajectories of this system of equations are simply unit-velocity curves around the circle:

$$
\begin{equation*}
\gamma(\theta)=\left(\cos \left(\theta-\theta_{0}\right), \sin \left(\theta-\theta_{0}\right)\right), \tag{2.11}
\end{equation*}
$$

where the phase shift $\theta_{0}$ comes from the initial condition.
This can be implemented easily as follows:

```
;;; First, construct the circle:
(define circle (make-sphere 1))
;;; Here's a trivial vector field on the circle:
(define (circle-field p)
    (let ((x (vector-ref p 0))
                (y (vector-ref p 1)))
        ;; IMBEDDING->TANGENT takes an imbedded tangent vector to the tangent
        ;; bundle of the given (imbedded) manifold.
```

```
    (imbedding->tangent circle p (vector (- y) x))))
;;; Integrate the ODE:
(define circle-path
    (v.field->flow circle
                    (v.field->local-field-maker circle-field)
                            (make-rk4-integrator (* 2 pi 1e-3))
                    ;; LOCAL-DISTORTION checks the numerical error in the current
                    ;; chart.
                            local-distortion))
;;; The real answer (with no phase shift):
(define (real-circ t)
    (vector (cos t) (sin t)))
;;; Here is a test run: After 2*pi seconds, the path should end up where it
;;; started. Let's compare the results of using the manifold and using the
;;; tranditional approach:
(define result (circle-path (vector 1 0) (* 2 pi)))
;Value: result
;;; RESULT is a list of pairs of the form (time-index position), sorted in
;;; *descending* order by time index. Thus, (CAAR RESULT) returns the final
;;; time index, and (CADAR RESULT) returns the final position.
;;; The difference in time index:
(abs (- (caar result) (* 2 pi)))
;Value: 1.127986593019159e-13
;;; The difference in position:
(vector:distance (cadar result) (vector 1 0))
;Value: 4.447015332496363e-14
;;; Here is the more tranditional approach: Simply embed the circle in the
;;; plane, and integrate in two real variables (and hope the trajectory
;;; actually stays on the circle):
(define (traditional-circle-field p)
    (let ((x (vector-ref p 0))
            (y (vector-ref p 1)))
        (vector (- y) x))
;Value: traditional-circle-field
(define traditional-result
    (let ((next-step (make-rk4-integrator (* 2pi 1e-3))))
        (let loop ((t 0) (x (vector 1 0)) (result '()))
            (if (<= t 2pi)
                    (let* ((new (next-step x traditional-circle-field (lambda () #f)))
                        (dt (integrator:get-dt new))
```

```
            (new-x (integrator:get-new-x new)))
        (loop (+ t dt) new-x (cons (list t x) result)))
        result))))
;Value: traditional-result
;;; The error in time index is the same:
(abs (- (caar traditional-result) 2pi))
;Value: 1.127986593019159e-13
;;; The error in position is actually larger: This is because, as stated
;;; before, the traditional method allows the trajectory to veer off the
;;; circle, whereas the manifold approach enforces the constraint strictly.
(vector:distance (cadar traditional-result) (vector 1 0))
;Value: 8.16059276567945e-11
```

Notice that the manifold approach actually produced a more accurate "walk" around the circle!

### 2.3.2 The spherical pendulum

The next example is the one we started out with: The spherical pendulum. As opposed to our previous example, this one actually comes from a physical problem. Furthermore, this particular problem can be understood analytically, so that the motion generated by the integrator can be checked closely for consistency with the actual physical situation.

For this integration, the integration is done on the phase space (the cotangent bundle of the sphere). The vector field could very well have been generated using the following Hamiltonian:

```
;;; The phase space:
(define T*R^3 (make-cotangent-bundle R^3))
;;; The Hamiltonian for a point mass in a uniform gravitational field:
(define falling-hamiltonian
    (make-real-map
        T*R^3 (lambda (p)
                        (+ (* 1/2 (vector:magnitude^2 (cotangent:get-coords p)))
                        (vector-third (cotangent:get-anchor p))))))
;;; Define the Hamiltonian:
(define T*S"2 (make-cotangent-bundle S"2))
(define spherical-inclusion*
    (let* ((chart (car (manifold:get-finite-atlas R^3)))
        (f (lambda (v)
                            (apply make-cotangent
```

```
                                    (cons chart (cotangent->imbedding S~2 v))))))
        (make-simple-map T*S^2 T*R^3 f)))
(define spherical-hamiltonian
    (smooth-map:compose falling-hamiltonian spherical-inclusion*))
;;; We can even generate the vector field from the Hamiltonian directly:
(define spherical-field
    (hamiltonian->v.field spherical-hamiltonian))
(define spherical-init
    (imbedding->cotangent S^2 (vector 1 0 0) (vector 0 1 .5)))
(define spherical-path
    (v.field->flow T*S^2
                                    (v.field->local-field-maker spherical-field)
                (make-rk4-integrator 1e-3)
                (check-vector-conservation-law
                        (smooth-map:get-point-function spherical-hamiltonian)
                        spherical-init)))
;;; Try to integrate a few time steps:
(define result
    (show-time
        (lambda )
            (spherical-path spherical-init .01))))
process time: 122020 (95550 RUN + 26470 GC); real time: 135198
;Value: result
(/ 135198 1000. 60) ;; 135198 msec. = 2.25 minutes.
;Value: 2.2533000000000003
(length result)
;Value: 10
(for-each
    (compose write-line
                (smooth-map:get-point-function spherical-hamiltonian)
            cadr)
    result)
#(.62500000000009046)
#(.6250000000008399)
#(.62500000000008247)
#(.6250000000008603)
#(.6249999999981802)
#(.6249999999053644)
#(.6249999999049425)
#(.6249999999039165)
#(.6249999999048136)
#(.625)
;No value
```

As seen above, this approach produces reasonable answers: For a short integration,
the Hamiltonian (which equals energy, in this case) is conserved, as expected. However, this approach is very inefficient. Instead, one could derive Hamilton's equations for this Hamiltonian over some atlas of the 2-sphere, and carry these local vector fields to other charts.

```
(define make-spherical-pendulum
    (let* ((C1 (make-cotangent-chart (make-spherical-chart 2 '(2 0 1) 0)))
                (C2 (make-cotangent-chart (make-spherical-chart 2 '(1 0 2) pi)))
                (T*S^2 (charts->manifold (list C1 C2))))
        (lambda (g mass length)
            (let ((k1 (/ (* mass (square length))))
                (k2 (* mass g length)))
                    (lambda (p)
                        (let* ((chart (manifold:find-best-chart T*S`2 p))
                                (x (chart:point->coords p chart))
                                (phi (vector-ref x 0))
                                (theta (vector-ref x 1))
                                    (p_phi (vector-ref x 2))
                                    (p_theta (vector-ref x 3)))
                                    (make-tangent chart
                                    p
                                    (if (eq? chart C1)
                                    (vector (* k1 p_phi)
                            (* (/ k1 (square (sin phi))) p_theta)
                            (+ (* k1 (square p_theta)
                                    (/ (* (square (sin phi))
                                    (tan phi))))
                            (* k2 (sin phi)))
                            0)
(vector (* k1 p_phi)
                            (* (/ k1 (square (sin phi))) p_theta)
                            (+ (* k1 (square p_theta)
                            (/ (* (square (sin phi))
                                    (tan phi)))
                            (* k2 (cos phi) (sin theta)))
                            (* k2 (sin phi) (cos theta)))))))))))
```

This way of defining vector fields requires a bit more work, and tends to produce rather unreadable programs. However, it is sufficiently fast to generate some real data. The local integrator used is a simple 4th-order Runge-Kutta with a fixed step size of $1 \times 10^{-3}$, and the constants are normalized so that $l=g=m=1$. The initial condition, in these units, is $q=(1,0,0), p=(0,1,0.5)$.

Figure $2-8$ shows the relative error in energy conservation, and Figure 2-9 shows the relative error in angular momentum conservation.

Notice that in the code for the integration, check-vector-conservation-law was only asked to minimize the error in energy conservation. Hence, in Figure 2-8, the relative error in energy conservation has been kept rather constant. However, the error in angular momentum makes a few large jumps, probably at the occasions when the integrator decides to switch charts. This indicates that in order to obtain the most accuracy, perhaps one


Figure 2-8: Relative error in energy conservation for the spherical pendulum.


Figure 2-9: Relative error in angular momentum conservation for the spherical pendulum.


Figure 2-10: A contour of the reduced Hamiltonian for the spherical pendulum.
should try to minimize the error in a number of conservation laws. This is what is done with the example of rigid body motion.

Finally, Figure 2-10 shows more evidence that this integrator has found the correct solution: Since angular momentum is conserved for the spherical pendulum, we know that the angular motion (about the vertical axis) of the pendulum may be decoupled from its vertical motion, and the system may be reduced to one with a lower degree of freedom. In this figure, the $z$ vs. $p_{z}$ plot shows that the trajectory of the reduced system is a closed curve. This is because energy is also conserved in the reduced system, and hence trajectories of the reduced system must follow equipotential curves of the reduced Hamiltonian.

### 2.3.3 Rigid body motion and coordinate singularities

Our last example, and the most important, is rigid body motion. Its importance stems from the fact that, although the vector fields describing its motion are perfectly smooth, the coordinate systems traditionally used to describe it contain coordinate singularities, so that usual integrations of rigid body motion can produce inaccuracies near those coordinate singularities.


Figure 2-11: Euler angles for a rigid body.

Furthermore, the configuration space for the rotational motion of rigid bodies is the space of all orientations of a rigid body, or equivalently the space of all rotation matrices in three dimensions. ${ }^{17}$ The manifold structure of this space is rather abstract, and since it is really a 3 -manifold imbedded in the 9 -dimensional space of all $3 \times 3$ matrices, we can no longer rely on our geometric intuition to approach this problem. This is one of the most important examples of an abstract manifold.

Traditionally, orientations of rigid bodies are described by Euler angles, depicted in Figure 2-11. As hinted at earlier, this coordinate system has the problem that the coordinates "blow up" (the Jacobian of the coordinate map becomes singular) when the rigid body is standing vertically, as a bit of analysis will show. This is known as a coordinate singularity because the singularity is part of the coordinate system, not a feature of the dynamics.

The traditional approach to this problem is to work entirely in Euler angles. This works well so long as the trajectory does not come near the coordinate singularity. But when it does, the singularity can have a serious effect on numerical accuracy, which is often reflected in fluctuations in the conserved quantities. In this example, the results of a numerical integration of rigid body motion is presented using the traditional and the

[^10]

Figure 2-12: Relative error in energy conservation for rigid body motion in Euler angles.
manifold method. The principal moments of inertia of the rigid body are $1, \sqrt{2}$, and 2 , with mass set to $m=1$. The initial conditions, in Euler angles, are $\phi=0, \theta=1, \psi=0$, $\dot{\phi}=-0.01, \dot{\theta}=-0.1$, and $\dot{\psi}=-0.01$; these initial conditions have been chosen to take the trajectory close to the coordinate singularity in Euler angles, so that the effects of the singularity on conserved quantities can be observed. The integration was performed using a time step of 0.01 , for 100.0 time units (which equals 10,000 time steps). The integration in Euler angles used a Bulirsch-Stoer integrator, which the manifold integrator also used as its local integrator.

Figure 2-12 shows the relative error in energy conservation for a trajectory that comes relatively near the singularity. Figure $2-13$ shows the analogous plot for the manifold method.

In Figure 2-12, the maximum absolute value is $8.43194301271212 \times 10^{-14}$, and the corresponding average is $2.6428202894715013 \times 10^{-14}$. In contrast, in Figure 2-13, the maximum absolute value of the error is $1.394387463191693 \times 10^{-14}$, and the average absolute value of the error is $4.31070783106112 \times 10^{-15}$. Thus, the manifold approach actually conserves energy better: In terms of relative error, it outperforms the traditional approach


Figure 2-13: Relative error in energy conservation for rigid body motion using the manifold approach.


Figure 2-14: Relative error in conserving the $x$ component of the angular momentum for rigid body motion using Euler angles. The maximum absolute value of the error is $4.163336342344337 \times 10^{-14}$, while the average absolute value of the error is $1.9975479603751012 \times 10^{-14}$.
by about six times.
Note that in Figure 2-12, the curve has a rather sharp peak at time index 4000. That is a consequence of a close encounter between the trajectory and the coordinate singularity. Such a peak can be seen in all of the following plots that were generated using the Euler angles (Figures 2-14, 2-16, and 2-18), and are absent from the plots generated by using the manifold integrator (Figures 2-13, 2-15, 2-17, and 2-19).

Similar comparisons can be made using the components of the angular momentum, as shown in Figures 2-14 through 2-19.

In contrast to the spherical pendulum, in this example all the components of angular momentum (as computed from the inertial frame), as well as the energy function, are used in the integration. Thus, the manifold integrator attempts to minimize deviations from initial values of conserved quantities, which improves their conservation at the cost of making it harder to check how well the system does.


Figure 2-15: Relative error in conserving the $x$ component of the angular momentum for rigid body motion using the manifold approach. The maximum absolute value of the error is $2.7755575615628914 \times 10^{-15}$, while the average absolute value of the error is $3.8748171338198744 \times 10^{-16}$.


Figure 2-16: Relative error in conserving the $y$ component of the angular momentum for rigid body motion using Euler angles. The maximum absolute value of the error is $4.3375450673823944 \times 10^{-13}$, while the average absolute value of the error is $8.348734810181229 \times 10^{-14}$.


Figure 2-17: Relative error in conserving the $y$ component of the angular momentum for rigid body motion using the manifold approach. The maximum absolute value of the error is $1.7798707703661857 \times 10^{-14}$, while the average absolute value of the error is $2.1083459210375576 \times 10^{-15}$.


Figure 2-18: Relative error in conserving the $z$ component of the angular momentum for rigid body motion using Euler angles. The maximum absolute value of the error is $4.352060412667006 \times 10^{-13}$, while the average absolute value of the error is $9.479992724704404 \times 10^{-14}$.


Figure 2-19: Relative error in conserving the $z$ component of the angular momentum for rigid body motion using the manifold approach. The maximum absolute value of the error is $1.1322645212381266 \times 10^{-14}$, while the average absolute value of the error is $1.5603516120091776 \times 10^{-15}$.

### 2.4 Directions for future work

Clearly, in order for this to be useful, several improvements are required. Among these, the most important is probably efficiency: While the manifold integrator is, in many cases, more accurate than traditional methods, the cost of integrating in several charts simultaneously can make such integrators prohibitively slow. One solution is to integrate in one chart at a time, and to have much more sophisticated methods for when and how to switch from one chart to another. While not nearly as elegant as the current approach, this would probably be much more efficient.

Another problem is the difficulty in constructing manifolds. As shown by the example of covering the sphere using stereographic projection, constructing a manifold can take quite a bit of work (especially without the aid of the inverse function theorem). Thus, there need to be better tools, or at any rate larger libraries, for constructing and combining manifolds.

## Chapter 3

## Linear partial differential equations

This chapter describes the application of the manifold abstraction to the numerical solution of linear partial differential equations. For simplicity, the discussion is restricted to scalar equations over two-dimensional manifolds. This is because some of the algorithms described here depend on efficient mesh generators, which are most easily constructed for two dimensions. ${ }^{1}$ However, it should be noted that there exist much more powerful mesh generators than the one used here, and hence the programs developed in this section should generalize to higher dimensions without too much difficulty [6].

Appendix A briefly describes some background material on partial differential equations, including a brief treatment of finite element methods and an even less complete description of iterative solution methods for sparse linear systems of equations. Readers unfamiliar with these topics may wish to take a look at Appendix A first, and to use Vichnevetsky [27] as a more in-depth reference. Petersson [23] describes the solution of PDEs using multiple coordinate systems in a more specialized and less abstract context, as do Chesshire and Henshaw [7].

Note that this chapter focuses on elliptic boundary-value problems, although many of the ideas extend to more general problems. Hyperbolic initial-value problems are considered in the next chapter. The rest of this chapter begins with an exploration of theoretical representations of partial differential operators on manifolds. Then $\S 3.2$ discusses different approaches to the discretization of PDEs on manifolds. These approaches are developed and analyzed in more detail in later sections.

[^11]
### 3.1 Partial differential operators on manifolds

In Chapter 2, first-order ordinary differential equations were redefined as smooth vector fields on differentiable manifolds. By using the tangent bundle construction, higher-order ODEs also became representable in a coordinate-independent fashion. This approach provided a natural framework for representing ODEs using multiple coordinate systems, and for developing these ideas into functional programs that improved the accuracy of numerical integrations. The questions that naturally follow are: How can PDEs be represented in a coordinate-independent fashion? And can similar improvements in accuracy be made?

We begin with a simple observation: Let $M$ be a differentiable manifold, and let $f$ be a smooth real-valued function on $M$. Given a point $p, d f_{p}$ is a linear transformation from $T_{p} M$ into $T_{f(p)} R$, by definition. But the tangent space to $R$ at $f(p)$ is just another copy of $R$, so for any tangent vector $v \in T_{p} M$, the value of the differential of $f$ at $p$ on $v, d f_{p}(v)$, is just another real number. By definition, this corresponds to the directional derivative of $f$ in the direction $v$ in local coordinates, scaled by the length of $v .{ }^{2}$ Since this gives us a way to define the directional derivative of $f$ in the direction $v$ in a coordinate-independent way, we can turn the argument around and say that the vector $v$ operates on the function $f$.

More precisely, let $v$ be a smooth vector field on $M$, and define $v_{p}(f)$ to be $d f_{p}\left(v_{p}\right)$, where $v_{p}$ is the value of the vector field at $p$. Furthermore, define the function $v[f]$ by the equation:

$$
\begin{equation*}
v[f](p)=v_{p}(f)=d f_{p}\left(v_{p}\right) \tag{3.1}
\end{equation*}
$$

Since $v$ and $f$ are smooth, so is $v[f]$. Furthermore, $v$ as an operator on functions is linear, and satisfies the product rule:

$$
\begin{equation*}
v[f \cdot g]=v[f] \cdot g+f \cdot v[g] \tag{3.2}
\end{equation*}
$$

As an operator, then, $v$ has the properties of a differential operator. In fact, one can easily check that, in local coordinates, this turns the vector field $v$ into a linear first-order partial differential operator. Conversely, let $(U, V, \phi)$ be a chart. Then it is not difficult to verify that every first-order differential operator of the form

$$
\begin{equation*}
L f(x)=\sum_{i=1}^{n} a_{i}(x) D_{i} f(x) \tag{3.3}
\end{equation*}
$$

where $f$ is a smooth function on the open subset $V$ of $R^{n}$, uniquely generates a "local vector

[^12]field" on the corresponding subset $U$ of the manifold via the mapping $\phi^{-1}$. Hence, we can define first-order partial differential operators on manifolds to be smooth vector fields. Since vector fields are already coordinate-independent objects, this means first-order operators are also coordinate-independent. Furthermore, higher-order operators may be produced by linear combinations and compositions of first-order operators, so linear partial differential operators can be defined in a nicely coordinate-independent way on manifolds. A linear PDE on a manifold then takes the form
\[

$$
\begin{equation*}
L f=g \tag{3.4}
\end{equation*}
$$

\]

where $f$ and $g$ are smooth functions on the domain $M$, and $L$ is a linear partial differential operator, as described above. Furthermore, if $M$ is a manifold with boundary and $h$ is a smooth function on the boundary $\partial M$ of $M$, then a function $f$ is said to satisfy the boundary value problem with boundary data $h$ if $L f=g$ and $f=h$ on $\partial M$.

Unfortunately, this definition of partial differential operators is too abstract to be useful for practical implementations. In fact, it is very difficult to develop a general representation of differential operators that is efficient for all numerical methods. Thus, each method in this chapter uses a different representation of operators, and programs are structured to provide flexibility with respect to the choice of representation. However, this theoretical definition is still important for the logical framework it provides, and for demonstrating a different way to view vector fields on manifolds. In practice, though, it is Equation (3.3) and its higher-order generalizations that play a more important role in computation.

### 3.2 Approaches to discretization

General comments. Differential equations determine unknown functions. Thus, to facilitate numerical computation, it is often necessary to parametrize the set of possible solutions using finitely many variables, and to reduce the PDE itself to a system of algebraic equations that determine the values of these variables. This process of reducing a PDE into a system of algebraic equations is called discretization.

In general, one can describe discretization in terms of two separate but interdependent steps: First, one must choose a representation for the approximate solution, so that a finite set of variables can be mapped to a function approximating the true solution. This often involves series expansions, such as Fourier series, power series, or expansion in terms of finite element basis functions. For these cases, the finite set of variables to which the unknown function has been reduced are, respectively, the Fourier coefficients, the Taylor coefficients, or values of the given function at specified sample points. The choice of a rep-
resentation, informally, corresponds to the geometric part of discretization: In choosing a representation for approximate solutions, one often needs to first discretize (i.e. represent using a finite number of parameters) the domain of the PDE. Of course, as finite element methods show, there is more to choosing representations than simply discretizing (or triangulating) the domain-One must also choose the order of the basis functions and various other parameters.

In contrast, the derivation of discrete algebraic equations can be said to discretize the $P D E$ itself. This step often involves either replacing the differential operator with finite difference operators, as in standard finite difference schemes, or by invoking some other formulation of physical problems, such as variational principles or Galerkin's orthogonality condition. ${ }^{3}$ To some extent, this component of the discretization process can be performed independently of the domain discretization in that one can often use the same discretized domain to discretize different PDEs that are defined over the same domain. However, the method of discretizing the PDE, be it finite elements or finite differences, must work very closely with the discretized domain. Thus, the two components are not truly independent, although it is important to recognize the flexibility and modularity in the structure of PDE solvers.

In this report, the focus will be on finite difference and finite element methods, so the domain discretization will involve choosing a discrete set of sample points and, for finite elements, generating the appropriate mesh.

Global methods versus local ones. The discussion above on discretization applies unambiguously to the discretization of PDEs whose domains are regions in Euclidean spaces. However, in the case of manifolds, we have a choice in the order in which the various steps are carried out because of the existence of multiple coordinate systems: One choice is to discretize the entire manifold first, and then discretize the PDE. For example, using finite elements, we would first triangulate the entire manifold before invoking variational principles to derive the discretized equations. In this report, this type of discretization is called global discretization.

On the other hand, we can first discretize the PDE locally, so that for each chart there exists a set of discretized equations. These sets of discretized equations must then somehow be combined to form a global system of equations that determine the approximate solution everywhere. This is called local discretization.

Since finite difference methods are inherently local, the distinction between global and local discretization is very little when one uses finite difference techniques. However, finite

[^13]element methods require triangulations, and the general problem of triangulating manifolds is a rather difficult one in computational geometry. There appears to be no welldocumented way of performing such triangulations except for low dimensions. ${ }^{4}$ With improvements in computational-geometric algorithms, the global discretization approach may become tractable someday, but it is too difficult to use in general with currently available tools. In contrast, local discretization methods do not suffer from such handicaps because we can always choose charts with simple images in $R^{n}$, which simplifies the local triangulation process. ${ }^{5}$

Consequently, this chapter concentrates on local methods: Each chart is independently discretized in the local discretization phase, and the resulting local equations are then combined to form a global set of equations in the combination phase. ${ }^{6} \S 3.3$ considers local discretization using finite difference techniques, where the primary problem is the formulation of local equations and their solution. §3.4 then discusses the use of finite element methods, which require special attention to the combination phase; some simple ideas are proposed and tested first, followed by a somewhat more efficient and accurate algorithm. Finally, $\S 3.5$ revisits the topic of mesh generation on manifolds and discusses some of the difficulties involved.

There is much more work to be done in the application of the manifold abstraction to the numerical solution of PDEs, and $\S 3.6$ suggests some of these possible directions.

### 3.3 Finite differences on manifolds

Recall that finite difference techniques generally involve the use of difference quotients to replace derivatives, thus transforming partial differential equations into linear algebraic equations which can then be solved using a variety of numerical techniques. Approximate solutions are represented by their values at some set of chosen sample points, often referred to as nodes in this document, ${ }^{7}$ and algebraic equations are derived to relate the values at these discrete sample points to each other.

[^14]

Figure 3-1: Copying nodes in the overlap between two charts to enforce constraints on unknown values, thus combining local equations into a global system. In this figure, the triangular nodes belong to chart 1 , while the circular nodes belong to chart 2.

There are several possibilities for applying finite difference techniques to manifolds. What follows is the pseudocode for one of the simplest methods:

```
;;; This is the pseudocode for a finite differences algorithm on manifolds.
;;; Actually, this can easily be turned into a working program, but since most
;;; of the following material has already been implemented in C for speed, the
;;; Scheme versions were never implemented.
(define (finite-difference-on-manifold M L g h)
    ;; M should be a manifold, L a linear differential operator, and G and H
    ;; should be smooth functions on M. The solution U is a function such that
    ;; (L U) = F over M, and where U = G on the boundary of M.
    (let ((charts (manifold:get-finite-atlas M)))
    ;; Based on the local geometry of each chart, construct a collection of
    ;; sample points. Then for each node, compute its finite difference
    ;; coefficients with respect to its neighbors in each chart:
    (for-each
    (lambda (nodes)
        (for-each
        (lambda (node)
            (node:set-fd-coefficients! (compute-fd-coefficients node nodes)))
        nodes))
        (process-node-lists (map make-nodes charts) charts))))
```

This program contains a number of auxiliary procedures: Make-nodes takes a chart and constructs a list of nodes for that chart. Manifold:get-finite-atlas returns a finite atlas (i.e. a finite list of charts) for the manifold, if such a thing exists. Process-node-lists is a procedure that copies nodes between charts in overlapped regions (see Figure 3-1), so that nodes that lie in the overlap of two charts will exist in both charts and agree on the value of
the approximate solution at that point. Finally, the procedure compute-fd-coefficients locally discretizes the PDE, and can use any method it prefers to derive the finite difference coefficients of node with respect to its neighbors in chart.

Note that by copying nodes between lists in process-node-lists, we have implicitly constrained the system of equations to be consistent with each other on overlapped regions between charts. Thus, two sample points $x_{1} \in C_{1}$ and $x_{2} \in C_{2}$ are guaranteed to have the same value if $x_{1}$ and $x_{2}$ really correspond to the same point $p$ in $M$. Process-node-lists thus performs all the necessary work for the combination phase. On the other hand, compute-fd-coefficients is the part of the program that controls how information flows between different parts of the discretized domain. For example, since many physical systems arise from local interactions, this procedure can be written to consider only those nodes in the list nodes that are physically close to the given node, node.

The combination phase of this local method, as described above, may seem trivial. However, because nodes are copied between charts, it is in general impossible to guarantee that nodes lie on regular grids. This causes two problems: First, local discretization becomes more difficult, since many standard methods depend on regular grids (we will see such a method later). Second, it often turns out that in the irregular case, the resulting finite difference equations are not sufficiently structured to be solvable by iterative methods such as relaxation. ${ }^{8}$ But the application of direct or semi-direct methods to large matrices can be computationally intensive and numericaly undesirable, and hence the resulting set of linear algebraic equations can become very difficult to solve. The price we paid for simplicity in the combination phase is that the local problem becomes more difficult.

Chesshire and Henshaw avoid these difficulties by using a different approach [7]: Their method uses locally regular grids for local discretization, and instead of copying nodes (which destroys the regularity of local grids in the method outlined above), the combination phase is carried out by using interpolation functions between nodes. While this works well for some problems, however, it relies on much more complicated procedures for the combination phase and restricts the types of charts one could use. Thus, their method is not explored in this section, although variations on their idea are explored later in the context of finite elements.

The rest of this section focuses on the local problem of obtaining and solving finite difference approximations for PDEs because the problem is already non-trivial at that level, and adding the complication of solving PDEs on manifolds probably would not help. ${ }^{9}$ Both

[^15]

Figure 3-2: The discrete Dirichlet problem.
the simple method described above and that of Chesshire and Henshaw involve more difficulties, and thus in this section we only consider the application of finite differences to irregularly-distributed sample points over subsets of Euclidean space. This is an interesting problem in its own right.

### 3.3.1 Generating coefficients for irregular sample points

This section discusses the problem of local discretization using finite differences. As such, all domains are open subsets of Euclidean spaces unless otherwise stated.

## The discrete Dirichlet's problem

As mentioned in $\S 3.3$, one of the primary problems encountered in implementing the algorithm above is the formulation of finite difference techniques using irregularly-distributed sample points. Before tackling this more difficult case, though, let us revisit the canonical example of finite differences: Laplace's equation on a regular rectangular grid and the discrete version of Dirichlet's problem (see Figure 3-2).

The basic idea is this: Let $f$ be a real-valued differentiable function of one real variable. By the definition of the derivative, we have:

$$
\begin{equation*}
f^{\prime}(x) \approx \frac{f(x+h / 2)-f(x-h / 2)}{h} \tag{3.5}
\end{equation*}
$$

This is the central-difference approximation, and has nicer numerical properties than the standard forward-difference approximation.
ometry to improve their numerical properties. But this turns out to be a rather difficult problem. For more details, Clark, et. al., present and analyze one possible way of carrying out this procedure, and describe its application to image processing [8].

Applying this approximation twice to $f$ at $x$, we have an estimate of the second derivative of $f$ :

$$
\begin{equation*}
f^{\prime \prime}(x) \approx \frac{f(x+h)+f(x-h)-2 f(x)}{h^{2}} \tag{3.6}
\end{equation*}
$$

Now suppose we are interested in solving the boundary value problem for Laplace's equation over some region $\Omega$ in $R^{n}$. Cover the space $R^{n}$ by a lattice $L_{h}=\left\{\left(x_{1}, \ldots, x_{n}\right)\right.$ : $\left.x_{i}=k_{i} h, k_{i} \in Z\right\}$ with spacing $h>0$, and choose $h$ sufficiently small so that the domain $\Omega$ may be approximated by a subset $\Omega_{h}$ of $L_{h}$. Applying the formula above, we obtain:

$$
\begin{equation*}
\nabla^{2} u(x, y) \approx \frac{u(x+h, y)+u(x-h, y)+u(x, y+h)+u(x, y-h)-4 u(x, y)}{h^{2}} \tag{3.7}
\end{equation*}
$$

Upon rearrangement and setting the Laplacian of $u$ to 0 , this yields the familiar formula:

$$
\begin{equation*}
u(x, y) \approx \frac{u(x+h, y)+u(x-h, y)+u(x, y+h)+u(x, y-h)}{4} . \tag{3.8}
\end{equation*}
$$

This formula is sufficient to determine approximate solutions of Laplace's equation over a regular lattice with reasonable accuracy for domains with sufficiently smooth boundaries and boundary data.

## Polynomial interpolation

However, we cannot generalize this method to other irregular sample points because we made heavy use of the regularity of the grid in its derivation: The approximation formula (3.6) was valid because the sample points are regularly spaced, and an approximation of the Laplacian operator could be made because the lattice is generated by the orthogonal vectors $h \hat{x}$ and $h \hat{y}$, which lets us take the appropriate derivatives for computing the Laplacian. Thus, this method would not work if the sample points did not lie on a regular grid.

A different approach to finite differences is thus necessary. One natural idea is polynomial interpolation ${ }^{10}$ : In any finite difference method, the primary goal is to express the partial differential equation as a set of coupled finite difference equations. Since we are only concerned with linear operators here, it is natural to take these finite difference equations to be linear. In particular, let the $i$ th sample point be $p_{i}$, and let $L$ be a linear differential operator. Then for each sample point, we would like to find coefficients $a_{i j}$ such that:

$$
\begin{equation*}
L u\left(p_{i}\right)=\sum_{j} a_{i j} u\left(p_{j}\right) \tag{3.9}
\end{equation*}
$$

[^16]where the index $j$ ranges over all other nodes. Furthermore, since many physical problems involve only local interactions, and because of concerns for computational efficiency on parallel machines, the indices $a_{i j}$ are chosen so that $a_{i j}$ is non-zero only if $p_{i}$ and $p_{j}$ are physically close. Deciding whether two sample points are close or not is, of course, a parameter that needs to be chosen. Usually, one can call two sample points close if $\left|p_{i}-p_{j}\right|<$ $R$ for some fixed radius $R$; in that case, $p_{i}$ and $p_{j}$ are called neighbors.

One way of computing the coefficients $a_{i j}$ for some fixed $i$ is as follows: Suppose that we would like to choose the coefficients for some point $p_{i}$ with respect to its neighbors $p_{n_{i k}}, k=1, \ldots, n_{i}$. For concreteness, let the domain be a subset of the plane. Then we can require that the approximation (3.9) is exact on some set of test functions, $\phi_{1}, \phi_{2}, \ldots, \phi_{m_{i}}$. Substituting the basis functions into Equation (3.9), this gives:

$$
\begin{equation*}
L \phi_{j}\left(p_{i}\right)=\sum_{k=1}^{n_{i}} a_{i n_{i k}} \phi_{j}\left(p_{n_{i k}}\right), j=1,2, \ldots, m_{i} \tag{3.10}
\end{equation*}
$$

Clearly, this is a set of $m_{i}$ linear equations in the $n_{i}$ variables $a_{i n_{i k}}$ (recall that $i$ is fixed). If we have enough basis functions $\phi_{j}$ so that $m_{i}=n_{i}$, and if the basis functions are chosen so that the $L \phi_{j}$ can be easily computed, then the equations (3.10) provide an efficient means of determining the unknown coefficents $a_{i n_{i k}}$. Indeed, when this is applied to the rectangular grid, where each grid point is given its immediate neighbors in the $\hat{x}$ and $\hat{y}$ directions as neighbors, this process yields the approximation (3.8).

### 3.3.2 Solving linear algebraic equations

While this method gives reasonable approximations of the differential operator $L$, there is a serious problem: The iterative methods usually used to solve the resulting linear algebraic equations, such as successive overrelaxation, do not converge, while the use of direct or semi-direct methods are often not possible for very large systems of equations.

One idea is to take advantage of the following well-known theorem: If $A$ is a symmetric positive-definite matrix, then successive overrelaxation converges for all overrelaxation factors $0<\bar{\omega}<2$. Now, suppose we wish to solve the linear system of equations $A x=b$ for some non-singular matrix $A$. Then $A^{T} A x=A^{T} b$ is equivalent to the original system of equations. Furthermore, if $A$ is nonsingular, then $A^{T} A$ is positive-definite and the theorem applies. Additionally, this computation can be carried out locally: Since the $j$ th column of $A$ consists of the coefficients $a_{n_{j k} j}, k=1,2, \ldots, n$, which are the coefficients of $p_{n_{j k}}$ with respect to $p_{j}$, two columns share a non-zero entry if and only if the corresponding sample points are within two radii of each other (see Figure 3-3). Since the entries of $A^{T} A$ are actually the dot products of the columns of $A$, the computation of $A^{T} A$ remains mostly local, with the neighborhood radius of each node increasing from $R$ to $2 R$.


Figure 3-3: The case of irregularly-distributed nodes: In performing the "transpose trick," two nodes have non-zero coefficients for each other if and only if they are within two hops.

Unfortunately, this clever idea is not as magical as it may seem at first: First, by multiplying a matrix with its own transpose, the condition number of the matrix is squared. ${ }^{11}$ This tremendously worsens the numerical properties of the matrix. Furthermore, the theorem quoted earlier states that the relevant spectral radius, $\rho$, is less than 1. However, it does not bound $\rho$ away from 1. Thus, the actual spectral radius is often so near 1 that, in the presence of round-off error, the method converges too slowly to be useful, and we are forced to explore other methods.

### 3.3.3 Numerical examples

This sections presents the results of some numerical experiments using finite differences. Out of a desire to compute using a large number of nodes rather quickly, the programs have been written in C. Thus, the source code will not be included here because they are not very illuminating.

The problem in which we are interested is the rectangular slot problem: Consider the unit square $\Omega=[0,1] \times[0,1]$, depicted in Figure 3-4. Given the electric potential on the boundary of $\Omega$ and the condition that there are no charges in the interior of $\Omega$, what is the electric potential everywhere inside $\Omega$ ? From electrostatics, we know that the solution must satisfy Laplace's equation. Furthermore, analytical solutions of this problem can be easily derived using Fourier methods, so that numerical answers can be checked against the true

[^17]

Figure 3-4: Determining the electric potential in a rectangular slot, with boundary conditions specified by Equation (3.11). The plot is generated by dividing the unit square into smaller squares, over which the nodal values are averaged. This reduces the number of points that need to be plotted.
solution. ${ }^{12}$
For our purposes, it is useful to just settle on boundary conditions whose corresponding solution is easy to compute. One such example is:

$$
h(x, y)=\left\{\begin{array}{cc}
1+\sin (\pi x), & y=1  \tag{3.11}\\
1, & \text { otherwise } .
\end{array}\right.
$$

The exact solution for these boundary values is:

$$
\begin{equation*}
u(x, y)=1+\frac{\sinh (\pi y) \cdot \sin (\pi x)}{\sinh (\pi)} \tag{3.12}
\end{equation*}
$$

Notice that practically every function involved has the constant 1 added to it. This bounds solution values away from zero so that meaningful relative errors may be computed; it should not add significantly to the numerical error, since 1 is of the same order of magnitude as the solution values. ${ }^{13}$

[^18]\[

h(x, y)=\left\{$$
\begin{array}{cc}
1, & y=1  \tag{3.13}\\
0, & \text { otherwise } .
\end{array}
$$\right.
\]



Figure 3-5: The electric potential in a rectangular slot, determined by finite difference computations on a regular rectangular grid. About 100,000 iterations (with $\bar{\omega}=1.9$ ) were run, so the solver may not have converged to the "true" approximate solution yet.

A note on graphics. It is vital to note that in this section, all plots of sample values over the unit square are produced by dividing the unit square into rather coarse grids first, and then averaging over the sample values. This simplifies the task of plotting, but at the risk of making the data appear more smooth than it is. So please take care not to be misled by the apparent simplicity of the plots.

Regular grid. First, let us use the approximation (3.8) to approximate the solution on a regular rectangular grid. The actual grid used consists of 10,000 nodes, placed at regular intervals in the unit square $\Omega$ on a $100 \times 100$ grid. After applying Equation (3.8) to each node for about $10^{6}$ iterations, ${ }^{14}$ the resulting values are checked against the actual solution.

Figure 3-4 shows the shape of the electric potential arising from the boundary conditions specified in Equation (3.11). Figure 3-5 shows the values obtained from the regular grid approximation. Note that they are qualitatively alike.

In fact, one can plot the error between the two; this is shown in Figure 3-6. Notice that the error reaches its maximum near the non-zero boundary values.

[^19]

Figure 3-6: The absolute difference between the functions depicted in Figure 3-4 and Figure $3-5$. The maximum absolute error is 0.0000291001797184 , the minimum absolute error is 0.0000000157547926 , and the average absolute error is 0.0000115848344767 . The maximum relative error, on the other hand, is 0.0000216438565018 , the minimum relative error is 0.0000000157534190 , and the average relative error is 0.0000093994479478 .

Randomly-distributed sample points and simple averaging. The next idea depends on an alternative derivation of the approximation (3.8): Let $u$ be a function over some region $\Omega$. For every point $p$ and any real $r>0$, denote the closed ball of radius $r$ centered at $p,\{q:|p-q| \leq r\}$, by $B_{r}^{n}(p)$, and denote its boundary (the $n-1$-sphere) by $S_{r}^{n-1}(p)$. Then, $u$ is said to have the mean-value property if for every point $p$ and radius $r$ such that $B_{r}^{n}(p)$ is contained in $\Omega, u(p)=\int_{S_{r}^{n-1}(p)} u d S$ (where $d S$ denotes the appropriate measure for a surface integral). A well-known theorem then states that $u$ satisfies Laplace's equation if and only if it has the mean value property.

The equivalence of Laplace's equation and the mean-value property has many important consequences; Ahlfors [3] contains more details. For our part, it can be used to derive another approach to Laplace's equation: One uses randomly-distributed nodes, ${ }^{15}$ but instead of trying to perform fancy derivations of finite difference coefficients, each node simply av-

[^20]

Figure 3-7: The potential computed by simple averaging using randomly-distributed nodes. As in previous figures, this plot is generated by dividing the unit square into smaller squares, over which the nodal values are averaged. So please keep in mind the comments at the beginning of this section: This plot may appear to be more smooth than the actual data because of the averaging procedure.
erages the values of its neighbors within a given radius $R$ and sets its own value to this average. The validity of this approach follows from the mean-value property and a simple volume integral over the closed ball of radius $R$ centered at each point $p$.

Figure 3-7 shows the approximate solution constructed this way; the smooth surface is generated by locally averaging nodal values. In this particular computation, there are 10,000 nodes in the rectangular slot, each having an average of 25 neighbors. Notice that it is qualitatively similar to Figures $3-4$ and $3-5$. However, as Figure $3-8$ shows, the error distribution is much less smooth and is much larger.

Furthermore, we can examine the relationship between the average error and parameters such as the radius $R$ and the number of nodes. Figure 3-9 plots average absolute error against the radius $R$ for a domain having a fixed number of nodes. We see that as $R$ decreases, the error decreases as well. This can be understood in terms of a node's ability to adapt to the approximate solution: Averaging over too large a neighborhood "stiffens" the approximate solution and makes convergence to solutions with large gradients difficult. Also, the semi-log plot shows that the error decreases approximately exponentially for sufficiently large $R$, though the curve tapers off as $R$ becomes smaller. However, the error cannot be made arbitrarily small by decreasing $R$ along, because nodes can become disconnected


Figure 3-8: The error distribution for the averaging method. The maximum absolute error is 0.0303547633666503 , the minimum absolute error is 0.0000001458319725 , and the average absolute error is 0.0044948995201661 . The maximum relative error, on the other hand, is 0.0246888943589139 , the minimum relative error is 0.0000001448811931 , and the average relative error is 0.0035004079565252 .


Figure 3-9: The average absolute error versus the radius $R$. The domain contains 1,000 nodes while the radius ranged from 0.2 to 0.05 . Successive overrelaxation is performed on each configuration for 100,000 iterations, with $\bar{\omega}=1.7$.


Figure 3-10: The average relative error versus total number of nodes. The number of nodes varies from 1,000 to 5,000 , and the radii are changed to keep the average number of nodes per neighborhood at around 27 nodes. Successive overrelaxation is performed on each configuration for 100,000 iterations, with $\bar{\omega}=1.7$.
from each other for sufficiently small $R$, and the boundary data would then have no way of "propagating" to interior nodes.

Figure 3-10 shows the analogous plot for the average absolute error versus the total number of nodes, with the density held constant by changing the radius. This log-log plot demonstrates an approximate power law governing the relation between the total number of nodes (given fixed density) and the average absolute error.

However, despite its simplicity and reasonable accuracy, the averaging method is limited by its lack of generality: Because it uses properties specific to Laplace's equation, it is not immediately applicable to other elliptic differential equations. This is one of the advantages of generating finite difference coefficients using polynomial interpolation, as described in §3.3.1.

Randomly-distributed sample points and polynomial interpolation. Let us now take a look at the finite difference coefficients generated using polynomial interpolation. Unlike the case of regular grids, the iteration diverges rather quickly. For the ease of computation, this section examines systems with smaller numbers of nodes - The tests


Figure 3-11: The approximate solution generated by applying direct matrix inversion to the system of equations generated by polynomial interpolation. The maximum absolute error is 0.0081833977839731 , the minimum absolute error is 0.0000000779550711 , and the average absolute error is 0.0004690292039753 . The maximum relative error is 0.0076113517976692 , the minimum relative error is 0.0000000745384821 , and the average relative error is 0.0004207582072576 .
here use 300 interior nodes distributed uniformly in the unit square and 144 nodes spaced evenly along the boundary, with the same boundary conditions (3.11).

For a system this size, one could explicitly compute the spectral radius for various iteration methods. ${ }^{16}$ Indeed, for the example here, the spectral radius for Gauss-Seidel is 73.75932386604968 , while that of Jacobi iteration is 6.69818594658326 . Thus, both iteration methods diverge for this system. However, as a test of the accuracy of the coefficients themselves, we can directly invert the matrix using LU decomposition. ${ }^{17}$ The result, shown in Figure 3-11, demonstrates that polynomial interpolation actually produces fairly accurate answers-If one had the ability to solve the resulting equations.

The transpose trick. So what happens if we actually attempt to apply the "transpose trick" described in §3.3.2? Does this really improve the stability of Gauss-Seidel iterations? The answer is a lackluster affirmative: The spectral radius for Gauss-Seidel iteration is 0.99999999123756 , while that of Jacobi iteration is 7.52337630885650 . Thus, Gauss-Seidel

[^21]

Figure 3-12: The approximate solution generated by applying direct matrix inversion to the system of equations generated by the "transpose trick." The maximum absolute error is 0.0079623718237853 , the minimum absolute error is 0.0000001702022259 , and the average absolute error is 0.0004585412757934 . The maximum relative error is 0.0074057762673312 , the minimum relative error is 0.0000001577217479 , and the average relative error is 0.0004110492850620 .
(in theory) converges for this problem, even though the spectral radius is close enough to 1 that convergence is very slow. Furthermore, the condition number of the matrix before multiplying by the transpose is $2.016135227435024 \times 10^{6}$, while after multiplying by the transpose it becomes $2.382088963154271 \times 10^{12}$ —Roughly squared, as expected.

Thus, instead of applying iteration to these equations, LU decomposition is applied directly as in Figure 3-11. The result is shown in Figure 3-12.

### 3.4 Finite elements on manifolds

An alternative to finite difference techniques is to employ finite element methods in local discretization, which in general do not require regular grids to perform efficiently (as do finite difference methods). However, the combination of local equations into a global system can be more problematic for finite elements than for finite differences.

The basic idea of finite elements on manifolds is simple: For each chart ( $U, V, \phi$ ), one can map the open set $U$ onto the open subset $V$ of $R^{n}$. Since $V$ is an open subset of $R^{n}$, one can generate a mesh that covers almost all of $V$ in a number of ways: One way is to always ensure that $V$ is of a simple shape by choosing the appropriate mapping $\phi$; then it is
a easy to generate a regular grid over $V$. Another way is to generate a set of nodes that fill $V$ "densely," and to triangulate them using a mesh generation algorithm such as quickhull [6], which works for general $n$-dimensional convex polytopes. Having generated a mesh over each chart, one can then apply standard finite element methods, such as Rayleigh-Ritz or Galerkin's method, to the open subset $V$ of $R^{n}$. This yields locally discretized equations for each chart.

The next step is to combine the equations. One straightforward proposal is to choose a set of nodes in the overlap region between charts, and to constrain the unknown value at each of those nodes to the interpolated value from the other chart, thus generating a relation between unknown variables in different charts. The nodes chosen to form these constraints are called interpolation nodes, and choosing good ones turns out to be rather tricky: Too few, and not enough information propagates between charts to generate a good solution. Too many, and the resulting equations become overconstrained and cannot come anywhere close to the real solution.

Before discussing these issues in detail, however, it is useful to devlelop a deeper understanding of what it means to integrate functions over manifolds.

### 3.4.1 Integration on manifolds

Integration is a very powerful tool in the study of partial differential equations, particularly in the formulation of numerical methods. This is because integrals are much easier to compute accurately and have a number of other nice properties, and can often be used to reformulate PDEs in ways that simplify numerical solution methods. For example, finite element methods often rely on variational principles (as in the Rayleigh-Ritz method) or orthogonality conditions (as in Galerkin's method) to discretize PDEs: In the former case, the computation of the action functional to be minimized requires integration over the domain of the PDE, and in the latter case, the evaluation of the inner product on the function space of possible solutions again requires the integration of functions over the domain.

While these ideas are all straightforward to define on subsets of Euclidean space, it is less obvious how one can arrive at a coordinate-independent definition of integration on manifolds. Integration, as opposed to differentiation, is inherently a global operation, not a local one, and thus the definition of integrals is more difficult than that of differentials.

There is no unique way to define the integral of a real-valued function on manifolds. However, one could integrate real-valued functions over Riemannian manifolds (see §2.1.4), where a "smoothly-varying" inner product is defined on each of tangent spaces. There is another useful approach to integration that relies on "differential forms." Since this material
will not be needed for our purposes here, a discussion is postponed until §B.2.

## Partitions of unity

In view of the usefulness of tangent vectors on manifolds, which were defined using the fact that manifolds locally "look like" Euclidean spaces, one natural idea would be to reduce the problem of integrating a function over the whole manifold to the problem of integrating a function over a chart. That is, the problem of integration can be divided into two subproblems: The first is how to reduce the problem of integration to a local problem, and the second is how to define integration locally in a consistent way so that the integral of a function over a small subset of the manifold is independent of the chart chosen to evaluate that integral.

It turns out that the two approaches to integration mentioned above differ only in how they solve the second subproblem. The common solution to the first subproblem, called a "partition of unity," is a simple but powerful idea.

Let $\left\{\rho_{i}\right\}$ be a set of smooth real-valued functions on a manifold $M$, let $U_{i}$ denote the interior of the support of $\rho_{i}$, and let $\mathcal{A}$ be an atlas that is compatible with the atlas of $M$. Then $\left\{\rho_{i}\right\}$ is a partition of unity subordinate to $\mathcal{A}$ if:

1. $\rho_{i}(x) \geq 0$ for all $x \in M$.
2. For each $i$, there exists a chart $(U, V, \phi) \in \mathcal{A}$ such that the support $\bar{U}_{i}$ of $\rho_{i}$ is contained in $U$. Furthermore, $\bar{U}_{i}$ is compact. ${ }^{18}$
3. $\sum_{i} \rho_{i}(x)=1$ for all $x \in M$.
4. Every point $x \in M$ has a neighborhood $W$ such that $W$ is contained in only finitely many of the sets $U_{i}$.

For any atlas on any manifold, there exists a partition of unity subordinate to it. For a proof of this fact, see Munkres [21], Guillemin and Pollack [14], or Warner [28]. In this discussion, the atlas to which a partition of unity is subordinate may not be mentioned explicitly; in such cases, the atlas of the manifold is assumed. ${ }^{19}$ Incidentally, finite element basis functions furnish a nice example of a partition of unity.

[^22]Suppose, now, that we have already found a nice way to define an integral operator " $\int$ " on real-valued functions over the manifold $M$. What properties should it have? First of all, integrals should be linear, that is, the integral of two functions $f$ and $g$ should satisfy

$$
\begin{equation*}
\int_{M}(a f+b g)=a \int_{M} f+b \int_{M} g \tag{3.14}
\end{equation*}
$$

for real constants $a$ and $b$. Now note that for any function $f$ and any partition of unity $\left\{\rho_{i}\right\}$, the following equation holds for all $x \in M$ :

$$
\begin{equation*}
f(x)=\sum_{i} \rho_{i}(x) f(x) \tag{3.15}
\end{equation*}
$$

This expression is well-defined, because even though the collection $\left\{\rho_{i}\right\}$ may be infinite, axiom 4 shows that for each $x$, only finitely many of the numbers $\rho_{i}(x)$ is non-zero. Thus, this potentially infinite series is actually a finite sum for each $x$, and the expression is well-defined. The equation then follows from the fact that the $\rho_{i}$ sum to 1.

Combining this with the linearity of integrals, we obtain:

$$
\begin{align*}
\int_{M} f & =\int_{M} \sum_{i} \rho_{i} f  \tag{3.16}\\
& =\sum_{i} \int_{M} \rho_{i} f \tag{3.17}
\end{align*}
$$

But each of the functions $\rho_{i} f$ has compact support. Furthermore, the support of $\rho_{i} f$ must be a subset of the support of $\rho_{i}$, which is contained entirely in some chart.

Conversely, suppose that we have a way of integrating functions whose supports lie entirely within a chart. It is easy to show that the choice of a partition of unity to combine these integrals does not affect the final outcome: Let $\left\{\rho_{j}^{\prime}\right\}$ be another partition of unity subordinate to the atlas $\mathcal{A}^{\prime}=\left\{U_{j}^{\prime}\right\}$. Then:

$$
\begin{equation*}
\sum_{i} \int_{U_{i}} \rho_{i} f=\sum_{i, j} \int_{U_{i} \cap U_{j}^{\prime}} \rho_{i} \rho_{j}^{\prime} f=\sum_{j} \int_{U_{j}^{\prime}} \rho_{j}^{\prime} f . \tag{3.18}
\end{equation*}
$$

We have thus reduced the problem of finding a reasonable definition of integrals of functions on manifolds to a local problem: How can we integrate functions whose supports lie entirely in a given chart?

## Integration on Riemannian manifolds

Consider now open subsets $V_{1}$ and $V_{2}$ of Euclidean $n$-space. Suppose $f$ is a smooth bounded real-valued function on $V_{2}$ (boundedness is generally required to ensure that the integral is
finite), and that there exists a smooth bijective map $\phi$ from $V_{1}$ to $V_{2}$. Using the change of variables theorem, we know that the integral of $f$ over $V_{2}$ can be written in two ways:

$$
\begin{equation*}
\int_{y \in V_{2}} f(y) d y=\int_{x \in V_{1}} f(\phi(x))|\operatorname{det} D \phi(x)| d x \tag{3.19}
\end{equation*}
$$

where traditional notation, rather than functional notation, was used for the sake of clarity.
As stated in §3.4.1, one major aspect of defining integration on manifolds is finding a consistent definition of local integrals. In view of Equation (3.19), this amounts to figuring out what geometric information is necessary to construct objects that transform like determinants, so that integrals of functions over "small" subsets of the manifold are the same no matter what chart is used. The approach here is to relate determinants to a local measure of volume in tangent spaces of the manifold, so that the function analogous to the determinant can be defined geometrically. ${ }^{20}$

The geometry of determinants. Let us begin with the geometric interpretation of the determinant: Let $S$ be a set of $n$ vectors $B=\left\{v_{1}, v_{2}, \ldots, v_{n}\right\}$ in Euclidean $n$-space. Every such set $S$ defines a parallelpiped:

$$
\begin{equation*}
\left\{v \in R^{n}: v=\sum_{i=}^{n} a_{i} v_{i}, \sum_{i=0}^{n} a_{i} \leq 1, a_{i} \geq 0\right\} \tag{3.20}
\end{equation*}
$$

where $v_{0}=\sum_{i=1}^{n} v_{i}$. This generates a convex polyhedron with vertices at the origin, each of the points $v_{i}$, and the point $v_{0}=\sum v_{i}$; in the case $n=2$, this is just the definition of a parallelogram. The $n$-dimensional volume of this geometric object is then $|\operatorname{det} A|$, where $A$ is the matrix whose columns are the vectors $v_{1}, v_{2}, \ldots, v_{n}$.

Now, this definition of volume implicitly used the structure of Euclidean space. The determinant depends on the components of the matrix $A$, which in turn depend on the particular basis chosen. In the Euclidean case, there is a standard basis, but general vector spaces do not have special bases singled out for them, and hence the determinants of linear transformations are not well-defined. However, for inner product spaces, the determinant $i s$ well-defined, up to a sign:

Let $V$ and $W$ be $n$-dimensional inner product spaces, and let $L$ be a linear transformation from $V$ to $W$. Choosing bases $B_{V}$ and $B_{W}$ for $V$ and $W$, respectively, we can write $L$ as a matrix with real components. Its determinant is then well-defined with respect to these bases. In particular, let $B_{V, 1}$ and $B_{V, 2}$ be orthonormal bases for $V$, and let $B_{W, 1}$ and $B_{W, 2}$

[^23]be orthonormal bases for $W$. If we let $L_{i}$ be the matrix representation of $L$ with respect to the bases $B_{V, i}$ and $B_{W, i}$, then elementary linear algebra shows that:
\[

$$
\begin{equation*}
L_{2}=A_{W}^{-1} \cdot L_{1} \cdot A_{V} \tag{3.21}
\end{equation*}
$$

\]

where $A_{V}$ is the matrix representation of the basis $B_{V, 2}$ with respect to the basis $B_{V, 1}$, and $A_{W}$ is the matrix representation of the basis $B_{W, 2}$ with respect to the basis $B_{W, 1}$. But the bases $B_{V, i}$ and $B_{W, i}$ are chosen to be orthonormal for $i=1,2$, so the matrices $A_{V}$ and $A_{W}$ are orthogonal, and their determinants are $\pm 1$. Thus, $\operatorname{det} L_{2}= \pm 1 \operatorname{det} L_{1}$, and we see that for inner product spaces, one can define the determinant in a consistent way up to a factor of $\pm 1$.

We can therefore make the following definition: Let $L$ be a linear transformation from an inner product space $V$ to another inner product space $W$, both of dimension $n$. Then the function $|\operatorname{det} L|$ is defined to be the absolute value of the determinant of $L$ with respect to any orthonormal bases for $V$ and $W$. By the argument above, this is well-defined. Furthermore, like ordinary determinants, this has the following properties: $|\operatorname{det} I|=1$ for the identity operator $I$, and given inner product spaces $V_{1}, V_{2}$, and $V_{3}$, and linear transformations $L_{1}: V_{1} \rightarrow V_{2}$ and $L_{2}: V_{2} \rightarrow V_{3}$, where the dimensions of the $V_{i}$ are all $n$, $\left|\operatorname{det} L_{2} L_{1}\right|=\left|\operatorname{det} L_{2}\right| \cdot\left|\operatorname{det} L_{1}\right|$.

Integrals on compact Riemannian manifolds. Let $M$ be a Riemannian manifold, and for each point $x \in M$, let $g_{x}$ denote the inner product on the tangent space $T_{x} M$. Suppose $f$ is a smooth real-valued function on $M$ whose support is a compact subset of $U$ for some chart $(U, V, \phi)$. Define the integral of $f$ on $U$ by:

$$
\begin{equation*}
\int_{U} f=\int_{V} f \circ \phi^{-1}\left|\operatorname{det} d \phi^{-1}\right| \tag{3.22}
\end{equation*}
$$

Since the tangent spaces of $M$ are inner product spaces (recall that $M$ is a Riemannian manifold), and $V$ as a subset of $R^{n}$ has a canonical inner product, the expression $\left|\operatorname{det} d \phi^{-1}\right|$ is well-defined. ${ }^{21}$ Furthermore, suppose the support of $f$ is contained in both $U_{1}$ and $U_{2}$ for some charts $\left(U_{1}, V_{1}, \phi_{1}\right)$ and $\left(U_{2}, V_{2}, \phi_{2}\right)$. Let $U$ be the intersection of $U_{1}$ and $U_{2}$, and let $W_{i}=\phi_{i}(U)$. Then $f$ is also supported in $U$, and:

$$
\begin{equation*}
\int_{U_{1}} f=\int_{U} f \tag{3.23}
\end{equation*}
$$

[^24]\[

$$
\begin{align*}
& =\int_{W_{1}} f \circ \phi_{1}^{-1}\left|\operatorname{det} d \phi_{1}^{-1}\right|  \tag{3.24}\\
& =\int_{W_{2}}\left(f \circ \phi_{1}^{-1} \circ\left(\phi_{1} \circ \phi_{2}^{-1}\right)\right)\left|\operatorname{det} d \phi_{1}^{-1}\right| \cdot\left|\operatorname{det}\left(d\left(\phi_{1} \circ \phi_{2}^{-1}\right)\right)\right|  \tag{3.25}\\
& =\int_{W_{2}} f \circ \phi_{2}^{-1}\left|\operatorname{det} d \phi_{2}^{-1}\right|  \tag{3.26}\\
& =\int_{U_{2}} f \tag{3.27}
\end{align*}
$$
\]

and the integral $\int_{U} f$ is well-defined. But by our earlier argument using partitions of unity in §3.4.1, this means the integral is well-defined on manifolds.

One last note: This discussion actually skirts the issue of convergence. While each local integral $\int \rho_{i} f$ is well-defined because $\rho_{i} f \circ \phi^{-1}$ has compact support in $V$, there is nothing that guarantees that the sum $\int f=\sum_{i} \int \rho_{i} f$ converges. In general, it does not always converge, and one often requires that the partition of unity be finite. A manifold for which there exists a finite partition of unity must be compact. ${ }^{22}$

## Implementation in Scheme

Having gone to such lengths to discuss integration on manifolds, the reader might suspect that one could build an elaborate computational scheme for computing integrals of realvalued functions over Riemannian manifolds. However, in practice it often happens that the manifold in question is an open subspace of $R^{n}$ (or, in cases where boundary conditions are necessary, closures of open subspaces of $R^{n}$ ). In such cases, it suffices to use the Euclidean structure directly to define integrals, and the code for manipulating finite element basis functions implement the ideas in the previous section automatically. As a complete implementation of these ideas is not necessary for testing the use of multiple coordinate systems to solve PDEs, such routines have not been implemented at this time. The purpose of this treatment of integration has primarily been for the theoretical insight it provides; like partial differential operators in $\S 3.1$, the code used in this chapter can seem ad-hoc and confusing without a proper framework in mind.

### 3.4.2 More about boundaries

This section picks up where $\S 2.1 .8$ left off: In order to discuss the computational solution of elliptic boundary-value problems on manifolds, it is necessary to build a computational framework for working with boundary charts and manifolds with boundaries. This section discusses the implementation of manifolds with boundaries in Scheme.

[^25]

Figure 3-13: A boundary chart for the solid disc in the plane.


Figure 3-14: Another boundary chart for the solid disc in the plane.

Add-boundary-to-chart and make-boundary-chart are the primary procedures for computing with boundaries of manifolds. Add-boundary-to-chart takes as arguments a chart ( $U, V, \phi$ ), an index $i$, and an optional argument level $L$, and declares the subset $\left\{p \in U: x_{i}=L, x=\phi(p)\right\}$ of $V$ the boundary of the chart. This creates boundary charts for the original manifold. While this is a slight deviation from the definition of boundary charts in §2.1.8, it is clearly equivalent and slightly simplifies programming with these abstractions. Make-boundary-chart ${ }^{23}$ then constructs a chart for the boundary manifold out of a boundary chart for the original manifold.

The actual construction of a manifold with boundary can be rather messy, so the code is omitted here. Figures 3-13 through 3-15 show three charts that cover the solid disc $\left\{x \in R^{2}:|x| \leq 1\right\}$, the first two being boundary charts and the third covering the center of the disc. Figure 3-16 shows how these charts overlap.

[^26]

Figure 3-15: A third chart for the solid disc in the plane; this one covers only the interior and does not intersect the boundary.


Figure 3-16: All three charts together, covering the unit disc.

### 3.4.3 Computing with finite elements on manifolds

The previous sections, together with Appendix A, contain the material necessary for developing finite elements on manifolds. Since the subject of partial differential equations is sufficiently vast and complicated that many issues of theoretical and computational importance need to be resolved in very different fashions in different cases, the programs have been designed to provide only a logical skeleton into which all the components fit, and the individual components, such as the finite element basis functions and their integraton over domains, are very flexible. Consequently, the best way to understand the algorithms and representations used for these computations is to examine how it works for a concrete example; otherwise the program can seem excessively abstract.

The main program is divided into three parts: The first is a finite element program (FEM) that performs the local finite element assembly, etc., and has no knowledge of manifolds. Indeed, this portion stands on its own as a finite element PDE solver over Euclidean spaces. The second part is a set of additions to the manifold code developed in the Chapter 2 that help manage geometric structures such as boundaries for the sake of setting boundary values and solving PDEs. Finally, the third part is a set of tools that oversee the finite element assembly process on manifolds, and has various routines that combine local equations into global ones in different ways.

The primary example in this section, as in Appendix A, is the boundary value problem for Laplace's equation. The domain of solution is the unit disc (see Figure 3-16), which was given the structure of a manifold with three charts (see Figures 3-13 through 3-15). As stated before, this is a natural problem because of its simplicity and importance in physical problems. Furthermore, one can easily derive analytical solutions for simple boundary values, and for more complicated boundary values traditional finite element methods (over subspaces of Euclidean space) are known to perform reasonably well.

### 3.4.4 Local finite-elements

First, let us discuss the local finite element program. It depends on explicit computational representations of nodes and elements and uses these abstractions to isolate different stages in the finite element assembly process and to clarify the interdependence of different components. In this discussion, unless explicitly stated, all objects exist in Euclidean spaces.

In this system, nodes are objects that have coordinates, carry values, and have some extra fields (such as various ID numbers that identify them from other nodes in the ensemble), and flags that identify them as boundary nodes. Since each element object also keeps track of the nodes that they contain, each node is also assigned a local ID by the element. Conversely, each node must also keep track of the elements to which they belong.

In terms of elements and nodes, then, the finite element assembly process can be expressed rather concisely as follows:

```
(define (assemble-equations source nodes)
    ;; SOURCE is a function from R^2 to R, and NODES is expected to be a vector.
    (let* ((ncount (vector-length nodes))
            (bcount 0)
            (index-map (make-vector ncount)))
    ;; First, assign each node an index and count the number of boundary nodes.
    (do ((i O (+ i 1)))
            ((>= i ncount))
        (node:set-id! (vector-ref nodes i) i)
        (if (node:boundary? (vector-ref nodes i))
            (set! bcount (+ bcount 1))))
    ;; Next, create a mapping from node indices into matrix row number. (The
    ;; matrix has one row per interior node.)
    (let loop ((i 0) (row 0))
        (if (< i ncount)
            (if (node:boundary? (vector-ref nodes i))
                    (begin
                        (vector-set! index-map i #f)
                                (loop (+ i 1) row))
                            (begin
                        (vector-set! index-map i row)
                                (loop (+ i 1) (+ row 1))))))
    ;; Loop over the nodes to create row entries:
    (let* ((icount (- ncount bcount))
                (big-matrix (make-sparse-matrix icount (1+ icount))))
        (do ((i O (+ i 1)))
            ((>= i ncount))
            (if (not (node:boundary? (vector-ref nodes i)))
            (let ((row (vector-ref index-map i)))
                ;; Compute the source term for this row:
                (sparse-matrix-set! big-matrix row icount
                                    (node:compute-source (vector-ref nodes i)
                                    source))
                    ;; Combine boundary values:
                (for-each
                (lambda (pair)
                        (let ((id (car pair))
                            (val (cadr pair)))
                            (if (node:boundary? (vector-ref nodes id))
                            (sparse-matrix-set!
                        big-matrix row icount
```



Figure 3-17: As defined in Appendix A, each node corresponds to a vertex in a triangulation, and to each node $i$ there corresponds a finite element basis function $\phi_{i}$. The support of $\phi_{i}$ is the union of all those elements adjacent to node $i$, and hence the intersection of the supports of two basis functions $\phi_{i}$ and $\phi_{j}, i \neq j$, consists of a union of elements as well. Element:compute-integrals, when given an element $E$ and an index $i$ belonging to $E$, returns the set of all integrals of the form $\int_{E} \phi_{i} \cdot L \phi_{j}$ for all $j$ that are neighbors of $i$.

```
    (- (sparse-matrix-ref big-matrix row icount)
    (* val (node:get-value (vector-ref nodes id)))))
(sparse-matrix-set! big-matrix row
    (vector-ref index-map id) val))))
    (node:assemble (vector-ref nodes i))))))
big-matrix))(
```

Note that this FEM assembly program does not actually compute the integrals, but calls node: assemble to recursively construct the appropriate coefficients and combine them.

```
(define (node:assemble node)
    (let ((l (append-map
                            (lambda (element index)
                            (element:compute-integrals element index))
            (node:get-elements node)
            (node:get-local-ids node))))
    ;; ELEMENT:COMPUTE-INTEGRALS returns a list of pairs, where each pair takes
    ;; the form (node-id . coefficient). MERGE-TERMS then sorts and adds up
    ;; coefficients that have the same ID.
        (merge-terms l + (lambda (x y) (< (car x) (car y))))))
```

Node: assemble calls element : compute-integrals, which returns a list of pairs of the form (node-index . integral), which represent the element's contribution to the finite element integrals involving the basis function centered at the given node. More precisely,
let $i$ be the index of the current node, and let $j$ denote the index of one of its neighbors, and let $E$ denote an element shared by these two nodes (see Figure 3-17). ${ }^{24}$ Then element: compute-integrals and node: assemble compute and return a list of pairs of the form:

$$
\begin{equation*}
\left(j, \int_{E} \phi_{i} \cdot L \phi_{j}\right) . \tag{3.28}
\end{equation*}
$$

Merge-terms then adds up contributions corresponding to the same node index $j$.
This shows that all routines for integrating basis functions and dealing with the differential operator can be isolated in the element abstraction: The FEM assembly program and the nodes exist merely for "book-keeping" purposes, and all the information about the geometry of the domain and the action of the differential operator are encapsulated in the elements. The element abstraction thus isolates all the components that need to be changed in order to modify the type of basis functions used and the method used to integrate them; this simplifies the method's application to manifolds.

Constructing elements and differential operators. The construction of elements is much more complicated than the mere packaging of data. It takes as arguments three procedures for constructing important data structures. The first of these, make-operator, takes a list of nodes and returns a list of structures that represent the differential operator (or an approximation thereof) over the element described by the given nodes. It is organized in such a convoluted way because oftentimes it is useful to have the ability to approximate differential operators with variable coefficients with operators whose coefficients are locally constant. To facilitate this, operators need to "know" the element over which it is operating, and hence we have the make-operator constructor.

To complicate matters even more, it is often useful to split a differential operator $L$ into three components: An $m$-vector-valued differential operator $L_{\text {left }}$, a second $m$-vector-valued opreator $L_{\text {right }}$, and a bilinear form (on vectors in $R^{m}$ ) $\langle$,$\rangle , satisfying the equation$

$$
\begin{equation*}
\int\left\langle L_{l e f t} f, L_{r i g h t} f\right\rangle=\int(f \cdot L f) \tag{3.29}
\end{equation*}
$$

where $f$ is an arbitrary differentiable function of compact support, which, for example, can be a basis function. ${ }^{25}$ The reason for this is that finite element basis functions are often piecewise polynomial functions, and hence are only differentiable finitely many times. In

[^27]general, the more degree of differentiability one requires, the higher the order of the polynomials. Since higher-order polynomials require more nodes, their storage and manipulation require more computational resources. Conversely, one can often reduce the amount of data needed by reducing the order of the polynomials. This is possible if one integrates by parts and split the differential operator into two parts. For example, the Laplacian is often represented by the gradient operator $L_{\text {left }}=L_{\text {right }}=\nabla$, which when integrated by parts to yield the (negative) Laplacian operator $-\nabla^{2}$; this allows the use of basis functions that are continuous with piecewise-continuous first partials, such as piecewise-linear functions. ${ }^{26}$

Thus, make-operator returns left-op, right-op, and combine, which correspond to $L_{\text {left }}, L_{\text {right }}$, and $\langle$,$\rangle , respectively. This structure also allows the use of the usual repre-$ sentation of differential operators: Just let right-op compute the differential operator, let left-op be the identity operator, and replace combine with a function product operation.

The other two arguments of element-maker are simpler: Make-integrator takes as argument a list of nodes and returns a procedure capable of integrating basis functions over the element defined by those nodes, and make-basis-function creates a basis function data structure. Note that basis functions are generally abstract data structures that represent mathematical functions, not computational procedures, and their representations are completely flexible: The entire program works so long as make-integrator and make-basis-function agreed a priori upon a consistent representation of basis functions. In practice, as stated above, piecewise polynomial basis functions are often used because their images under differential operators are easy to compute, as are their integrals.

```
;;; Note that this implicitly assumes that elements are the convex hull of
;;; their vertices.
;;; The (meta-)constructor for element-constructors:
(define (element-maker make-operator
    make-integrator
    make-basis-function)
    ;; MAKE-INTEGRATOR should take as argument a list of nodes, and returns a
    ;; procedure that takes a variable number of functions (at least 1) and
    ; integrates their product over the domain specified implicitly as the
    ;; convex hull of the vertex nodes.
    ; MAKE-BASIS-FUNCTION should take as argument a list of nodes and the index
    ;; of the node that is to be the center of the basis function, and return
    ;; some structure representing basis functions.
    ;; We place no restrictions on the representation of functions over elements,
    ;; so long as the particular instances of MAKE-BASIS-FUNCTION and
    ;; MAKE-INTEGRATOR agree a-priori on the representation.
```

[^28]```
; MAKE-OPERATOR should take a list of nodes and return LEFT-OP, RIGHT-OP,
and COMBINE procedure, satisfying (INTEGRATE (COMBINE (LEFT-OP F)
(RIGHT-OP G))) = (INTEGRATE F (OP G)), i.e. implement integration by parts
so that basis functions can be less smooth.
The list of nodes facilitates the interpolation of variable coefficients
in the operator. This may not be a good interface, as it makes artificial
assumptions on the contract between basis functions and operators (as is
the explicit use of LEFT-OP and RIGHT-OP).
(define (make-element vertex-nodes other-nodes)
;; The first part stores the coefficients, the second part the source
; terms. What about coefficients? Maybe we should incorporate the
;; source term into the differential operator.
(let* ((nodes (append vertex-nodes other-nodes))
(number-of-nodes (length nodes))
    (n-choose-2 (choose (+ number-of-nodes 2) 2))
    (element
            (vector (make-vector n-choose-2 0)
                    (make-vector n-choose-2 0)
                    vertex-nodes
                    other-nodes
                    (make-vector number-of-nodes #f)))
        (op (make-operator nodes)))
    ;; Add the element to the nodes:
    (let loop ((nodes nodes) (i 0))
        (if (not (null? nodes))
            (begin
                (node:add-element (car nodes) element i)
                (loop (cdr nodes) (+ i 1)))))
    ;; Initiailize elements (and hiding the hair)...
    (let ((integrate (make-integrator vertex-nodes))
            (local-form (operator:get-local-form op)))
        (do ((i 0 (+ i 1)))
            ((>= i number-of-nodes))
            (element:set-basis-function!
            element i (make-basis-function nodes i)))
        (do ((i 0 (+ i 1)))
            ((>= i number-of-nodes))
        (let ((f (element:get-basis-function element i)))
            (do ((j i (+ j 1)))
                            ((>= j number-of-nodes))
                            (let ((g (element:get-basis-function element j)))
                            (element:set-coeff! element i j
                            (integrate (local-form f g)))
                            (element:set-source! element i j (integrate f g)))))))
element))
```

```
make-element)
```

This also shows that, as a matter of efficiency, elements can be called on to evaluate the integrals first when one constructs the domain. One can then work with different boundary values (or source functions, in the case of Poisson's equation) without recomputing the finite element integrals.

### 3.4.5 Basic FEM algorithm on manifolds

There are two top-level programs that manage the computation of finite element equations on manifolds. The first program manages mesh generation and element construction, while the second program uses these elements and the local finite-element assembly program to generate a sparse matrix that represents the discretized system of linear equations.

What follows is the main portion of the code for the first program: ${ }^{27}$

```
(define (pde:domain-maker generate-node-lists process-complex)
    (lambda (M
            make-vertices
            make-extra-nodes
            tesselate
            - argl)
        ;; First, make the bounding nodes of the convex domain, and then
        ;; triangulate and make the extra nodes:
        (let ((atlas (manifold:get-finite-atlas M)))
            (if (not atlas)
            (error "Error: Can only do FEM with finite atlases."))
        (write-line '(tesselating domain...))
        ;; Do something more complicated here to reduce the overlap:
        (let loop ((charts atlas)
                            (node-lists (generate-node-lists make-vertices atlas argl)))
        (if (not (null? charts))
                            ;; TESSELATE should return a list of lists, where each list
                            ; contains the elemental faces of a given dimension (in some given
            ; ; polytope). In the planar case, this reverses the convention in
            ;; fem.scm: The list should be sorted by dimension in *descending*
            ;; order.
```

[^29]```
        (let* ((chart (car charts))
                (nodes (car node-lists))
                (complex (process-complex (tesselate nodes) (cdr charts)))
                (extra-nodes (make-extra-nodes complex)))
    ;; By default, use FEM-DISCRETIZE. Can replace with others.
    (make-pde-chart chart extra-nodes fem-discretize complex)
    (loop (cdr charts) (cdr node-lists)))))
;; Construct elements. We don't need to explicitly mark boundaries
;; because manifolds should already have such structures defined.
(lambda (operator make-integrator make-basis-function)
    (let ((element-maker (pde:element-maker operator
                                    make-integrator
                                    make-basis-function)))
    (write-line '(constructing elements...))
    (for-each
        (lambda (chart)
            ;; Construct the elements:
            (write-line
            '(making ,(length (complex->faces (chart:get-complex chart)))
                        elements...))
            (let* ((make-element (element-maker chart))
                    (new-elements (map make-element
                                    (complex->faces
                                    (chart:get-complex chart))
                                    (chart:get-extra-nodes chart))))
            (chart:set-elements! chart new-elements)))
        atlas())))(
```

This program is a "meta-constructor" for domain constructors, and returns a procedure that adds sufficient structure to a given manifold (such as nodes and local triangulations, etc.) that finite element analysis can be performed. It provides only a logical skeleton into which other procedures fit; the real work is done by procedures like generate-node-lists, process-complex, make-vertices, make-extra-nodes, and tesselate.

Given the appropriate procedures for constructing nodes and meshes on charts, the program generates nodes and constructs meshes for each chart. Then, some of the nodes are "pruned" away to control the size of the number of nodes shared between charts. ${ }^{28}$ The expression (make-pde-chart chart extra-nodes fem-discretize complex) attaches extra data structures to chart, so that in a later stage the information obtained here can

[^30]be used to construct the elements. ${ }^{29}$ Finally, yet another procedure is returned that takes the information obtained above, as well as representations of the differential operator, constructors for basis functions, and integrators of basis functions, and actually constructs the elements.

Having constructed elements and prepared the domain of solution for finite element analysis, the second top-level program generates the discretized equations given boundary data and a source function:

```
;;; Given a domain with constructed elements, a source function, and a boundary
;;; value function, produce the appropriate discretized equation. The nodes
;;; are left with indices that specify their corresponding row in the matrix.
(define (pde:equation-maker merge-equations)
    (lambda (domain source boundary-value . extra-args)
    ;; EXTRA-ARGS gives us finer control over the discretization.
    ;; DOMAIN should be a manifold that already has PDE structures constructed.
    ;; Hence, it contains information about the operator (through the elements
    ;; in its discretized charts).
        BOUNDARY-VALUE is irrelevant for domains without boundary. Just specify
    ;; anything (but do put in something).
    (let* ((M domain)
            (charts (manifold:get-finite-atlas M))
            (nodes (list->vector (append-map chart:get-nodes charts)))
            (ncount (vector-length nodes)))
        ;; CHART:DISCRETIZE-PDE should return a list of linear equations. First,
        ;; set the boundary values:
        (write-line '(,ncount nodes generated...))
        (write-line '(setting boundary values...))
        (do ((i 0 (+ i 1)))
            ((>= i ncount))
        (let ((node (vector-ref nodes i)))
            (if (node:boundary? node)
                (node:set-value! node (boundary-value node)))))
        ;; Next, compute the local equation systems:
        (write-line '(computing ,(length charts) local systems of equations...))
        (let ((equations (append-map
                        (lambda (chart)
                            (chart:discretize-pde chart source extra-args))
                            charts)))
```

            ; ; Compute constraints:
    [^31]

Figure 3-18: A point $y$ in some chart, with its "neighbors" $n_{1}$ through $n_{5}$, which are the nodes belonging to the element that contains $y$. (Since the elements of a triangulation partition whatever chart they cover, each point lies in only one element except for points lying in the boundaries of elements.) This figure is drawn using the coordinate system of $C_{2}$, and it illustrates two charts, with the dotted lines outlining the element (of $C_{1}$ ) to which the point $x=\phi_{1}(p)$ belongs while the dashed line delineates the boundary of the image $\phi_{2}\left(U_{1} \cap U_{2}\right)$ of $C_{1}$ in this coordinate system. The dotted element has a curved boundary because the entire image is seen in the coordinate system of $C_{2}$.

```
(write-line '(merging local equations...))
(merge-equations domain equations)))))
```

Once again, this program only serves as a logical skeleton. All the major components of the programs, such as the procedure merge-equations, are easily modifiable. This facilitates the testing of different methods for performing these tasks. Indeed, the following sections will explore a couple different implementations of generate-node-lists, process-complex, and merge-equations that control how much charts overlap and how local equations are merged into a global set of equations.

### 3.4.6 Interpolation between charts

Finally, we come to the most delicate part of the problem: How does one actually combine local equations into a global set of equations? This process is determined by the procedures generate-node-lists, process-complex, and merge-equations, which are passed into pde:domain-maker and pde:equation-maker as arguments.

As mentioned at the beginning of $\S 3.4$, one natural idea is the following: Let $C_{1}=$ ( $U_{1}, V_{1}, \phi_{1}$ ) and $C_{2}=\left(U_{2}, V_{2}, \phi_{2}\right)$ be charts on the manifold $M$. Suppose the $i$ th node in the discretized domain is at the point $p \in M$, and that $p$ lies in the intersection $U_{1} \cap U_{2}$. Let $x=\phi_{1}(p)$ be the coordinate vector corresponding to $p$ in $V_{1}$, and let $y=\phi_{2}(p)$ be the coordinate vector corresponding to $p$ in $V_{2}$. Then one could simply constrain the unknown value at $x, a_{i}$, to the value at the corresponding point $y=\phi_{2}(p)$, interpolated from basis functions in $C_{2}$. More precisely, let $n_{i}$ be the indices of the nodes in the element $E$ containing $y$ in $C_{2}$ (see Figure 3-18). Then the constraint we want is:

$$
\begin{equation*}
a_{i}=\sum_{k} \phi_{n_{k}}(y) a_{n_{k}}, \tag{3.30}
\end{equation*}
$$

where $a_{j}$ denotes the sample value $u\left(p_{j}\right)$ of the approximate solution at the $j$ th node, with position $p_{j}$. Since the expressions $\phi_{n_{i}}(y)$ can be computed without reference to any unknowns, we see that this is a linear equation relating unknown nodal values. Thus, the constraints generated this way may simply be "appended" onto the system of locallydiscretized equations for each chart, each of which is also linear. Doing this for a sufficiently large number of nodes that lie in the overlap of two charts should generate enough extra equations to relate the local equations derived for each chart. It should be noted that this process of appending constraints produces overdetermined systems, for which exact solutions generally do not exist. Thus, a least-squares approximation is the best one could do. This can be done by computing the normal equations, which finds an approximate solution to the overdetermined system $A x=b$ by minimizing the magnitude of the error $A x-b$ with respect to the natural inner product of Euclidean space.

A program that implement a general procedure for combining equations and constraints into a large matrix is shown below. It relies on make-constraints to construct the constraint equations, and the main body of the program performs the tedious task of constructing the matrix row by row:

```
;;; This complicated-looking procedure performs the simple task of forming a
;;; sparse matrix out of locally-discretized equations and constraint
;;; equations. The constraints are generated with the help of
;;; MAKE-CONSTRAINTS.
(define (append-constraint-equations make-constraints)
    (lambda (domain equations)
    ;; First, set IDs and clear hidden states:
    (write-line '(setting node ids...))
    (let loop ((id 0) (nodes (manifold:get-nodes domain)))
        (if (not (null? nodes))
            (let ((node (car nodes)))
                (node:set-constraint! node #f)
```

```
    (if (node:boundary? node)
        (begin
            (node:set-id! node 'boundary-node!)
                (loop id (cdr nodes)))
            (begin
                (node:set-id! node id)
                (loop (+ id 1) (cdr nodes)))))))
;; Next, generate constraints:
(write-line '(generating constraints...))
(with-values
    (lambda () (make-constraints domain))
    (lambda (c-count clists)
    (let* ((eq-count (length equations))
                (m (+ eq-count c-count))
                (n (+ eq-count 1)))
            (write-line '(constructing a matrix of dimension (,m ,n)...))
            (let ((mat (make-sparse-matrix m n)))
                ;; First, copy the equations:
            (write-line '(copying ,eq-count equations...))
            (for-each
                    (lambda (eq)
                    (let ((i (equation:get-id eq)))
                    (sparse-matrix-set!
                        mat i eq-count (equation:get-constant eq))
                        (for-each
                            (lambda (term)
                            (sparse-matrix-set! mat i (term:get-id term)
                                    (term:get-coeff term)))
                                    (equation:get-terms eq))))
            equations)
            ;; Next, copy the constraints:
            (write-line '(copying ,c-count constraints...))
            (let next-clist ((i eq-count) (clists clists))
                (if (null? clists)
                mat
                (let next-constraint ((clist (car clists)) (i i))
                        (if (null? clist)
                            (next-clist i (cdr clists))
                    (let ((constraint (car clist)))
                                    (sparse-matrix-set!
                                    mat i eq-count (equation:get-constant constraint))
                                    (for-each
                                    (lambda (term)
                                    (sparse-matrix-set! mat i (term:get-id term)
                                    (term:get-coeff term)))
                                    (equation:get-terms constraint))
                                    (next-constraint (cdr clist) (+ i 1)))))))))))))
```

While the basic idea of interpolating unknown values from other charts is simple enough, there are some unresolved details here: For one thing, what does it mean to create constraints for "a sufficiently large number of nodes"? Is it necessary to create constraints for all nodes in the overlap, or just some specially-chosen interpolation nodes? Which ones should we use? Furthermore, let $C_{1}, C_{2}$, and $C_{3}$ be charts, let $p \in M$ be a point contained in all three charts, and let $x_{i}=\phi_{i}(p)$ be the image of $p$ in the chart $C_{i}$. Since there are three charts, there are three different constraints we can generate using the recipe above by considering different pairs of charts. Is it better to generate all three constraints, or to generate only one or two of them? Since the basis functions and triangulations in different charts are by no means related to each other, one would expect that the constraints are independent of each other, and hence this is a non-trivial question. Clearly, this problem extends in general to any node that lies in more than two charts, and if not all possible constraints are to be generated, then which ones should we use?

Since there are many possible choices here and no obvious candidate, it seems reasonable to try a couple of different ideas and see how well they perform:

1. Generate all constraints for all nodes in the overlaps between all pairs of charts.
2. Put the set of all charts in some linear ordering, and generate all constraints for all nodes in the overlaps of adjacent charts (in the given ordering).

The following program, make-all-constraints, implements the first of the ideas enumerated above by generating all constraints between all pairs of charts:

```
(define (make-all-constraints domain)
    (let ((constraints
                (append-map
                (lambda (pair)
                    (let ((chart-1 (car pair))
                        (chart-2 (cadr pair)))
                    (append (constrain-all-nodes chart-1 chart-2)
                                    (constrain-all-nodes chart-2 chart-1))))
                (pairs (manifold:get-finite-atlas domain)))))
        (values (length constraints) (list constraints))))
(define (constrain-all-nodes chart-1 chart-2)
    (append-map
        (lambda (node)
        (if (node:boundary? node)
            '()
                        (let ((eq (chart:pointwise-constraint node chart-2)))
                        (if eq
                                    (list eq)
                            '())))
    (chart:get-nodes chart-1)))
(define (pairs 1)
    (let loop ((l l) (result '()))
```

```
(if (null? 1)
    result
    (loop (cdr l)
        (let ((a (car l)))
                                (let loop ((l (cdr 1)) (result result))
                                (if (null? 1)
                    result
                        (loop (cdr 1) (cons (list a (car 1)) result))))))))
```

It can be passed into append-constraint-equations to construct the constraints. This program is rather straightforward: For all pairs of distinct charts, generate all possible constraints from nodes in the overlap between these two charts.

The next program implements the second idea, which involves ordering the charts. Since atlases are represented by Scheme lists, the implicit ordering of lists is used to linearly order the charts.

```
(define make-all-ordered-constraints
    (let ((exists? (lambda (node) #t)))
        (lambda (domain)
            (let* ((charts (manifold:get-finite-atlas domain))
                    (result-1 (charts->constraints charts exists?))
                    (result-2 (charts->constraints (reverse charts) exists?)))
                (values (+ (car result-1) (car result-2))
                            (append (cadr result-1) (cadr result-2)))))))
;;; The charts come in a ordered list, so that implicit ordering is used as the
;;; linear ordering we need.
(define (charts->constraints charts good-node?)
    ;; The predicate GOOD-NODE? lets the calling procedure control which nodes to
    ;; use. In this case, it simply uses all non-boundary nodes
    (let next-chart ((charts charts)
                                    (count 0)
                                    (clists '()))
        (if (null? charts)
                (list count clists)
                ;; Go through each node in the chart and check for constraints:
                (let ((chart (car charts)))
                        (let next-node ((nodes (chart:get-nodes chart))
                            (count count)
                            (clist '()))
                (if (null? nodes)
                        (next-chart (cdr charts) count (cons clist clists))
                        (let ((node (car nodes)))
                            ;; We only want to create constraints for nodes that do not
                    ;; already have a constraint:
                            (if (and (good-node? node)
                            (not (node:get-constraint node))
```

```
                    (not (node:boundary? node)))
(let ((eq (make-constraint node (cdr charts))))
    (if eq
                (next-node (cdr nodes) (+ count 1) (cons eq clist))
                (next-node (cdr nodes) count clist)))
(next-node (cdr nodes) count clist)))))))))
(define (make-constraint node charts)
    (let loop ((charts charts))
        (if (null? charts)
            #f
            (let ((eq (chart:pointwise-constraint node (car charts))))
                (if eq
                            eq
                            (loop (cdr charts)))))))
```

This program is a bit more complicated: Charts->constraints takes a list of charts and produces a list of constraints, such that a node $n$ in a chart $C_{i}$ is constrained to a chart $C_{j}$ if and only if $j$ is the least integer greater than $i$ such that $C_{j}$ contains $n$. The same procedure is then called again to construct constraints in the reverse direction, so that constraints exist for charts adjacent in this linear ordering (or as close to being adjacent as possible).

Both of the programs above call chart:pointwise-constraint, which can be implemented thusly:

```
(define (chart:pointwise-constraint node chart)
    ;; The coefficients of a linear constraint for some node x should simply be
    ; the value at p of the basis function centered at x. This linearity
    ;; depends only on the fact that the solution is approximated by a linear
    ;; combination of basis functions.
    (if (chart:member? (node:get-point node) chart)
        (let* ((x (chart:point->coords (node:get-point node) chart))
                            (element (chart:coords->any-element x chart)))
        (if element
            (let loop ((nodes (element:get-nodes element))
                            (i 0)
                            (const 0)
                            (terms (list (make-term node -1))))
                (if (null? nodes)
                    (begin
                        (node:set-constraint! node chart)
                            (make-equation node const terms))
                    (let ((neighbor (car nodes))
                            (coeff (evaluate-basis-function
                            (element:get-basis-function element i) x)))
                            (if (node:boundary? neighbor)
                    (loop (cdr nodes)
                                    (+ i 1)
                                    (- const (* (node:get-value neighbor) coeff))
                            terms)
                            (loop (cdr nodes)
                                    (+ i 1)
```

```
                                    const
                                    (cons (make-term neighbor coeff) terms))))))
    #f))
#f))
```

It simply finds an element of chart to which node belongs, and loops through the nodes of the given element to evaluate the basis functions and compute the coefficients.

### 3.4.7 Some numerical results.

To test the ideas above, we should perform some numerical experiments. The canonical problem on which every FEM program should cut its teeth is the boundary value problem for Laplace's equation. For us, the domain will be the unit disc $\left\{(x, y) \in R^{2}: x^{2}+y^{2} \leq 1\right\}$ in the plane (see Figure 3-16), with the boundary value

$$
\begin{equation*}
f(\theta)=\cos (2 \theta) \tag{3.31}
\end{equation*}
$$

Using the angle addition formula for cosines, one finds that $f(\theta)=\cos ^{2} \theta-\sin ^{2} \theta$. But the function $g(x, y)=x^{2}-y^{2}$ satisfies Laplace's equation everywhere, and $g(\cos \theta, \sin \theta)=f(\theta)$ for all $\theta$, so $g$ must be the true solution corresponding to the boundary data $f$. This gives us a convenient problem on which to test the ideas above and an exact solution against which to compare answers.

So far we have only seen how to implement the auxiliary procedure merge-equations: The constructor append-constraint-equations, given either make-all-constraints or make-all-ordered-constraints, should return a procedure that constructs constraint equations for pde: equation-maker. But we also need to implement the auxiliary procedures for pde:domain-maker. To do this, we need the procedures make-nodes-for-each-chart and do-nothing-to-complex, which, as their names suggest, are very simple procedures. We will need more complicated auxiliary procedures later on, but these simple programs suffice for now.

The definitions of key data structures are shown below:

```
;;; The procedure that prepares the domain for the PDE solver:
(define pde:make-simple-domain
    (pde:domain-maker make-nodes-for-each-chart do-nothing-to-complex))
;;; Two different ways for generating constraints:
(define combine-equations-with-overlap1
    (pde:equation-maker
        (append-constraint-equations make-all-constraints)))
(define combine-equations-with-overlap2
```

```
    (pde:equation-maker
    (append-constraint-equations make-all-ordered-constraints)))
;;; Construct the domain of the PDE:
(define disc
    (make-ball 2 make-spherical-sphere))
;;; Construct the Laplacian. Note that OPERATOR:IMBEDDED-POLY-OP simply
;;; packages the operators left-op, right-op, and combine. This splits the
;;; Laplacian into two parts through integration by parts.
(define imbedded-poly-laplacian
    (make-operator
        disc
        (operator:imbedded-poly-op
        poly-gradient
        poly-gradient
        (lambda (v w) (basis:scalar* -1 (basis:dot v w))))))
;;; The true solution of Laplace's equation that we're trying to approximate:
(define (test-function node)
    (let ((x (x-coord-map node))
            (y (y-coord-map node)))
        (- (square x) (square y))))
```

Having defined the necessary auxiliary procedures, we can now try to compute the solution of Laplace's equation:

```
;;; Prepare the domain for FEM:
(define make-test-domain
    (pde:make-simple-domain disc ;; The domain.
                                    make-mesh ;; A generic vertex generator.
                                    make-no-extra-nodes ; ; No edge nodes, just vertices.
                    planar-triangulate ;; A generic mesh generator.
                            ;; Some extra parameters:
                            '(rectangular 10 5)
                            '(spherical 5 10)))
(tesselating domain...)
;Value: make-test-domain
;;; Construct the elements and initialize finite element integrals:
(make-test-domain
    ;; The Laplacian we just constructed.
    imbedded-poly-laplacian
;; Integrates directly in Euclidean space -- It cheats!
make-triangular-imbedded-integrator
;; Make some generic piecewise-polynomial basis functions.
```

```
pde:make-imbedded-poly-basis-function)
(constructing elements...)
(making 72 elements...)
process time: 4880 (4470 RUN + 410 GC); real time: 5744
(making 72 elements...)
process time: 4960 (4540 RUN + 420 GC); real time: 5761
(making 70 elements...)
process time: 4810 (4370 RUN + 440 GC); real time: 5616
;No value
;;; Assemble the equations, generate constraints, and build the matrix
;;; equation:
(define mat1
    (combine-equations-with-overlap1 disc ;; The domain again.
                                    0-function ;; No source term.
                                    test-function)) ;; The true solution.
(141 nodes generated...)
(setting boundary values...)
(computing 3 local systems of equations...)
(40 equations generated for 50 nodes.)
(40 equations generated for }50\mathrm{ nodes.)
(41 equations generated for 41 nodes.)
(merging local equations...)
(setting node ids...)
(generating constraints...)
(constructing a matrix of dimension (267 122) ...)
(copying 121 equations...)
(copying 146 constraints...)
process time: 13560 (12180 RUN + 1380 GC); real time: 20325
;Value: mat1
;;; Try the other method:
(define mat2
    (combine-equations-with-overlap2 disc 0-function test-function))
(141 nodes generated...)
(setting boundary values...)
(computing 3 local systems of equations...)
(40 equations generated for 50 nodes.)
(40 equations generated for 50 nodes.)
(41 equations generated for 41 nodes.)
(merging local equations...)
(setting node ids...)
(generating constraints...)
(constructing a matrix of dimension (235 122) ...)
(copying 121 equations...)
(copying 114 constraints...)
process time: 9800 (8860 RUN + 940 GC); real time: }1491
;Value: mat2
;;; Neither matrices are square, of course, because of the constraint
;;; equations:
(sparse-matrix-size mat1)
;Value 62: (267 122)
```

```
(sparse-matrix-size mat2)
;Value 63: (235 122)
;;; Use least-squares to solve these guys:
(define mat1 (sparse-normal-equations mat1))
;Value: mat1
(define v1 (sor mat1 1000 1.9 ))
(residual: 5.731092683758376e-16)
;Value: v1
(define mat2 (sparse-normal-equations mat2))
;Value: mat2
(define v2 (sor mat2 1000 1.9))
(residual: 7.216449660063518e-16)
;Value: v2
```

Note that we tested both constraint-generation systems without having to recompute the finite element integrals. This is one of the principal advantages of structuring the program to exploit the modularity of the finite element method.

The numerical experiments consist of a series of 11 tests, with the number of nodes ranging from 63 to 3,601 ; note that because some methods discard unnecessary nodes, the actual number used for computation may change between methods. The code used to run the numerical experiments themselves are very similar to what is shown above, and hence will not be listed separately. Table 3.4.7 shows the statistics based on results generated using make-all-constraints, while Table 3.4 .7 shows the statistics for the results generated using the other method.

Note that in both tables, the maximum absolute error remains fairly constant. This may hint at a deeper reason for the method's failure. Such issues are discussed in the next section, where this situation is analyzed a little more closely.

Figure 3-19 plots the average absolute error against the number of nodes using the data from Table 3.4.7, while Figure 3-20 does the same for Table 3.4.7.

Figure 3-21 plots the true solution, while Figure 3-22 plots one solution obtained by make-all-constraints. As one can see, they are qualitatively similar, even though numerically the solution is fairly far off.

### 3.4.8 The problem with interpolation

As can be seen from the data in the previous section, neither of the methods work very well, even though they employed relatively straightforward algorithms and obtained qualitatively reasonable results.

| Total number <br> of nodes | Absolute error |  |  | Relative error |  |
| :--- | :--- | :--- | :--- | :--- | :--- |
|  | Maximum | Minimum | Average | Maximum | Minimum |
| 121 | 0.186186 | 0.000346268 | 0.0321264 | 3.04321 | -1.98107 |
| 253 | 0.174211 | $5.49186 \mathrm{e}-05$ | 0.0296698 | 18.854 | -79.1678 |
| 433 | 0.170829 | $3.3609 \mathrm{e}-05$ | 0.0299808 | 11.44 | -14.8512 |
| 661 | 0.167295 | 0.00010959 | 0.0310626 | 30.1526 | -16.9254 |
| 937 | 0.163327 | $1.76278 \mathrm{e}-05$ | 0.031479 | 38.7539 | -44.89 |
| 1261 | 0.160884 | $3.38679 \mathrm{e}-06$ | 0.0323654 | 53.2708 | -52.5735 |
| 1633 | 0.160982 | $4.98298 \mathrm{e}-06$ | 0.0327103 | 76.6556 | -64.7096 |
| 2053 | 0.162743 | $7.24373 \mathrm{e}-06$ | 0.0334327 | 102.64 | -83.955 |
| 2521 | 0.163858 | $1.6586 \mathrm{e}-05$ | 0.0342905 | 109.606 | -131.524 |
| 3037 | 0.163727 | $8.36536 \mathrm{e}-06$ | 0.0352394 | 153.382 | -154.745 |
| 3601 | 0.165879 | $1.22342 \mathrm{e}-05$ | 0.0365282 | 200.408 | -188.901 |

Table 3.1: Statics of the results generated by make-all-constraints.

| Total number <br> of nodes | Absolute error |  |  | Relative error |  |
| :--- | :--- | :--- | :--- | :--- | :--- |
|  | Maximum | Minimum | Average | Maximum | Minimum |
| 121 | 0.192343 | 0.000764133 | 0.0337441 | 2.17113 | -1.65951 |
| 253 | 0.180941 | $5.30935 \mathrm{e}-06$ | 0.029554 | 27.2596 | -68.7197 |
| 433 | 0.166176 | $9.66714 \mathrm{e}-06$ | 0.0281985 | 13.4546 | -8.88716 |
| 661 | 0.16295 | $5.27949 \mathrm{e}-05$ | 0.0291729 | 18.9087 | -19.8467 |
| 937 | 0.15868 | $4.36369 \mathrm{e}-07$ | 0.0294747 | 34.479 | -23.4737 |
| 1261 | 0.158527 | $2.13021 \mathrm{e}-05$ | 0.0300879 | 39.5776 | -43.8798 |
| 1633 | 0.159253 | $4.22971 \mathrm{e}-06$ | 0.0305563 | 58.4892 | -56.2954 |
| 2053 | 0.157801 | $1.44311 \mathrm{e}-05$ | 0.030769 | 62.5005 | -86.9364 |
| 2521 | 0.159835 | $1.02857 \mathrm{e}-06$ | 0.031765 | 108.839 | -90.1434 |
| 3037 | 0.161139 | $1.10304 \mathrm{e}-05$ | 0.032667 | 123.56 | -132.926 |
| 3601 | 0.163639 | $1.53283 \mathrm{e}-06$ | 0.033938 | 165.795 | -163.194 |

Table 3.2: Statistics of the results generated by make-all-orderd-constraints.


Figure 3-19: Average absolute error versus number of nodes. The results were generated using make-all-constraints.


Figure 3-20: Average absolute error versus number of nodes. The results were generated using make-all-ordered-constraints.


Figure 3-21: The true solution to the disc problem. Note that this plot is generated in a fashion similar to Figures $3-5$ through 3-12: The domain is divided into a simple square grid, over which the sample values are averaged. This reduces the number of points to be plotted. The surface generated is a hyperbolic paraboloid of one sheet, as expected.


Figure 3-22: The sample solution generated by using all possible constraints.


No constraints.


Too many constraints.
Figure 3-23: Enforcing too many constraints causes basis functions to become too dependent on each other.

The main problem appears to be that the interpolation approach produces more equations than unknowns, which in general yields overdetermined systems of equations. There are two consequences of this overdetermination: First, geometrically speaking, the basis functions become too rigid. Becuase these methods enforced too many constraints on nodal values in overlaps, the basis functions in different charts become very tightly dependent on each other, and the approximate solution itself (which consists of linear combinations of basis functions) becomes too "stiff" to conform to the real solution (see Figure 3-23). As a result, much of the numerical accuracy is lost.

A second problem may be that in order to solve a large system of overdetermined system of equations,

$$
\begin{equation*}
A x=b \tag{3.32}
\end{equation*}
$$

where the number of rows of $A$ far exceeds its number of columns, one would normally have to compute the normal equations: ${ }^{30}$

$$
\begin{equation*}
A^{T} A=A^{T} b \tag{3.33}
\end{equation*}
$$

Now, this should look somewhat familiar. It is, in fact, our friend from §3.3.2, where the

[^32]

Figure 3-24: Only nodes near the edge in their own charts are allowed to become interpolation nodes. This reduces the amount of "rigidity" in the approximate solution.
"transpose trick" was used in an attempt to make relaxation converge for a class of sparse matrices. In this case, however, more than convergence is at stake: If $A$ is not square, it simply does not make sense to apply relaxation! But in multiplying $A$ by $A^{T}$, we have once again made the system of equation even more ill-conditioned. ${ }^{31}$ Furthermore, the resulting Gauss-Seidel iteration matrix again has a spectral radius close to 1 , making convergence extremely slow.

### 3.4.9 Other approaches to FEM on manifolds

How can we avoid the problems associated with overdetermined systems of equations? There are a few alternatives. First, we can use more sophisticated methods of generating constraint equations and choosing interpolation nodes, such as the methods proposed in Chesshire and Henshaw [7] or Petersson [23]. While this will not avoid the necessity of computing the normal equations, it does hold the hope of minimizing the effects of the rigidity problem.

## Improving interpolation methods

For the sake of completeness, let us take a brief look at how well these variations on interpolation methods work. The basic algorithms tested here are:

1. The idea of Chesshire and Henshaw, CMPGRD.
2. Same as make-all-ordered-constraints, except nodes in overlap regions are allowed to become interpolation nodes if and only if they are near the chart's edge.

The second idea above attempts to create an interpolation geometry depicted in Figure 3-24. Contrast this with Figure 3-23, and one sees that this should help make the system of equations less overdetermined while still propagating enough information to arrive at a reasonable solution.

[^33]

Figure 3-25: The results generated using Chesshire and Henshaw's CMPGRD algorithm.


Figure 3-26: The results generated using the idea depicted in Figure 3-24.


Figure 3-27: The idealized case, where charts do not overlap but intersect nicely along a common edge.

Figure 3-25 shows the result of the Chesshire-Henshaw algorithm, while $3-25$ shows the results of using the second idea. The accuracy should have improved slightly. However, relaxation converges sufficiently slowly that the improvement in accuracy, if any at all, is probably lost in the noise.

## A method that works

This section describes a method that actually works fairly well compared to the interpolation methods of earlier sections. It avoids the problem of generating overdetermined systems of equations, and the global matrix of equations it generates is guaranteed to be symmetric positive-definite, and thus solvable by relaxation without having to worry about normal equations and condition numbers. This method involves "pretending" as if the mesh were global, even if it were not, and for this reason it is referred to as the "semi-local method" here, even though by our earlier definition this is a strictly local discretization method.

The basic idea is simple: Suppose that charts, instead of overlapping, fit together like jigsaw puzzle on the manifold along well-defined boundaries (see Figure 3-27). ${ }^{32}$

Suppose now that the $i$ th node lies on the boundary between these two "charts." From $C_{1}$ the node obtains an equation of the form

$$
\begin{equation*}
u_{i}=\sum_{j} a_{i j} u_{j}+b \tag{3.34}
\end{equation*}
$$

where the $u_{j}$ are the unknowns sample values, and the $a_{i j}$ are the finite element coefficients. Similarly, from $C_{2}$ the node obtains:

[^34]\[

$$
\begin{equation*}
u_{i}^{\prime}=\sum_{j} a_{i j}^{\prime} u_{j}^{\prime}+b^{\prime} . \tag{3.35}
\end{equation*}
$$

\]

Now, consider what the constraint approach actually does: In this idealized case, the node in question does not lie inside an element, but rather is also a node of the other chart. Thus, the constraint approach must append the equation

$$
\begin{equation*}
u_{i}^{\prime}-u_{i}=0 . \tag{3.36}
\end{equation*}
$$

This is equivalent to the system of two equations: ${ }^{33}$

$$
\begin{align*}
& u_{i}=\frac{1}{2}\left(\sum_{j} a_{i j} u_{j}+\sum j a_{i j}^{\prime} u_{j}^{\prime}+b+b^{\prime}\right),  \tag{3.37}\\
& u_{i}=\frac{1}{2}\left(\sum_{j} a_{i j} u_{j}-\sum j a_{i j}^{\prime} u_{j}^{\prime}+b-b^{\prime}\right) .
\end{align*}
$$

But consider the finite element integrals in Equation (A.28) of §A.2.3: In order the obtain the correct finite element equation over the whole mesh, the correct equation is the top equation, which is the sum of the two contributions from the charts. In the context of finite elements, the bottom equation makes no sense at all. ${ }^{34}$ Thus, the constraint approach overdetermines the discretized system of equations, and the addition of this extra equation destroys the accuracy of the approximation method in this idealized case.

This is a fairly clear indication that we should add the equations corresponding to the same node in different charts. Furthermore, this generates one equation for each interior node, instead of two as in the interpolation case. And, because of the form of the finite element integrals in Equation (A.28), the matrix is guaranteed to be symmetric positive semi-definite; invertibility then guarantees positive-definiteness.

Now, in general, charts will not cover the manifold this nicely. However, we can always try to make the overlap as small as possible (in terms of nodes shared by charts), and then pretend as if we are in the idealized case and apply the equations above.

More formally, the following is the semi-local algorithm. Note that $\left\{C_{i}\right\}$ is a given list of charts.

1. Construct a set of nodes $N_{i}$ for each chart $C_{i}$.
2. For each node $n$ in $N_{i}$ and for each chart $C_{j}$ with $j>i$, check if $n$ belongs to $C_{j}$. If so, remove $n$ from $N_{i}$. This completely removes the overlap (in terms of sample points) between charts.

[^35]

Figure 3-28: The closer a chart is to the bottom of the "stack," the more likely it will keep its nodes. The lower nodes are then copied to the top charts. Intuitively, think of this as cutting holes from the top charts, and then "pasting" them downwards onto lower charts.
3. For each remaining node $n$ in $N_{i}$ and each chart $C_{j}$ with $j<i$, check if $n$ belongs to $C_{j}$. If so, make a copy of $n$ and add it to $N_{j}$. This restores some overlap. Furthermore, while this cannot guarantee that local meshes agree in intersections of charts, it does guarantee that all charts share all nodes in overlap regions.
4. Triangulate and initialize elements; perform local FEM computation. The previous step may have restored too much overlap, so the meshes may have to be "trimmed."
5. For each node $n^{\prime}$ of chart $C^{\prime}$, if it is a copy of some node $n$ in another chart $C$, then add the equation of $n^{\prime}$ in $C^{\prime}$ to the equation of $n$ in $C$, and remove the variable corresponding to $n^{\prime}$.

Figure 3-28 depicts what the semi-local algorithm does to the overlap between charts. Steps 2 and 3 above are carried out by the following implementation of the auxiliary procedures generate-node-lists:

```
;;; Generate lists of nodes for each chart, and then reduce the overlap:
(define (generate-node-lists make-nodes charts argl)
```

```
    ;; Generate a list of nodes for each chart, then loop over the charts. Note
    ;; that the earlier a chart is in the list, the less likely its nodes are to
    ;; survive.
(let next-chart ((charts charts)
                    (lists (make-nodes-for-each-chart make-nodes charts argl))
                    (result '())
                    (reversed '())
                    (count 0))
    (if (null? charts)
        (copy-overlap-nodes count result reversed)
        (next-chart (cdr charts)
                            (cdr lists)
                            (cons (remove-overlap-nodes (car lists) (cdr charts))
                    result)
                            (cons (car charts) reversed)
                            (+ count 1)))))
(define (make-nodes-for-each-chart make-nodes charts extra-args)
    (map (lambda (chart) (apply make-nodes (cons chart extra-args))) charts))
;;; Take out all nodes in NODES that belong to any of the charts in CHARTS.
(define (remove-overlap-nodes nodes charts)
    (let next-node ((nodes nodes) (result '()))
        (if (null? nodes)
            result
            (let* ((node (car nodes))
                            (p (node:get-point node)))
                (let next-chart ((charts charts))
                        (if (null? charts)
                        (next-node (cdr nodes) (cons node result))
                        (if (chart:member? p (car charts))
                            (next-node (cdr nodes) result)
                            (next-chart (cdr charts)))))))))
;;; For each node list in LISTS, take each node and see if it's in one of the
;; charts that come after the node's own chart in list-order. If so, make a
;;; copy of that node and put it in the corresponding chart. Note that the
;;; order of node lists is reversed.
(define (copy-overlap-nodes count lists charts)
    (let ((v (make-vector count '())))
        (let next-list ((lists lists) (charts charts) (i 0) (result '()))
            (if (null? lists)
                result
                (let next-node ((nodes (car lists)))
                        (if (null? nodes)
                            (next-list (cdr lists) (cdr charts) (+ i 1)
                            (cons (append (vector-ref v i) (car lists)) result))
                            (let ((node (car nodes)))
                                (if (or (node:local-boundary? node)
                            (node:boundary? node))
                            (let ((p (node:get-point node)))
                                    (let next-chart ((charts (cdr charts))
                                    (j (+ i 1))
```

```
(1 (cdr lists)))
(if (null? charts)
(next-node (cdr nodes))
(let ((chart (car charts)))
(if (chart:member? p chart)
(let ((other (close-node p (car 1)))) (if other
(node:set-constraint! other node)
(vector-set! v j
(cons
(node:copy node chart)
(vector-ref \(\vee \mathrm{j})\) )))))
(next-chart (cdr charts) (+ j 1) (cdr l))))))
(next-node (cdr nodes)))))))))
```

```
;;; A kluge to make sure nodes do not become too close to each other:
(define close-node
    (let* ((close-enuf? (make-comparator .01))
                (too-close? (lambda (p q)
                            (close-enuf? (vector:distance p q) 0))))
        (lambda (p l)
            (let loop ((1 1))
                (if (null? 1)
                #f
                    (if (too-close? p (node:get-point (car l)))
                            (car 1)
                            (loop (cdr l))))))))
```

After this stage, the amount of overlap between charts (in terms of how many nodes are shared) should have been reduced. But more importantly, the fact that nodes are shared will help us construct the equations later. ${ }^{35}$ However, the amount of overlapping after this stage may still be too much, so after triangulation it is necessary to "trim" the mesh a bit. This is accomplished through the following implementation of the auxiliary procedure process-complex:

```
;;; After filtering out nodes, local boundary information becomes useless...
(define (exact-overlap complex charts)
    (kill-extra-nodes complex charts)
    (resurrect-only-connected-nodes complex charts)
    (keep-only-live-nodes complex charts))
(define (kill-extra-nodes complex charts)
    ;; Figure out which nodes to keep by looking at the overlaps:
    (write-line '(processing ,(length (complex->vertices complex)) nodes...))
    (let next-node ((nodes (complex->vertices complex)))
```

[^36]```
    (if (not (null? nodes))
    (let ((node (car nodes)))
        (let ((p (node:get-point node)))
            (let next-chart ((charts charts))
                (if (null? charts)
                    (next-node (cdr nodes))
                    (let ((chart (car charts)))
                                    (if (chart:member? p chart)
                                    (let ((node (car nodes)))
                                    (node:kill! node)
                                    (node:set-local-boundary! node #f)
                                    (next-node (cdr nodes)))
                                    (next-chart (cdr charts)))))))))))
(define (resurrect-only-connected-nodes complex charts)
    ;; Only keep nodes that are connected to live ones:
    (write-line '(figuring out overlaps...))
    (let loop ((faces (complex->faces complex)) (keep '()))
        (if (null? faces)
            (for-each
                (lambda (face)
                (for-each
                    (lambda (node)
                        (if (not (node:active? node))
                        (begin
                                    (node:set-local-boundary! node #t)
                                    (node:resurrect! node))))
                face))
                keep)
            (if (at-least-one-live-node? (car faces) charts)
                (loop (cdr faces) (cons (car faces) keep))
                (loop (cdr faces) keep))))
(define (keep-only-live-nodes complex charts)
    ;; Figure out which faces/edges/etc. to keep:
    (write-line '(processing complex...))
    (let loop ((complex complex) (result '()))
        (if (null? complex)
            (reverse result)
            (let inner-loop ((faces (car complex)) (okay-faces '()))
                (if (null? faces)
                        (loop (cdr complex) (cons okay-faces result))
                        (let* ((face (car faces))
                            (list? (list? face)))
                            (if (or (and list? (not (memq #f (map node:active? face))))
                            (and (not list?) (node:active? face)))
                                (inner-loop (cdr faces) (cons face okay-faces))
                                    (inner-loop (cdr faces) okay-faces)))))))
(define (at-least-one-live-node? face charts)
    (memq #t (map node:active? face)))
```

This works much like the earlier routines: It removes all possible overlap, then "grows" the mesh back a little bit. But because this stage occurs after the triangulation, the structure of the mesh can be used to control how much overlap there is. And because the earlier stage ensured that intersecting charts share nodes in overlap regions, this guarantees that this geometric configuration is as close to the ideal situation in Figure 3-27 as possible.

The following definitions then combine the local equations into a global system of equations, and construct the top-level programs:

```
;;; Generate the sparse matrix by adding appropriate equations together:
(define (merge-equations domain equations)
    (let ((nodes (manifold:get-nodes domain))
            (count 0)
            (mat #f))
        ;; First, assign IDs to nodes, and create the matrix:
        (write-line '(creating matrix...))
        (let loop ((nodes nodes) (i 0))
            (if (null? nodes)
                (begin
                (set! count i)
                (set! mat (make-sparse-matrix count (+ count 1))))
            (let ((node (car nodes)))
                (cond ((node:boundary? node)
                            (node:set-id! node 'boundary-node!)
                            (loop (cdr nodes) i))
                            ((node:get-constraint node)
                        (node:set-id! node 'constrained-node!)
                            (loop (cdr nodes) i))
                            (else
                            (node:set-id! node i)
                            (loop (cdr nodes) (+ i 1))))))
        ;; Next, start filling in equations while keeping track of constraints:
        (write-line '(copying equations...))
        (let next-eq ((equations equations))
            (if (null? equations)
                (begin
                        (write-line '(done!))
                        mat)
                        (let* ((eq (car equations))
                            (i (node:get-real-id (equation:get-node eq))))
                                (sparse-matrix-set! mat i count
                            (+ (equation:get-constant eq)
                                    (sparse-matrix-ref mat i count)))
                (let next-term ((terms (equation:get-terms eq)))
                    (if (null? terms)
                        (next-eq (cdr equations))
                        (let* ((term (car terms))
```

| Total number <br> of nodes | Absolute error |  |  | Relative error |  |
| :--- | :--- | :--- | :--- | :--- | :--- |
|  | Maximum | Minimum | Average | Maximum | Minimum |
| 63 | 0.105298 | 0.000442996 | 0.0131643 | 0.547058 | -1.19462 |
| 130 | 0.0745854 | 0.000150227 | 0.00777591 | 2.99789 | -10.7455 |
| 225 | 0.049322 | 0.00010038 | 0.00369715 | 0.991634 | -2.70366 |
| 337 | 0.0532307 | $1.88416 \mathrm{e}-06$ | 0.0067024 | 5.5545 | -1.92022 |
| 485 | 0.0762939 | $1.29948 \mathrm{e}-06$ | 0.00677602 | 22.5097 | -7.5364 |
| 655 | 0.0420157 | $4.13828 \mathrm{e}-06$ | 0.00207558 | 2.01222 | -0.65778 |
| 843 | 0.0232905 | $1.88216 \mathrm{e}-07$ | 0.00137413 | 5.82382 | -1.90627 |
| 1062 | 0.0270354 | $7.00353 \mathrm{e}-07$ | 0.0012742 | 2.89441 | -0.983369 |
| 1297 | 0.0233997 | $4.69356 \mathrm{e}-06$ | 0.00224749 | 3.20791 | -9.62969 |
| 1562 | 0.0187541 | $1.03235 \mathrm{e}-07$ | 0.00139054 | 0.542516 | -1.62893 |
| 1862 | 0.0172077 | $7.19112 \mathrm{e}-07$ | 0.000983776 | 6.20396 | -2.05861 |

Table 3.3: Statistics of the results generated by the "semi-local method."

```
                                    (j (node:get-real-id (term:get-node term)))
                    (val (term:get-coeff term)))
(sparse-matrix-set! mat i j (+ (sparse-matrix-ref mat i j)
                                    val))
(next-term (cdr terms)))))))))
```

;; ; Construct the top-level programs:
(define combine-equations-without-overlap
(pde:equation-maker merge-equations))
(define pde:make-domain-without-overlaps
(pde:domain-maker generate-node-lists exact-overlap))

Like append-constraint-equations, this program mostly performs the tedious task of matrix construction. Overlaps between elements from different charts are a source of error for this method. However, the algorithm very carefully reduces the amount of overlap between charts to the minimum required for the merging process.

Table 3.4.9 shows the results generated by this method, while Figure 3-29 shows an approximate solution generated this way.

What is more interesting is a plot of the relative error in Figure 3-30: By a comparison with Figure $3-16$, one sees that the the areas with the highest relative error are very much correlated with chart boundaries, which is where we would expect the errors to be maximized.

Figure 3-31 shows a plot of the average absolute error versus the number of nodes, which should be convincing evidence that this method, while not extremely accurate, does converge to the true solution at a reasonable rate as the number of nodes is increased.


Figure 3-29: An approximate solution of the boundary value problem generated by the semi-local method.


Figure 3-30: The relative error for the solution plotted in Figure 3-29.


Figure 3-31: The average absolute error versus the number of nodes. The data is generated using the semi-local methods, with the same parameters as earlier experiments.

### 3.5 Some comments on mesh generation

This section contains a few brief comments regarding the difficulty of triangulating manifolds, and hence using global discretization methods, for integrating PDEs on manifolds. In particular, a standard theorem of differential topology states that every manifold can be covered by a mesh of "triangular" elements. ${ }^{36}$ More precisely, Munkres [18] presents a proof that every manifold has the structure of a simplicial complex. The proof is constructive and works by first triangulating each chart locally (in Euclidean space), and then refining the triangulations on overlapped regions between charts so that they can be "pasted together." While this construction is very suggestive from a computational viewpoint, there is a catch: The proof requires the computation of the interesections between a large number of simplicies. While this mostly involves only linear equations, and is in principle computable, in practice this can be extremely expensive in terms of computational resources. Thus, the mathematical proof does not actually supply a solution to the computational problem of triangulating a manifold.

In fact, the merging of local meshes into global ones is the main bottleneck of the entire process. As shown by the quickhull algorithm [6], one can always efficiently triangulate convex subsets of Euclidean spaces. Thus, the only major problem is the merging of local

[^37]meshes into global ones.
One possible solution is to use abstractions other than manifolds to describe spaces with complex geometries. For example, instead of building local coordinate systems that overlap arbitrarily, one could imagine building complex spaces by deforming and "pasting" lines and squares and cubes and other such topological objects. One can indeed build a large class of spaces this way (in theory), and such spaces are called $C W$ complexes. Differentiable manifolds are all examples of CW complexes, so in principle one could use this abstraction to do local triangulation and, because the pieces fit along the boundary exactly (instead of in some hard-to-determine overlap), one could merge the meshes more easily.

One important thing to note is that, in the end, a decision on how spaces are constructed should be driven by actual applications because it is almost impossible to arrive at a general computational framework for any class of numerical problems without a context. For example, even though many computational geometry algorithms are restricted to lowdimensions ( 2 or 3 ), for most structural engineering problems this is sufficient to generate reasonable models. Furthermore, in fluid problems, the spatial dimension is often low, and while the geometry of the domain is a significant part of the difficulty of simulating fluid flows, it is not the only difficulty. The abstract manifold approach developed in this report are probably most suited to solving problems from mathematical physics, where abstract mathematical spaces are perhaps more commonly encountered.

### 3.6 Directions for future work

There are a number of alternatives that may help surmount the difficulties described in earlier sections.

### 3.6.1 Improvements to finite differences

There are a few directions in which finite difference methods may be improved. One is to develop better algorithms for solving large sparse systems of linear equations, so that the unstable coefficients generated by finite difference techniques using irregular sample points would become solvable.

A distinctly different approach would be to simply do finite differences on regular grids, and to basically follow the Chesshire-Henshaw idea. While their idea works well for some special problems, however, there are cases when their idea produces less reasonable answers. For a discussion of this, see [23].

### 3.6.2 Improvements to finite elements

To improve the performance of finite element methods on manifolds, on the other hand, probably requires more work. While FEMs work admirably well with irregular sampling geometry, the complexity of the geometric problem of combining local equations into a global system can be rather daunting, as was shown in this section. Clearly, much more work needs to be done in this domain, and there are many variations on these ideas. Part of the difficulty of this problem is that, in view of the variational formulation of Laplace's equation, the problem of combining local equations is that of a constrained minimization problem, which are often non-trivial. On the other hand, perhaps a standard technique like Lagrange multipliers would work nicely for this case. There are many other things to try.

On the other hand, one of the difficulties that arises with the semi-local method is that it gives charts little control over the geometry of their local meshes because nodes are copied between charts. Thus, while the method produces reasonably good results and has nice convergence properties, it does accumulate quite a bit of truncation error due to geometric defects. It would be very useful to generalize the idea in a way that still allows regular local grids, so as to minimize the effects of geometry on accuracy.

### 3.6.3 Other methods

Finally, there could be breakthroughs in mesh generation on arbitrary $n$-manifolds. Although most current work have focused on low-dimensional problems because of their potential applications in engineering and computer graphics, this is a rather active research area and much is being discovered. A global finite element method should work rather nicely on a manifold.

Or one could exploit the meshless methods developed by Duarte and Oden [11], which explicitly build partitions of unity using discrete sample points without first generating a mesh. This has the advantage that one does not need to think about combining meshes to use these methods on manifolds. Furthermore, their method can utilize essentially RayleighRitz or Galerkin approximations, so that the resulting linear equations are solvable by iterative methods.

## Chapter 4

## Problems with Time

This topic of this chapter is the numerical solution of partial differential equations that describe how certain physical systems evolve in time. Again, as in the solution of elliptic boundary value problems on manifolds, it is possible to break this problem into two components: First, we must have a way of locally integrating the PDE; and second, the local solutions must be combined to form a global solution. It is also possible, of course, to discretize the entire manifold first before solving the equations, but it will turn out that the difficulties one must overcome in global methods are not all that different from those of local methods. Because of the nontrivial nature of solving such equations even in the case where the domain has trivial geometry, this chapter focuses on the local problem.

Standard PDE solvers generally perform finite element or finite difference approximations in space first, so as to compute the time derivative, and then step forward uniformly in time at regular intervals-As one would with ordinary differential equations. ${ }^{1}$ While this approach works well enough for many problems, it is rather unsatisfactory philosophically: We have good reason to believe that physical reality does not distinguish among time-like directions, and that any time axis is just as fundamental and just as arbitrary as any other. Thus, a coordinate-independent description of fundamental physical processes and the equations that govern them should not depend on the existence of a unique time axis. More pragmatically, there exist physical problems for which it is helpful to use different frames of reference, and a properly coordinate-independent formulation of PDEs should not be restricted to advancing along an arbitrarily chosen time axis. The use of regular time steps implicitly gives the time coordinate a special status, which complicates any attempt at coordinate-independent representations and solutions.

[^38]One natural solution to this dilemma is the following: Instead of discretizing the spatial dimensions and stepping forward in time, one simply discretizes the equation over spacetime ${ }^{2}$ and solve for the unknown solution over the entire spacetime region of interest in one step. One might expect, for example, that standard finite element techniques may be applied directly to the entire spacetime domain, and that the unknown solution can be solved over all spacetime events by solving one very large system of algebraic equations.

Perhaps not too surprisingly, this simple idea does not work, even though there are no obvious problems in the derivation. One reason for this failure is proposed in the next section, and, in view of this proposal, various ways for improving the accuracy are suggested in $\S 4.3$. $\S 4.4$ discusses some of the difficulties that arise in these improved methods, and also presents some problems that spacetime methods must, in general, overcome. Finally, possible directions for future research in this area are suggested in §4.5.

This chapter is more about open questions than solutions to well-posed problems, and as such may be seem less coherent than earlier chapters. However, it is hoped that the questions asked here will lead to other questions whose answers will some day shed light on the mathematical, physical, and computational structures involved in understanding partial differential equations. Also, because everything here is performed in subsets of Euclidean space, explicit programs probably do not aid in understanding, and are thus omitted in this chapter.

As in earlier chapters, the focus here will be on the simplest possible example that exhibits interesting behavior, which in this case is the linear wave equation.

### 4.1 The linear wave equation

While Laplace's equation is arguably one of the most important PDEs, there are other important equations that have fundamentally different behavior. One of these is the linear wave equation. This equation describes, for example, the propagation of electromagnetic waves in free space. It is therefore useful to identify one of the variables as time in some frame of reference, and to define $D_{t}=D_{n+1}$ so that time and space derivatives can be more easily distinguished. The wave equation in ( $n+1$ ) dimensions ( $n$ space dimensions plus time) is then:

$$
\begin{equation*}
\left(D_{t}^{2}-c^{2} \Delta\right) u=0 \tag{4.1}
\end{equation*}
$$

where $c>0$ is a real constant and $\Delta=\nabla^{2}=\sum_{i=1}^{n} D_{i}^{2}$ is the Laplacian operator over the space variables. For concreteness, this discussion will be restricted to the case $n=1$. In

[^39]this case, the wave equation also describes the behavior of a vibrating string with small oscillations. For convenience, let us define $D_{x}=D_{1}$ so that $\Delta=D_{x}^{2}$.

In constrast to Laplace's equation, the boundary value problem for the wave equation is ill-posed. That is, it does not always have solutions for arbitrary boundary conditions, and even when such solutions exist, they are often not unique. However, in the case when $n=1$ and $\Omega$ is the unit square $\left\{(x, t) \in R^{2} \mid 0 \leq x \leq 1,0 \leq t \leq 1\right\}$, one can specify initial conditions

$$
\begin{gather*}
u(x, 0)=f(x), \quad D_{t} u(x, 0)=g(x),  \tag{4.2}\\
u(0, t)=h(t), \quad u(1, t)=k(t) \tag{4.3}
\end{gather*}
$$

for some prescribed functions $f, g, h$, and $k$. Then the wave equation does have a unique solution. This is called the initial value problem. ${ }^{3}$

It is tempting to apply the finite element method directly to the initial value problem for the wave equation. In particular, Galerkin's method may seem generally applicable. However, there is good evidence that Galerkin's method, as presented in Appendix A, will almost always do poorly for the linear wave equation. This does not, of course, imply that finite element methods cannot be somehow adapted for the wave equation. First, though, let us take a closer look at why boundary value problems are ill-posed for the linear wave equation.

### 4.2 Initial value problems and characteristics

As stated in in the previous section, boundary value problems are ill-posed for the wave equation. The root of this problem is the existence of "characteristic manifolds," which describe the "propagation" of initial data. In this section, these notions will be examined a little more closely. However, a close analysis of the ill-posedness of the boundary value problem for the wave equation in terms of these concepts can be fairly complicated and involves many technical details. ${ }^{4}$ Thus, this discussion will instead focus on a simpler example, from which we can derive some informal observations on the wave equation.

[^40]
### 4.2.1 Characteristic curves for a first-order equation

There are many equations for which the boundary value problem is ill-posed. Among these are hyperbolic equations, for which initial value problems are well-posed. ${ }^{5}$ This is because of the existence of characteristics, along which one cannot specify arbitrary values of the solution and its derivatives of order less than $m$ (where $m$ is the order of the equation). Equivalently, characteristics propagate data about values of the solution and its lower-order partials, because the interdependence of the solution and its lower-order derivatives leads to equations that determine the evolution of the solution along characteristics.

To illustrate, consider the first-order linear equation

$$
\begin{equation*}
\left(D_{t}+c D_{x}\right) u=0 \tag{4.4}
\end{equation*}
$$

with constant $c>0$. Defining the coordinate transformation $f$ with inverse $g$ by

$$
\begin{align*}
& f_{\xi}(x, t)=x-c t,  \tag{4.5}\\
& g_{\tau}(\xi, \tau)=t  \tag{4.6}\\
& g_{x}(\xi+c \tau, g_{t}(\xi, \tau)=\tau
\end{align*}
$$

we obtain the new equation

$$
\begin{align*}
\left(D_{t}+c D_{x}\right) u= & \left.\left(D_{\tau} v \circ f\right) D_{t} f_{\tau}+\left(D_{\xi} v \circ f\right) D_{t} f_{\xi}\right)+ \\
& c\left[\left(D_{\tau} v \circ f\right) D_{x} f_{\tau}+\left(D_{\xi} v \circ f\right) D_{x} f_{\xi}\right]  \tag{4.7}\\
= & D_{\tau} v \circ f-c D_{\xi} v \circ f+c D_{\xi} v \circ f  \tag{4.8}\\
= & D_{\tau} v \circ f  \tag{4.9}\\
= & 0, \tag{4.10}
\end{align*}
$$

where $v(\xi, \tau)=u\left(g_{x}(\xi, \tau), g_{t}(\xi, \tau)\right)$. Thus, under this coordinate transformation, the equation becomes $D_{\tau} v=0$, so that $v$ is constant in $\tau$ and depends only on $\xi$. Thus, $v(\xi, \tau)=F(\xi)$ for some function $F$. Changing back to the old coordinates, this implies that a solution $u(x, t)$ of Equation (4.4) must take the form

$$
\begin{equation*}
u(x, t)=F(x-c t) \tag{4.11}
\end{equation*}
$$

Equivalently:

$$
\begin{equation*}
u(x, t)=u(x-c t, 0) . \tag{4.12}
\end{equation*}
$$

[^41]Conversely, any differentiable function in the form (4.11) satisfies the original equation. So the solution is completely determined by its values along the line $t=0$. The initial values $u(x, 0)$ are thus "propagated" along the lines $x=c t$, which are called characteristic curves (or simply characteristics). As shown above, one cannot specify arbitrary values at two distinct points along the same characteristic. Thus, the boundary value problem for the first-order linear equation (4.4) is, in general, ill-posed: Every characteristic intersects the boundary of any bounded spacetime region at least twice, ${ }^{6}$ and admissible boundary data are thus severely constrained.

### 4.2.2 Characteristics for general equations

Now consider an $m$ th-order partial differential equation over an ( $n+1$ )-dimensional domain $\Omega$. Let $S$ be an $n$-dimensional subspace of $\Omega$. In general, one can prescribe values for the derivatives of order less than $m$ on $S$, subject to some compatibility conditions-Partial derivatives of orders less than $m$ in directions tangent to $S$ must satisfy the chain rule. ${ }^{7}$ These compatibility conditions, together with the differential equation, usually produce enough equations to determine all derivatives $D^{\alpha} u$ of $u$ with $|\alpha| \leq m$, including the normal derivatives with respect to $S$ up to order $m$. If this is true everywhere on $S$, then $S$ is said to be non-characteristic. If the equations are singular everywhere on $S$, then $S$ is characteristic.

Intuitively, information on a characteristic subspace $S$ does not determine how the solution evolves outside of $S$. Since the coefficients of linear equations ${ }^{8}$ formed by the compatibility conditions and the differential equation consist of combinations of the unknown solution and their lower derivatives, the singularity of such a system of equations on a characteristic manifold implies that the quantities are not independent of each other. These constraints in turn determine derivatives tangential to the characteristic in terms of lowerorder normal derivatives and solution values, so that such data can be propagated along

[^42]

Figure 4-1: Characteristic lines of the wave equation. The interval on the initial line $\{t=0\}$ bounded by characteristics is called the domain of dependence of the solution $u$ at the given point: The value of the solution at the "tip" of the triangular region bounded by the characteristics (called an inverted light cone) can only depend on data in the domain of dependence; nothing outside the interval can affect the solution at that point.
the characteristic via another differential equation.
For $m$ th-order quasilinear partial differential equations $L[u]=b$, one can derive an algebraic criterion for characteristics (only the result is stated here): Let $L=\sum_{|\alpha| \leq m} A_{\alpha} D^{\alpha}$, where the $A_{\alpha}$ are functions of spacetime events, values of the unknown solution, and its derivatives of order strictly less than $m$. Then $S$ is characteristic if and only if for every point $p$ on $S$ and non-zero vector $v$ normal to $S$ at $p$, the equation $\sum_{|\alpha|=m} A_{\alpha} v_{p}^{\alpha}=0$ holds. For example, in the case of the linear wave equation, $v_{t}^{2}-c^{2} v_{x}^{2}=0$ must hold, so if a vector $v=\left(v_{x}, v_{t}\right)$ is normal to characteristics, then it satisfies $v_{t}=c v_{x}$ or $v_{t}=-c v_{x}$. Thus, the characteristics for the linear wave equation are the lines $x=c t$ and $x=-c t$ (see Figure $4-1$ ). Because solution values along characteristics cannot be completely independent, we see that the boundary value problem for the wave equation cannot be well-posed in the strictest sense.

### 4.2.3 Variational principles revisited

We will now examine variational principles more closely, and to develop some tools useful for analyzing the application of finite elements to the linear wave equation. It may be helpful for the reader to review the material in Appendix A first, particularly the derivation of the Rayleigh-Ritz method and its relation to Galerkin's method.

First, we need to derive a necessary condition for a function to minimize an action. Let $L: R^{5} \rightarrow R$ be a differentiable function, which is called the Lagrangian density, and for any
real-valued function $u$ on $\Omega$ let $\gamma_{u}$ be the function defined by

$$
\begin{equation*}
\gamma_{u}(x, y)=\left(u(x, y), D_{1} u(x, y), D_{2} u(x, y), x, y\right) \tag{4.13}
\end{equation*}
$$

Once again, define the action by

$$
\begin{equation*}
S(u)=\int_{\Omega} L \circ \gamma_{u}, \tag{4.14}
\end{equation*}
$$

and note that if $L$ is defined by

$$
\begin{equation*}
L(u, v, w, x, y)=\frac{1}{2}\left(v^{2}+w^{2}\right) \tag{4.15}
\end{equation*}
$$

then the action $S$ above becomes the action defined in Equation (A.21).
At this point, it is important to note that in what follows, it will be necessary to differentiate both the function $L$, which has a 5 -dimensional domain, and $u$, which has a 2-dimensional domain. To avoid confusion, in the rest of this section, differential operators on functions over $R^{5}$ will be written as $\partial_{i}$ instead of $D_{i}$; operators on functions over the 2-dimensional domain $\Omega$ will continue to be denoted by $D_{i}$.

To determine a necessary condition for action-minimizing functions, it is helpful to generalize the idea of directional derivatives. Let $h$ be any real-valued function that vanishes on the boundary $\partial \Omega$ of $\Omega$. Consider the real-valued function of a real variable,

$$
\begin{equation*}
V_{h}(s)=S(u+s h)=\int_{\Omega} L \circ \gamma_{u+s h} . \tag{4.16}
\end{equation*}
$$

As in the case when $u$ and $h$ belong to a finite-dimensional vector space, $V_{h}(s)$ computes the function $S$ along the one-dimensional subspace spanned by $h$. Hence, $D V_{h}(0)$ is the directional derivative of $S$ in the direction of $h$ at $u$. If $u$ is indeed a minimum of $S$, then it follows that $D V_{h}(0)=0$ for all "directions" $h$. Differentiating $V_{h}$ under the integral sign yields

$$
\begin{equation*}
D V_{h}(0)=\int_{\Omega}\left(\partial_{1} L \circ \gamma_{u}\right) \cdot h+\left(\partial_{2} L \circ \gamma_{u}\right) \cdot\left(D_{1} h\right)+\left(\partial_{3} L \circ \gamma_{u}\right) \cdot\left(D_{2} h\right)=0 \tag{4.17}
\end{equation*}
$$

Integrating by parts and noting that $h$ vanishes on the boundary $\partial \Omega$ gives us:

$$
\begin{equation*}
\int_{\Omega}\left(\partial_{1} L \circ \gamma_{u}-D_{1}\left(\partial_{2} L \circ \gamma_{u}\right)-D_{2}\left(\partial_{3} L \circ \gamma_{u}\right)\right) \cdot h=0 . \tag{4.18}
\end{equation*}
$$

This equation holds for all functions $h$ that vanish on the boundary of $\partial \Omega$, so the following equation must hold:

$$
\begin{equation*}
\partial_{1} L \circ \gamma_{u}=D_{1}\left(\partial_{2} L \circ \gamma_{u}\right)+D_{2}\left(\partial_{3} L \circ \gamma_{u}\right) . \tag{4.19}
\end{equation*}
$$

With $L$ define as in Equation (4.15), this gives us Laplace's equation. ${ }^{9}$
Note that even though the equivalence of the variational principle with Equation (4.19) has historically been called the principle of least action, the derivation above really finds the stationary points of the action functional. Thus, it is more appropriate to call it the principle of stationary action, though in the case of Laplace's equation it really is a minimum action principle.

### 4.2.4 Galerkin's method and the initial value problem

Let us now return to the question of applying Galerkin's method to the linear wave equation. The main problem is that the wave equation arises from a variational principle, and that Galerkin's method is equivalent to the Rayleigh-Ritz method. This would not be a problem if one is interested in solving boundary value problems, for then the stationary points of the action functional are solutions of the wave equation. But the boundary value problem for the wave equation is ill-posed, as indicated in $\S 4.2$, and in most applications initial value problems are more important. The difference between initial and boundary value problems is that data are specified at different parts of the domain, and in the initial value problem not all of the boundary of the domain has specified values. This geometric difference is where finite element methods break down.

Specifically, let $L$ be defined by

$$
\begin{equation*}
L(u, v, w, x, t)=\frac{1}{2}\left(w^{2}-c^{2} v^{2}\right) \tag{4.20}
\end{equation*}
$$

Using Equation (4.19), this generates the wave equation (4.1). But recall now that in the derivation of Equation (4.19), one of the crucial steps is integrating by parts and using the fact that the perturbation $h$ vanishes on the boundary to get rid of boundary terms. But such perturbations were natural because we were solving boundary value problems. However, if one is interested in the initial value problem, then the appropriate class of perturbations $h$ should vanish on the set $\{x=0\} \cup\{x=1\} \cup\{t=0\}$, and furthermore $D_{t} h$ should vanish on the initial line $\{t=0\}$. The function $h$ can now be nonzero along the subset $\{t=1\}$ of the boundary, and hence integrating by parts would not yield Equation

[^43](4.18). Instead, it gives
\[

$$
\begin{equation*}
\int_{\Omega}\left(\partial_{1} L \circ \gamma_{u}-D_{x}\left(\partial_{2} L \circ \gamma_{u}\right)-D_{t}\left(\partial_{3} L \circ \gamma_{u}\right)\right) h+\int_{t=1}\left(\partial_{3} L \circ \gamma_{u}\right) h=0 \tag{4.21}
\end{equation*}
$$

\]

Supposing $u$ is continuous and has continuous first derivatives, the boundary term in Equation (4.21) vanishes for all $h$ only if $D_{3} L \circ \gamma_{u}=0$ for $t=1$, which in the case of the wave equation means $D_{t} u(x, 1)=0$ for $0 \leq x \leq 1$. This cannot in general be true. Therefore, the boundary term is almost always nonzero, which implies that the integrand in the first term is also nonzero, and thus $u$ cannot satisfy the wave equation. ${ }^{10}$

One can easily show that Galerkin's method for the wave equation is equivalent to finding the stationary points of the approximate action, using an argument almost identical to that of $\S$ A.2.4. Thus, in the limit as the finite element approximation becomes more exact, the approximation constructed by Galerkin's method would converge to some stationary point satisfying the initial conditions (if it converges at all). As shown above, this function cannot satisfy the wave equation. In fact, one can derive lower bounds on the error using the variational principle.

We can also strengthen the argument to show that if such a action-minimizing function exists in the case of the initial value problem and has continuous first derivatives, then for every point $p=(x, 1)$ of the line $\{t=1\}$ such that $D_{t} u(p) \neq 0$, the residual $D_{t}^{2} u-c^{2} D_{x}^{2} u$ is unbounded in every neighborhood of $p$. Thus, $u$ cannot even have continuous second derivatives, and any solution that minimizes the action must contain singularities.

### 4.3 Variations on a theme of Lagrange

In view of the analysis above, there are a few natural variations on the Rayleigh-Ritz idea that may help produce reasonable solutions to the wave equation. In particular, it is possible to eliminate the boundary term from Equation (4.21), so that stationary points of the action functional are indeed solutions of the wave equation. There are a few ways of accomplishing this, and this section proposes two of them. ${ }^{11}$

### 4.3.1 Modifying the action principle

The first idea is to simply modify the Lagrangian density to change the form of Equation (4.21), so that the boundary integral

$$
\begin{equation*}
\int_{t=1}\left(\partial_{3} L \circ \gamma_{u}\right) h=0 \tag{4.22}
\end{equation*}
$$

[^44]

Figure 4-2: A typical cut-off function.
vanishes. This allows the rest of the derivation of Equation (4.19) to be carried through, so that the stationary points of the action do exist and correspond to solutions of the wave equation (or so one would hope).

More specifically, consider the following Lagrangian:

$$
\begin{equation*}
L(u, v, w, x, t)=\frac{1}{2}\left(c^{2} v^{2}-\rho(t) w^{2}\right) \tag{4.23}
\end{equation*}
$$

where $\rho$ is a cut-off function, as depicated in Figure 4-2. Cut-off functions provide a nice way to change the behavior of the differential equation in different regions of spacetime. In this particular case, we wish to choose constants $t_{1}$ and $t_{2}$ such that $\rho(t)=1$ for all $t \leq t_{1}$ and $\rho(t)=0$ for all $t \geq t_{2}$. For our purposes, set $t_{1}=\frac{1}{2}$ and $t_{2}=1$. $\rho$ then vanishes on the final line $\{t=1\}$.

We can apply Equation (4.19) to the Lagrangian density above, obtaining:

$$
\begin{equation*}
\rho(t) D_{t}^{2} u(x, t)+D \rho(t) \cdot D_{t} u(x, t)-c^{2} D_{x}^{2} u(x, t)=0 . \tag{4.24}
\end{equation*}
$$

Thus, for $t<\frac{1}{2}$, the equation is just the linear wave equation. For $\frac{1}{2}<t<1$, the equation slowly changes until at $t=1$, it becomes:

$$
\begin{equation*}
D_{x}^{2} u(x, t)=0, \tag{4.25}
\end{equation*}
$$

which is obviously no longer well-posed because it says nothing about the behavior of $u$ over time. Time ceases to have any meaning in this modified system after $t=t_{2}=1$.

The boundary integral term that we wanted to eliminate becomes:

$$
\begin{equation*}
\int_{t=1}\left(\partial_{3} L \circ \gamma_{u}\right) h=\int_{t=1} \rho(t) D_{t} u(x, t) d x=0 \tag{4.26}
\end{equation*}
$$

because $\rho$ was chosen to vanish on the line $\{t=1\}$.
Note the characteristics are no longer straight lines, and hence the speed of the wave is


Figure 4-3: The characteristics of this modified wave equation. The top boundary is where "time ends."
also no longer constant (see Figure 4-3). The top boundary, $\{t=1\}$, is where the meaning of time breaks down. The speed of propagation at time $t$ :

$$
\begin{equation*}
\frac{c}{\sqrt{\rho(t)}} \tag{4.27}
\end{equation*}
$$

Thus, the speed of the wave approaches infinity as time approaches $t_{2}$.

## Causality

Note that there is something suspicious about this method. After all, we are hoping to obtain, via this trick, accurate solutions of the wave equation in the spacetime region $\{t<1 / 2\}$ by modifying the equation in the region $\{t>1 / 2\}$. How can changes in the future affect the accuracy of solution in the past? Has some notion of causality been violated? Indeed, even though this trick does not provide accurate numerical solutions, it does generate symmetric systems of linear algebraic equations, which implies that unknown data from the future does somehow affect the past.

The "solution" to this apparent paradox is that the finite element method really has no built-in directionality. Thus, the Rayleigh-Ritz equations do not enforce any causal structure in spacetime, but instead only gives correlations between sample points. In a very informal sense, this can actually be advantageous: By correlating predications made from past data with constraints imposed in the future, one might even hope to improve the solution over the entire spacetime region of interest.

## The Lorentz metric

The usual Lagrangian for the wave operator can be expressed in terms of the Lorentz metric: ${ }^{12}$

$$
\begin{align*}
L\left(u(x, t), D_{x} u(x, t), D_{t} u(x, t), x, t\right) & =\frac{1}{2}\left[c^{2}\left(D_{x} u(x, t)\right)^{2}-\left(D_{t} u(x, t)\right)^{2}\right]  \tag{4.28}\\
& =\frac{1}{2} g^{*}\left(d u_{(x, t)}, d u_{(x, t)}\right) \tag{4.29}
\end{align*}
$$

where $g$ is the metric tensor, $g^{*}$ its dual metric on the dual space, and $d u_{(x, t)}$ denotes, as in Chapter 2, the differential of $u$ at $(x, t)$.

This has several consequences. First, it gives us a coordinate-independent way of describing the wave equation on arbitrary manifolds equipped with a Lorentz metric: Because metric tensors and differentials are already coordinate-free objects on manifolds, Equation (4.28) gives a coordinate-free way of describing the Lagrangian. Now, the variational principle itself can also be stated in a coordinate-free way, since integration of scalar functions can also be defined with respect to a Lorentz metric, as was done for Riemannian metrics in §3.4.1. ${ }^{13}$ So using this Lagrangian and Equation (4.19) gives us a consistent way of generalizing the wave equation to Lorentz manifolds. ${ }^{14}$ In the usual case of Euclidean spacetime with the flat metric, this gives us the usual wave equation.

Furthermore, this description also tells us what we are really doing when we put the time-dependent factor $\rho$ into the Lagrangian density: The metric itself is being made timedependent! Thus, spacetime is no longer flat, and Equation (4.27) shows that the "speed of light" becomes infinite in a finite amount of time in this coordinate system (see Figure 4-3. This may seem problematic from a physical point of view, and it is. It introduces curvature into spacetime and may even violate some conservation laws due to the coarseness of the discretization. The numerical results of the next section show that this method does not work very well.

[^45]| Number <br> of nodes | Absolute error |  |  | Relative error |  |
| :--- | :--- | :--- | :--- | :--- | :--- |
|  | Maximum | Minimum | Average | Maximum | Minimum |
| 14 | 3.46339 | 0.00629682 | 1.36863 | $5.65633 \mathrm{e}+16$ | $-4.27969 \mathrm{e}+15$ |
| 27 | 17.2273 | 0.577482 | 7.74683 | 55.7488 | -45.7939 |
| 44 | 7.10821 | 0.365194 | 3.23993 | $1.98809 \mathrm{e}+15$ | $-9.46197 \mathrm{e}+16$ |
| 65 | 4.34967 | 0.211686 | 2.05495 | 11.2617 | -14.2968 |
| 90 | 2.5196 | 0.0978345 | 1.06008 | $1.5842 \mathrm{e}+15$ | $-3.17402 \mathrm{e}+16$ |
| 119 | 2.05826 | 0.034336 | 0.820245 | 9.95072 | -10.4461 |
| 152 | 3.05573 | 0.0313615 | 1.13394 | $1.45998 \mathrm{e}+16$ | $-2.40796 \mathrm{e}+16$ |
| 189 | 8.72154 | 0.157251 | 3.54988 | 31.5282 | -44.0292 |
| 230 | 3.31555 | 0.0326151 | 1.20425 | $4.68941 \mathrm{e}+15$ | $-2.98963 \mathrm{e}+16$ |
| 275 | 2.48151 | 0.0037325 | 0.881522 | 11.2937 | -11.9207 |
| 324 | 2.22377 | 0.000174037 | 0.753538 | $5.73477 \mathrm{e}+14$ | $-2.11547 \mathrm{e}+16$ |
| 377 | 2.01804 | 0.00592493 | 0.675755 | 8.69369 | -12.2134 |
| 434 | 1.99763 | 0.000252334 | 0.634148 | $3.13461 \mathrm{e}+14$ | $-1.5434 \mathrm{e}+16$ |

Table 4.1: Statics of the results generated by modifying the wave equation.

## Numerical reults

In order to perform actual numerical experiments, it is necessary to choose a specific cut-off function. The actual $\rho$ used is:

$$
\begin{equation*}
\rho(t)=\rho_{0}\left(\frac{t-t_{1}}{t_{2}-t_{1}}\right), \tag{4.30}
\end{equation*}
$$

where $\rho_{0}$ is defined by:

$$
\rho_{0}(t)=\left\{\begin{array}{cc}
1, & t<t_{1}  \tag{4.31}\\
2 t^{3}-3 t^{2}+1, & t_{1} \leq t \leq t_{2} \\
0, & t_{2}<t
\end{array}\right.
$$

The function $\rho$ has the following properties (see Figure 4-2):

$$
\begin{array}{ll}
\rho\left(t_{1}\right)=1, & \rho\left(t_{2}\right)=0 \\
D \rho\left(t_{1}\right)=0, & D \rho\left(t_{2}\right)=0 \tag{4.32}
\end{array}
$$

so that it provides a fairly smooth transition between the linear wave equation (in the range $t<t_{1}$ ) to the degenerate equation (4.25) (in the range $t \geq t_{2}$ ).

Table 4.3.1 shows the data from numerical experiments performed using this method. It is unclear why the relative error jumps between entries, but it may have to do with accidental geometric configurations (i.e. the placement of nodes in the charts and how they overlap), since these jumps also exist in Table 4.3.2. The statistics are only collected over


Figure 4-4: Average absolute error versus number of nodes.
those nodes for which $t<1 / 2$, i.e. in the region where the modified equation agrees with the wave equation. The true solution, in this case, is:

$$
\begin{equation*}
u(x, t)=\cos (2 \pi(x-t)) \tag{4.33}
\end{equation*}
$$

where $c$ was set to 1 for convenience. The discretized equations are solved directly using LU decomposition with partial pivoting.

Figure 4-4 plots some of the results of Table 4.3.1. Clearly, this method does not work very well, although it does appear to slowly converge to the true solution.

Figure 4-5 plots the true solution of the wave equation over this square domain, while Figure 4-6 plots the solution generated by this method. As one can see, this method produces solutions that are only vaguely similar to the true solution in a qualitative sense. Figure 4-7 shows the absolute error distribution, which is sufficiently structured to lead one to suspect the existence of deeper causes of error and possible ways of improving the performance of this method. However, what those causes should be is not entirely clear. ${ }^{15}$

A discussion of possible reasons for the poor performance of this method is postponed until §4.4.1. First, let us take another look at a different approach to eliminating the troublesome boundary term in Equation (4.21).

[^46]

Figure 4-5: The "true" solution to the wave equation given in Equation (4.33).


Figure 4-6: The approximate solution generated by this method.


Figure 4-7: The absolute error. Note that this error is very structured, and hence hints at a deeper cause.

### 4.3.2 Modifying the domain

The second idea depends on modifying the geometry of the domain so that the "final line" $\{t=1\}$ does not exist at all (see Figure 4-8). More specifically, we extend and modify the geometry of the domain by "attaching" a triangle to the original spacetime domain. It is important to ensure that the triangular part of the domain has sides whose slopes are greate than $1 / c$; this makes sure that the boundaries remain timelike, so that boundary conditions can be imposed without making the problem ill-posed. With respect to Equation (4.21), this means the boundary term would no longer exist because boundary data would


Figure 4-8: Attempting to eliminate the boundary term in equation (4.21) by changing the shape of the domain.
be imposed over the entire boundary.

## Geometry and metrics

At first glance, this method and that of the previous section may seem very different: One modifies the wave equation but does not modify spacetime itself, while the other changes the shape of the domain without modifying the equation. However, the two are really more similar than they seem.

Let $X_{1}$ be the rectangular spacetime of Figure 4-3, and let $X_{2}$ denote the "house"-shaped spacetime of Figure 4-8. Consider the comments of $\S 4.3 .1$ : The wave equation really arises from the metric of spacetime, and the method of the previous section works by introducing curvature into spacetime. On the other hand, the "geometric method" of this section seems to have deformed the space without modifying the metric. But $X_{1}$ and $X_{2}$ are topologically equivalent-i.e. One can be continuously mapped onto the other bijectively. Thus, we can always map the oddly-shaped $X_{2}$ onto $X_{1}$ via a continuous transformation $\phi$.

Now, this mapping has an inverse $\phi^{-1}: X_{1} \rightarrow X_{2}$ that is also smooth. By using its differential ${ }^{16}$, we can "pull back" the flat metric from $X_{2}$ onto the space $X_{1}$ :

$$
\begin{equation*}
g_{p}^{1}(v, w)=g_{q}^{2}\left(d \phi_{q}^{-1}(v), d \phi_{q}^{-1}(w)\right) \tag{4.34}
\end{equation*}
$$

where $g^{i}$ is the metric of $X_{i}, p \in X_{1}$, and $q=\phi^{-1}(p) \in X_{2}$. The "pulled-back" metric $g^{1}$ then induces a dual metric $\left(g^{1}\right)^{*}$, which can be used to produce the modified wave equation on $X_{1}$ that is equivalent to the "flat" wave equation on $X_{2}$, in the sense that:

$$
\begin{equation*}
u_{1}(\phi(x, t))=u_{2}(x, t) \tag{4.35}
\end{equation*}
$$

where $u_{i}$ is the solution of the wave equation associated with the Lorentz metric $g^{i}$ on the space $X_{i}$.

While $X_{2}$ has a flat metric $g^{2}$, the metric $g^{1}$ induced by $\phi$ on $X_{1}$ is in general not flat, because the transformation $\phi$ is generally nonlinear. Thus, we see that this new method really can be thought of as just another way to modify the metric of spacetime. The modification, of course, differs from that of the previous section, and generates much more complicated characteristic curves.

One important thing to note is that one can only go so far in modifying the geometry of a space by changing its metric-The topology of the manifold will always stay invariant if the metric is smooth everywhere, even though the geometry changes. In order to generalize

[^47]| Number <br> of nodes | Absolute error |  |  | Relative error |  |
| :--- | :--- | :--- | :--- | :--- | :--- |
|  | Maximum | Minimum | Average | Maximum | Minimum |
| 6 | 14.2485 | 1.74795 | 9.29633 | $1.35509 \mathrm{e}+17$ | -14.2485 |
| 20 | 3.89164 | 0.138369 | 1.95244 | 7.78328 | -4.68721 |
| 42 | 0.627265 | 0.00518237 | 0.247108 | $1.02444 \mathrm{e}+16$ | $-6.52581 \mathrm{e}+15$ |
| 72 | 3.24599 | 0.00177442 | 1.1747 | 10.5042 | -5.74983 |
| 110 | 0.289144 | 0.00082902 | 0.0979768 | $2.97096 \mathrm{e}+15$ | $-7.02175 \mathrm{e}+14$ |
| 156 | 0.318759 | $6.53735 \mathrm{e}-05$ | 0.12427 | 1.18395 | -1.43249 |
| 210 | 0.194245 | 0.000291551 | 0.084536 | $3.06143 \mathrm{e}+15$ | $-4.19103 \mathrm{e}+14$ |
| 272 | 0.0736915 | 0.000189996 | 0.0240319 | 0.286878 | -0.300047 |
| 342 | 0.0496121 | $3.03278 \mathrm{e}-05$ | 0.0153055 | $2.52581 \mathrm{e}+14$ | $-1.80042 \mathrm{e}+14$ |
| 420 | 0.149277 | $1.30728 \mathrm{e}-05$ | 0.0369261 | 1.04892 | -0.677157 |
| 506 | 0.0927114 | 0.000108788 | 0.0293263 | $5.23132 \mathrm{e}+14$ | $-1.48843 \mathrm{e}+15$ |
| 600 | 0.812075 | 0.000495892 | 0.320429 | 3.90818 | -6.36069 |
| 702 | 0.0492277 | $5.3695 \mathrm{e}-07$ | 0.0108849 | $1.4358 \mathrm{e}+14$ | $-2.39808 \mathrm{e}+13$ |

Table 4.2: Statics of the results generated by modifying the spacetime domain.
this particular idea of deforming the spacetime domain to equations on more complicated manifolds, it may be necessary to apply topological transformations as well, so that this method would no longer be simply a variant of the algorithm presented in the previous section.

## Numerical results

Table 4.3.2 shows the data collected using this method. The first few entries were obtained using LU decomposition, but such direct methods fail for larger systems of equations, so relaxation had to be used. Since the wave operator does not produce symmetric positivedefinite matrices (as does the Laplacian), it is necessary to generate the normal equations by multiplying the matrix with its own transpose. Thus, the accuracy of the solution obtained by relaxation is rather limited (see $\S 3.3 .2$ ). However, despite these difficulties, this method clearly outperforms our previous attempt.

Figure 4-9 plots the average absolute error against the number of nodes. As one can see, this method works much better, although it still leaves much room for improvement.

Figure 4-10 shows the approximate solution generated this way, and Figure 4-11 shows the absolute error between this solution and the solution shown in Figure 4-5. Note that the solution in Figure 4-10 is at least qualitatively reminiscent of Figure 4-5.

For this particular method, there is one more parameter we can control: The only constraint on the slope of the triangular "extension" to our domain is that its sides have slope greater than $1 / c$. Thus, the slope of the sides can be varied, which affects the accuracy


Figure 4-9: Average absolute error versus number of nodes. Results are generated by modifying the domain of solution.


Figure 4-10: Approximate solution generated by extending and changing the shape of the domain.


Figure 4-11: The absolute error between Figure 4-5 and 4-10.

| Boundary <br> slope | Absolute error |  |  | Relative error |  |
| :--- | :--- | :--- | :--- | :--- | :--- |
|  | Maximum | Minimum | Average | Maximum | Minimum |
| 2.0 | 0.0492277 | $5.3695 \mathrm{e}-07$ | 0.0108849 | $1.4358 \mathrm{e}+14$ | $-2.39808 \mathrm{e}+13$ |
| 2.1 | 0.021287 | $8.38556 \mathrm{e}-06$ | 0.00718122 | $1.62293 \mathrm{e}+14$ | $-1.168 \mathrm{e}+14$ |
| 2.2 | 0.104445 | 0.000264129 | 0.0333464 | $8.29833 \mathrm{e}+14$ | $-7.40989 \mathrm{e}+14$ |
| 2.3 | 0.204355 | 0.000355765 | 0.0454508 | $1.09504 \mathrm{e}+15$ | $-8.71415 \mathrm{e}+14$ |
| 2.4 | 0.0987477 | $6.25076 \mathrm{e}-05$ | 0.0366781 | $1.18056 \mathrm{e}+15$ | $-7.67629 \mathrm{e}+13$ |
| 2.5 | 0.0390811 | $3.05883 \mathrm{e}-06$ | 0.00954907 | $2.82624 \mathrm{e}+14$ | $-2.05056 \mathrm{e}+14$ |
| 2.6 | 0.0415744 | $1.61621 \mathrm{e}-05$ | 0.0103743 | $4.5445 \mathrm{e}+13$ | $-2.62967 \mathrm{e}+14$ |
| 2.7 | 0.563769 | 0.000158676 | 0.22424 | $5.51249 \mathrm{e}+15$ | $-3.11361 \mathrm{e}+14$ |
| 2.8 | 0.738567 | $1.63661 \mathrm{e}-05$ | 0.252891 | $7.57622 \mathrm{e}+15$ | $-1.83677 \mathrm{e}+15$ |
| 2.9 | 1.12121 | 0.000821135 | 0.309993 | $7.87134 \mathrm{e}+15$ | $-6.65048 \mathrm{e}+15$ |
| 3.0 | 0.858581 | $6.42513 \mathrm{e}-05$ | 0.270326 | $6.98004 \mathrm{e}+15$ | $-3.78587 \mathrm{e}+15$ |

Table 4.3: Statistics obtained by varying the size of the triangular region added.


Figure 4-12: The relative error between Figure 4-5 and 4-10.
of solution. Table 4.3 .2 shows this data.
Figure 4-12 plots this data. As one can see, there is no clear indication of how one choose the slope to minimize the error.

### 4.4 Difficulties with the spacetime approach

This section offers some tentative explanations for the failure of the ideas from the previous section. Furthermore, we will discuss some issues faced by spacetime methods in general.

### 4.4.1 Why the variations failed

It turns out that both of the methods described above probably fail for the same reason: Numerical solvers for the wave equation (and all hyperbolic equations) seem to depend rather sensitively on the geometry of characteristics. In particular, it is often necessary to ensure that information is "propagated" in characteristic direction. Very informally, in terms of finite elements, this means that every node is connected to at least one neighbor in a characteristic direction, so that at least some information is propagated along characteristic lines.

Now, while this condition holds true for both methods over some regions of spacetime, it fails for both methods after some time $t_{c r i t}$ : For the first method, $t_{c r i t}=t_{1}=1 / 2$ because the slope of characteristic curves change after that time but the mesh stays the same. For
the second method, $t_{c r i t}=1$ because after that, the mesh changes to match the shape of the triangular region. Note that this indicates that we should try to choose the slope of the extended triangular region in the second method to be as close to $1 / c$ as possible, and also offers a hint of why the second method performs better than the first.

One might wonder how changes in characteristics or mesh geometry after $t_{c r i t}$ affects the accuracy of the solution before $t_{\text {crit }}$. The answer is that the comments on causality in §4.3.1 apply to both methods: Because the finite element method has no built-in notion of time and provides only correlations between past and future events, errors arising from inconsistencies between the mesh and characteristics after $t_{\text {crit }}$ naturally affect the accuracy of solution before $t_{c r i t}$.

### 4.4.2 Other problems

Aside from that of accuracy, there are other problems associated with applying spacetime methods to hyperbolic PDEs. One of the most serious is the computational resources required: While standard finite difference methods (or finite element methods with regular time steps) need only keep in memory the data associated with the current time step, plus or minus a few neighboring steps, spacetime methods-by their very nature-require all of the data over the spacetime domain. This can be costly in terms of storage requirements if the domain is large. For example, if one needs to understand both the short-term and long-term behavior of solutions, the spacetime region is likely to require a large number of sample points to represent.

Yet another issue is the solution of the discretized equations. Unlike Laplace's equation (or elliptic equations in general), hyperbolic equations almost never generate systems of linear equations that are solvable by relaxation directly. It is for this reason that we were forced to compute the normal equations before applying relaxation to produce Table 4.3.2. While direct methods work fairly well, they are limited by the size of the system one can solve, and in view of the comments above, one can see that spacetime methods can easily generate very large systems of equations.

One last issue is the solution of "true" initial value problems: As stated before, the particular version of the wave equation considered here is a mixed initial-boundary value problem because space, in our case, has finite extent, and both initial values in time and boundary values in space are given. In simulating the propagation of electromagnetic waves in free space, it would be necessary to understand how to simulate large space domains, since finite elements can only work for compact domains. ${ }^{17}$

[^48]
### 4.5 Directions for future work

Aside from the difficulties mentioned in the previous section, there are other issues of interest here. For one thing, the derivation of Equation (4.19) from the variational principle makes no reference to the initial data $D_{t} u(x, 0)=g(x)$, only the boundary data. Furthermore, the existence and importance of characteristics never arises, even though the variational principle is an equivalent way of formulating the wave equation. One natural question, then, is this: Is there a way to analyze the Lagrangian density itself, perhaps as a by-product of the tools used to derive Equation (4.19), that clarifies the importance of characteristics? And why does the initial time derivative not matter in the derivation? What is different between variational principles for PDEs and ODEs, such that the Euler-Lagrange equations hold for ODEs, even though it is initial value problems that are of interest in classical mechanics?

Yet another interesting direction, though only tangentially related to this topic, is that of information propagation. This idea has been mentioned informally throughout this chapter; it would be very interesting to formalize it. In particular, can we compute how much information is "propagated along characteristics"? Is there a way to understand the wellposedness of initial value problems for the wave equation, as well as the ill-posedness of boundary value problems, in terms of information propagation? What connections, if any, exist between information propagation and the Lagrangian density? Finally, how is information propagation in the PDE itself related to information propagation in the PDE solver, and can we use such ideas to estimate numerical accuracy?

It is the author's hope to follow up on some of these questions, and that they may lead to a deeper understanding of hyperbolic equations in general (both linear and nonlinear), and the wave equation in particular.


## Appendix A

## Background Material on Partial Differential Equations

Two general classes of numerical methods for solving partial differential equations are finite difference methods and finite element methods. While other methods, such as spectral decomposition methods, are very effective in special situations, they do not have the general applicability of finite differences and finite elements.

Finite difference methods are very simple. They depend upon the approximation of derivatives by difference quotients. For example, we know from differential calculus that the forward difference

$$
\begin{equation*}
\frac{f(x+h)-f(x)}{h} \tag{A.1}
\end{equation*}
$$

approximates the derivative of $f$ at $x$ for sufficiently small $h$. Finite difference methods are very popular because they are easy to understand and program, and generally run very efficiently on most computers. However, they often depend sensitively upon the particular way in which the domain is discretized, and can easily become numerically unstable. As a result, the literature is full of long, excruciating analyses of convergence criteria and error estimates. The reader will not be subjected to such tortures here.

Instead, this appendix treats finite elements in more depth. This will bring out several important ideas in the theory of partial differential equations along the way.

## A. 1 Matrix inversion

Before all else, one should know that the numerical solution of partial differential equations generally involves the solution of large systems of linear algebraic equations. Thus, it is useful to first examine some of the more popular methods for solving such systems of equations, and to keep these methods in mind throughout the rest of this appendix and the report itself. The reader is assumed to have some knowledge of elementary linear algebra, including familiarity with direct methods such as Gauss-Jordan elimination and LU decomposition (which terminate after a finite number of operations). This section describes some basic iterative methods.

## A.1.1 Iterative methods and relaxation

The basic problem is this: We wish to solve a system of linear equations:

$$
\begin{equation*}
A x=b \tag{A.2}
\end{equation*}
$$

where $A$ is an $n \times n$ matrix and $x, b$ are $n$-vectors, and where $n$ is a large positive integer. For such problems, direct methods such as Gaussian elimination or LU-decomposition require too much space and time to be useful.

One way of computing the solution $x$ is by noting that $x$ is the fixed point of the system of finite-difference equations:

$$
\begin{equation*}
x_{k+1}=(I-A) x_{k}+b, k=0,1,2, \ldots \tag{A.3}
\end{equation*}
$$

Iterating the equation above generates a sequence of vectors $\left\{x_{k}, k=0,1,2, \ldots\right\}$. If the sequence converges, then one would obtain a solution to the original linear system of equations (A.2). Letting $B=I-A$ in the above equation, it follows by induction that:

$$
\begin{equation*}
x_{k}=B^{k} x_{0}+\sum_{i=0}^{k-1} B^{i} b, \tag{A.4}
\end{equation*}
$$

where by convention $\sum_{i=0}^{-1} B^{i}=0 . B$ is called the iteration matrix, and the sequence $\left\{x_{k}\right\}$ converges to the solution $x$ for all initial conditions $x_{0}$ if and only if $\lim _{k \rightarrow \infty} B^{k}=0$ and the infinite series $\sum_{i=0}^{\infty} B^{i}$ converges. One could then show that this holds if and only if the spectral radius $\rho(B)$ is less than $1 .{ }^{1}$

## A.1.2 Jacobi iteration

For general $A$, the iteration matrix $B=I-A$ often has large eigenvalues, so the iteration would not converge. However, there are some modifications that do produce convergent iterations in many instances. These iterative methods, where a difference equation $B$ is obtained from the matrix $A$ and then iterated, are called relaxation methods. In the following, let $L$ denote the off-diagonal lower-triangular entries of $A$, let $D$ denote the diagonal entries of $A$, and let $U$ denote the off-diagonal upper-triangular entries of $A$, so that $A=L+D+U$.

The simplest among these methods, called Jacobi iteration, simply normalizes each row of the matrix by the diagonal entries, so that instead of $B=I-A$, one has:

$$
\begin{equation*}
B=I-D^{-1} A=-D^{-1}(L+U) \tag{A.5}
\end{equation*}
$$

In components, this is equivalent to:

$$
\begin{equation*}
x_{k+1}(i)=\frac{-\sum_{j=1, j \neq i}^{n} a_{i j} x_{k}(j)+\sum_{j=1}^{n} a_{i j} b_{j}}{a_{i i}} . \tag{A.6}
\end{equation*}
$$

Thus, one could perform the iterations rather efficiently if the matrix is sparse; i.e. has a large number of zeros. This method, of course, does not always converge, and Vichnevetsky contains a discussion of such issues [27].

[^49]
## A.1.3 Gauss-Seidel iteration

A slight variation, called Gauss-Seidel iteration, uses:

$$
\begin{equation*}
x_{k+1}(i)=\frac{-\sum_{j=1}^{i-1} a_{i j} x_{k+1}(j)-\sum_{j=i+1}^{n} a_{i j} x_{k}(j)+\sum_{j=1}^{n} a_{i j} b_{j}}{a_{i i}} \tag{A.7}
\end{equation*}
$$

That is, instead of updating all components $x_{k}(i)$ synchronously, the new components are used as soon as they become available. In matrix form, this means:

$$
\begin{equation*}
(L+D) x_{k+1}+U x_{k}=b \tag{A.8}
\end{equation*}
$$

or

$$
\begin{equation*}
x_{k+1}=-(L+D)^{-1} U x_{k}+(L+D)^{-1} b \tag{A.9}
\end{equation*}
$$

This method is somewhat better than the Jacobi method in that it updates the components successively instead of synchronously, so the storage requirements are less stringent and programs are generally more compact and efficient. However, one should be careful in using these methods because their convergence properties are different, although for a large class of problems they both converge.

## A.1.4 Overrelaxation

These iterative methods are, in general, relatively slow. In order to speed up the convergence, one often uses overrelaxation techniques by taking larger "steps" in each iteration. For Jacobi iteration, this means using:

$$
\begin{equation*}
x_{k+1}-x_{k}=\bar{\omega}\left(\left(I-D^{-1}(L+U)\right) x_{k}+D^{-1} b\right) \tag{A.10}
\end{equation*}
$$

or

$$
\begin{equation*}
x_{k+1}=\left((1-\bar{\omega}) I-\bar{\omega} D^{-1}(L+U)\right) x_{k}+\bar{\omega} D^{-1} b . \tag{A.11}
\end{equation*}
$$

The number $\bar{\omega}$ is called the overrelaxation factor when $1<\bar{\omega}<2$, and called the underrelaxation factor when $0<\bar{\omega}<1$. One could show that the iteration must necessarily diverge (that is, the spectral radius of the resulting iteration matrix $B$ must be greater than 1) unless $0<\bar{\omega}<2$. However, the converse does not hold: $0<\bar{\omega}<2$ does not guarantee convergence.

For Gauss-Seidel, a similar derivation yields:

$$
\begin{equation*}
x_{k+1}=\left((1-\bar{\omega}) I-\bar{\omega}(L+D)^{-1} U\right) x_{k}+\bar{\omega}(L+D)^{-1} b . \tag{A.12}
\end{equation*}
$$

This is known as successive overrelaxation.

## A. 2 A brief introduction to finite elements

## A.2.1 Introduction

This section briefly summarizes how numerical solutions of partial differential equations can be computed using the finite element method. In particular, it contains a derivation of the standard discretization of Laplace's equation in two dimensions. Most of this material
comes from Vichnevetsky [27]; it is an excellent introduction to numerical methods for partial differential equations. Johnson [17] also contains a clear and more detailed account of finite element methods. For an analytical approach, the opening chapters of Fritz John's text [16] offer a good introduction. The classic treatise on partial differential equations is Courant and Hilbert [10], which may be too encyclopedic to serve as an introduction but contains a lot of good stuff. A very brief but clear survey article appears in the McGraw-Hill Encyclopedia of Science \& Technology [5].

## Notational and mathematical conventions

This section will not rigorously define such terms as open set, closed set, and boundary, since these topological concepts should be fairly intuitive in this setting. It will only define some notations and terms not commonly covered in introductory calculus courses.

The boundary of a region $\Omega$ is denoted by $\partial \Omega$, and its closure $\bar{\Omega}$ is defined as the union of $\Omega$ and its boundary. Given a real-valued function $f$ over $\Omega$, its support is defined as the closure of the subset of points over which $f$ is nonzero, i.e. the set $\overline{\{x \in \Omega \mid f(x) \neq 0\}}$.

For the sake of precision (which is important for turning ideas into programs), functional notation will be used wherever appropriate. Thus, the integral of a real-valued function $f$ over open set $\Omega$ is

$$
\begin{equation*}
\int_{\Omega} f \tag{A.13}
\end{equation*}
$$

instead of

$$
\begin{equation*}
\int_{\Omega} f(x, y) d x d y \tag{A.14}
\end{equation*}
$$

That the above is an area integral should be clear from the context, since $\Omega$ is an open subset of the plane. Similarly, differential operators will operate on functions, not expressions. More precisely:

$$
\begin{equation*}
\frac{d}{d t} f(t)=(D f)(t), \frac{\partial}{\partial x} f(x, y)=\left(D_{1} f\right)(x, y), \frac{\partial^{n}}{\partial x^{n}} f(x, y)=\left(D_{1}^{n}\right) f(x, y) \tag{A.15}
\end{equation*}
$$

and so on. And, unless otherwise specified, all functions considered here will be continuously differentiable up to whatever order is required in its context. Note that, for emphasis, the derivative of a function evaluated at $t$ was written as $(D f)(t)$ above, but in general the differential operator $D$ takes precedence over functional evaluation, and $(D f)(t)=D f(t)$.

One last bit of notational convenience is the multi-index notation. A multi-index $\alpha$ is an $n$-tuple of non-negative integers ( $\alpha_{1}, \alpha_{2}, \ldots, \alpha_{n}$ ). Given an $n$-vector $x$, define $x^{\alpha}$ to be $x_{1}^{\alpha_{1}} \cdot x_{2}^{\alpha_{2}} \cdot \ldots \cdot x_{n}^{\alpha_{n}}$. Also, define the gradient operator $\nabla=D=\left(D_{1}, D_{2}, \ldots, D_{n}\right)$. Then $D^{\alpha}$ gives us a useful way to denote the differential operator $D_{1}^{\alpha_{1}} \ldots D_{n}^{\alpha_{n}}$. For convenience, define $|\alpha|$ to be $\alpha_{1}+\ldots+\alpha_{n}$.

## A.2.2 Partial differential equations

Before a discussion of algorithms for solving partial differential equations, some terminology and examples are needed. The focus here is on scalar differential equations, though some of these methods generalize to systems of equations.

## Basic definitions

A partial differential equation for a real-valued function $u$ of $n$ real variables is a relation of the form

$$
\begin{equation*}
F\left(x, u(x), D_{1} u(x), \ldots, D_{n} u(x), D_{1}^{2} u(x), \ldots\right)=0 \tag{A.16}
\end{equation*}
$$

where $F$ is real-valued function of finitely many real variables and $x$ denotes a real vector with $n$ components. A function $u$ is a solution of the PDE over the domain $\Omega$ if it satisfies Equation (A.16) for all $x$ in $\Omega . \quad F$ constrains the value of the solution $u$ and a finite number of its partial derivatives, and may depend on the coordinates. The order of a partial differential equation is the order of the highest-order partial derivative that appears in Equation (A.16). Depending on the specific function F, Equation (A.16) may have no solution, a unique solution, or more than one solution; the existence theory for solutions of partial differential equations is a large and complicated subject, and this report makes no attempt at presenting it. An $m$ th-order PDE is linear if it can be written as

$$
\begin{equation*}
\sum_{|\alpha| \leq m} A_{\alpha} D^{\alpha} u=L u=b, \tag{A.17}
\end{equation*}
$$

where the coefficients $A_{\alpha}$ of $L$, as well as $b$, are functions of the coordinates. This class of equations will be the most important to us.

Many equations arising from applications have infinitely many solutions, so one must prescribe additional constraints to obtain unique solutions. For an equation of order $m$ on a domain of dimension $n$, these constraints usually involve specifying the values of the solution and its derivatives of order less than $m$ on some ( $n-1$ )-dimensional subspace of the domain of solution. If a partial differential equation along with a constraint has a unique solution, the problem is said to be well-posed. ${ }^{2}$ As we shall see, different types of equations require different constraints to have existence and uniqueness of solutions. For example, some constraints make the equation overdetermined; that is, there may not be a solution of the differential equation that satisfies the given constraint. On the other hand, some constraints may make the equation underdetermined, and there may be more than one solution. In these cases, the problem is said to be ill-posed.

This section deals with equations over open subsets $\Omega$ of the plane ( $n=2$ ). Moreover, it concentrates on equations that are linear and homogeneous with constant coefficients:

$$
\begin{equation*}
a D_{1}^{2} u+b D_{1} D_{2} u+c D_{2}^{2} u+d D_{1} u+e D_{2} u+f u=0 \tag{A.18}
\end{equation*}
$$

where $a, b, c, d, e$, and $f$ are arbitrary real constants. Slightly more general equations are treated later.

## Laplace's equation

Here are, without proof, a number of facts regarding Laplace's equation. Given an open subset $\Omega$ of the plane, Laplace's equation for two variables is

$$
\begin{equation*}
D_{1}^{2} u+D_{2}^{2} u=0 \tag{A.19}
\end{equation*}
$$

[^50]where $u$ is a real-valued function on $\bar{\Omega}$. A function satisfying Laplace's equation is said to be harmonic. It is clear that Laplace's equation is a special case of Equation (A.18). In general, it has infinitely many solutions on a given domain $\Omega$. However, given a real-valued function $f$ on the boundary $\partial \Omega$, the requirement that the solution $u$ agrees with $f$ on $\partial \Omega$, i.e.,
\[

$$
\begin{equation*}
u(x, y)=f(x, y),(x, y) \in \partial \Omega \tag{A.20}
\end{equation*}
$$

\]

uniquely determines the solution $u$; this is one of the clasical results in the theory of PDEs. Note that in this case, our constraint only specifies the values of the solution on the boundary, not the values of its first partials.

Equation (A.20) is called the boundary condition, and Equations (A.19) and (A.20) together form the boundary value problem. Solving this equation allows us to determine, for example, the electric potential in a bounded, charge-free region given the potential on the boundary.

There is a beautiful way to reformulate the boundary value problem for Laplace's equation as a minimization problem. Let $f$ be a real-valued function on $\partial \Omega$, and let $X_{f}$ be the set of all real-valued functions $u$ on $\bar{\Omega}$ that agree with $f$ on $\partial \Omega$. Define the real-valued mapping

$$
\begin{equation*}
S(u)=\frac{1}{2} \int_{\Omega}\left(\left(D_{1} u\right)^{2}+\left(D_{2} u\right)^{2}\right) \tag{A.21}
\end{equation*}
$$

on the function space $X_{f} ; S$ is called the action. One can show that, among all functions $u$ that satisfy the boundary conditions, the solution of Laplace's equation minimizes $S$. This is an example of a variational principle, and is discussed in more detail in §refsec:variational.

## A.2.3 The Rayleigh-Ritz Method

Typically, the numerical solution of a partial differential equation involve two distinct steps. First, a way of representing the approximate solution is chosen, and the differential equation is reduced to some set of simpler equations that determine the approximate solution; this is known as discretization. Next, the discretized equations are solved, yielding the approximate solution. This section discusses only discretization methods, whereas the solution of large systems of linear algebraic equations was briefly described in §A.1. For a more thorough discussion of both aspects of this problem, see Vichnevetsky [27].

The specific discretization method developed here is known as the Rayleigh-Ritz method. The basic idea behind this method is simple: Given the domain $\Omega$ and a prescription of the boundary value $f$, choose a set of $N$ functions $\left\{\phi_{i}\right\}$ on $\bar{\Omega}$ and express the solution $u$ as a linear combination

$$
\begin{equation*}
u=\sum_{i=1}^{N} a_{i} \phi_{i} . \tag{A.22}
\end{equation*}
$$

The functions $\phi_{i}$ are the basis functions, and the Rayleigh-Ritz method requires them to have some specific properties (these are discussed later). These properties allow us to interpret the coefficients $a_{i}$ as values of the approximate solution $u$ at pre-specified sample points (or nodes) $p_{i}$. Having specified a representation of approximate solutions, an approximation of the action can be computed as a function of the unknown coefficients $a_{i}$ and the given boundary values. Minimizing this approximate action turns out to produce a system of linear equations, which can be solved to yield the coefficients $a_{i}{ }^{3}$

[^51]

Figure A-1: Finite elements on the unit disc.

## Constructing basis functions

The Rayleigh-Ritz method and a large class of other methods are collectively referred to as finite element methods because they all represent approximate solutions as linear combinations of a special type of basis functions. They rely on dividing the domain into a finite number of simple shapes, called elements, and expressing the approximate solution over each element as a sum of simple shapes. In what follows, the shapes are assumed to be triangles for simplicity, though in general they can be more complicated.

Here is a more detailed description of triangular elements: Choose a finite set of sample points $\left\{p_{i}\right\}$ in the domain $\bar{\Omega}$, such that the subset of sample points lying on the boundary $\partial \Omega$ is non-empty. Choose a finite collection of triangular subsets $T_{i}$ of $\bar{\Omega}$, such that the $T_{i}$ intersect each other only along their boundaries, and the sample points are precisely the vertices of the triangles. Furthermore, the union of the triangles $T_{i}$ should closely approximates $\Omega .^{4}$ As an example, Figure A-1 shows a crude division of the unit circle into triangular elements. In general, the more finely the elements tesselate the domain $\Omega$, the more accurate the approximate solution will be.

To each sample point $p_{i}$ we now associate a basis function $\phi_{i}$. Intuitively, the basis function $\phi_{i}$ is produced by "pasting together" simple shapes over each element adjacent to $p_{i}$; this arrangement, as will be shown later, makes the computation of finite element coefficients more efficient. More precisely, these are the requirements on the basis functions:

1. $\phi_{i}\left(p_{j}\right)= \begin{cases}1, & i=j \\ 0, & i \neq j\end{cases}$
2. $\sum_{i=1}^{N} \phi_{i}(x, y)=1$ for all $(x, y) \in \bar{\Omega}$.
3. The functions $\phi_{i}$ should be piecewise-differentiable, if not smooth everywhere.
4. The function $\phi_{i}$ should be nonzero only in the elements immediately adjacent to $p_{i}$.
[^52]The first requirement guarantees that if a function $u$ is expressed as linear combination of the basis $\left\{\phi_{i}\right\}$, as in Equation (A.22), then its $i$ th coefficient is simply

$$
\begin{equation*}
a_{i}=u\left(p_{i}\right) \tag{A.23}
\end{equation*}
$$

This is a particularly nice property, for if $p_{i}$ is a sample point on the boundary $\partial \Omega$, then the value of the approximate solution, $u\left(p_{i}\right)$, is just the given boundary value $f\left(p_{i}\right)$. But then $a_{i}=f\left(p_{i}\right)$, so that in the linear combination (A.22), the coefficients which correspond to boundary nodes do not need to be computed at all, thus reducing the number of unknowns. This is the way through which boundary values help determine the unknowns.

The second requirement ensures that if $a_{1}=a_{2}=\ldots=a_{N}=a$, then $u(x, y)=a$ for all $(x, y) \in \bar{\Omega}$; that is, constant functions are interpolated exactly by these basis functions. This ensures, for example, that if two approximate solutions constructed from these basis functions have the same values at all sample points, then they are equal everywhere. ${ }^{5}$

The third requirement is necessary because in the process of discretizing the PDE, it is necessary to take partial derivatives of the approximate solution. Finally, the last requirement makes precise the idea of pasting together simple shapes over elements adjacent to $p_{i}$. Note that the support of a basis function corresponding to a node $p$ is simply the union of the elements with $p$ as a vertex, and that the intersection of the supports of two basis functions must also be a union of elements. This is an important property for finite element computation.

One way of constructing basis functions that satisfy the requirements above are the socalled "tent functions," which are piecewise-linear functions constructed by linearly interpolating between neighboring nodes, and to let $\phi_{i}$ vanish uniformly outside of the elements adjacent to $p_{i}$. Note that by continuity, the last condition implies that $\phi_{i}=0$ at all nodes except for $p_{i}$, so requirements 1 and 4 are redundant.

## Discretization

With a specific representation of approximate solutions, one can now compute the unknown coefficients. To do this, use the approximate solution $u$ to compute an approximation of the action (A.21). This is then a real-valued mapping that depends on the (finitely many) unknown coefficients of $u$. One can then minimize this approximate action by equating its derivative to zero, which yields a set of linear equations. ${ }^{6}$

At this point, it is helpful to keep track of nodes that lie on the boundary $\partial \Omega$. Thus, let us relabel the sample points so that the basis functions $\beta_{i}, i=1,2, \ldots, N$ correspond to sample points on the boundary of $\Omega$, and let $\phi_{i}, i=1,2, \ldots, M$ continue to denote those that correspond to interior nodes. Let $a_{i}$ denote the coefficients of $\phi_{i}$, and let $b_{i}$ denote those of $\beta_{i}$. As noted in the previous section, the $N$ variables $b_{1}, b_{2}, \ldots, b_{N}$ are precisely the boundary values at the sample points on the boundary, so the only unknown values are $a_{1}, a_{2}, \ldots, a_{M} .{ }^{7}$

[^53]Let $T$ be a real-valued function of the $M$ unknown variables $a_{i}$, defined by

$$
\begin{align*}
T\left(a_{1}, \ldots, a_{M}\right) & =S\left(u\left[a_{1}, a_{2}, \ldots, a_{M}\right]\right)  \tag{A.24}\\
& =\frac{1}{2} \int_{\Omega}\left(D_{1} u\left[a_{1}, a_{2}, \ldots, a_{M}\right]\right)^{2}+\left(D_{2} u\left[a_{1}, a_{2}, \ldots, a_{M}\right]\right)^{2} \tag{A.25}
\end{align*}
$$

where $u\left[a_{1}, a_{2}, \ldots, a_{M}\right]$ is defined by

$$
\begin{equation*}
u\left[a_{1}, a_{2}, \ldots, a_{M}\right]=\sum_{i=1}^{M} a_{i} \phi_{i}+\sum_{i=1}^{N} b_{i} \beta_{i} \tag{A.26}
\end{equation*}
$$

and the action $S$ was defined in Equation (A.21).
To minimize $T$, simply differentiate under the integral sign. Via the chain rule, the partial derivatives of $T, D_{j} T\left(a_{1}, \ldots, a_{M}\right)$, are:

$$
\begin{equation*}
\int_{\Omega}\left(\left(\sum_{i=1}^{M} a_{i} D_{1} \phi_{i}+\sum_{i=1}^{N} b_{i} D_{1} \beta_{i}\right) \cdot D_{1} \phi_{j}+\left(\sum_{i=1}^{M} a_{i} D_{2} \phi_{i}+\sum_{i=1}^{N} b_{i} \beta_{i}\right) \cdot D_{2} \phi_{j}\right), \tag{A.27}
\end{equation*}
$$

for $j=1,2, \ldots, M$. Equating the derivatives of $T$ to 0 produces a system of equations:

$$
\begin{equation*}
\sum_{i=1}^{M} a_{i} \int_{\Omega}\left(D_{1} \phi_{i} \cdot D_{1} \phi_{j}+D_{2} \phi_{i} \cdot D_{2} \phi_{j}\right)=-\sum_{i=1}^{N} b_{i} \int_{\Omega}\left(D_{1} \beta_{i} \cdot D_{1} \phi_{j}+D_{2} \beta_{i} \cdot D_{2} \phi_{j}\right) \tag{A.28}
\end{equation*}
$$

with $j=1,2, \ldots, M$. This is a system of $M$ linear equations in $M$ unknowns. Indeed, let

$$
\begin{equation*}
a_{i j}=\int_{\Omega}\left(D_{1} \phi_{i} \cdot D_{1} \phi_{j}+D_{2} \phi_{i} \cdot D_{2} \phi_{j}\right) \tag{A.29}
\end{equation*}
$$

and let $A$ be the matrix $\left(a_{i j}\right)$. Define the $M$-vector

$$
\begin{equation*}
b=\left(-\sum_{i=1}^{N} b_{i} \int_{\Omega}\left(D_{1} \beta_{i} \cdot D_{1} \phi_{j}+D_{2} \beta_{i} \cdot D_{2} \phi_{j}\right)\right) . \tag{А.30}
\end{equation*}
$$

Then Equation (A.28) becomes simply

$$
\begin{equation*}
A u=b \tag{A.31}
\end{equation*}
$$

where $u$ is the vector of the unknown coefficients $a_{i}$.

## Some comments on finite elements

The derivation of the discretized equations (A.28) involves many integrals. But recall now that the basis functions were chosen so that a basis function associated with the node $p_{i}$ is nonzero only over those elements adjacent to $p_{i}$. Thus, the integrals in Equation (A.28) need only be evaluated over a finite number of elements. One can generally choose element shapes

[^54]

Figure A-2: Rectangular finite elements.
and basis functions to simplify the computation of these integrals, and the primary reason for the popularity of finite element methods is the efficiency with which these coefficients can be computed.

Additionally, this locality mirrors the fact that in many physical systems, most interactions are local and effects propagate with finite speed through the system. And because a coefficient is nonzero only if two nodes are neighbors (in the sense that they are vertices of the same element), the matrix $A$ defined by Equation (A.29) is usually sparse; that is, it contains many zeros. This lessens the storage requirements when working with systems with large numbers of sample points, as well as making iterative solution methods like relaxation more efficient. ${ }^{8}$

## Example

As an example, let us derive the standard finite difference equations for the boundary value problem using the Rayleigh-Ritz method. Consider a rectangular grid of points in the plane, a subset of which is shown in Figure A-2.

Let's use piecewise-linear tent functions on the elements, and suppose that the elements are isoceles triangles with base and height $h$. Let $\phi_{i}$ denote the basis function corresponding to the node $p_{i}$; it is then a tent function with its tip at the point $p_{i}$. To compute the coefficients corresponding to a typical node $p_{0}$ in the matrix $\mathbf{A}=\left(a_{i j}\right)$ of Equation (A.29), let $c_{i}=a_{0, i} .^{9}$ Since the interpolants are linear, their gradients are constant. Hence, the coefficients are simply the dot products of the interpolants multiplied by the area of the intersection of their supports; denote the intersection Support $\left(\phi_{i}\right) \cap \operatorname{Support}\left(\phi_{j}\right)$ by $\Omega_{(i, j)}$. The row of $\mathbf{A}$ corresponding to $p_{0}$ can have at most six non-zero entries, since $p_{0}$ has only six neighbors - namely $p_{1}, p_{2}, p_{3}, p_{5}, p_{6}$, and $p_{7} . p_{4}$ and $p_{8}$ (as well as any nodes in the system

[^55]that are not pictured in Figure A-2) are not neighbors of $p_{0}$ and hence those coefficients must vanish.

To compute $c_{1}$, note that $\Omega_{(1,0)}=A \cup G$. Over the region $A$, the gradients are

$$
\begin{equation*}
\nabla \phi_{1}=\frac{1}{h}(1,1), \nabla \phi_{0}=\frac{1}{h}(0,-1) \tag{A.32}
\end{equation*}
$$

and over the region $G$, they are

$$
\begin{equation*}
\nabla \phi_{1}=\frac{1}{h}(0,1), \nabla \phi_{0}=\frac{1}{h}(-1,-1) \tag{A.33}
\end{equation*}
$$

The area of each element is $\frac{1}{2} h^{2}$, so the coefficient $c_{1}$ is simply -2. Similarly, $c_{3}=c_{5}=c_{7}=$ -2 and $c_{2}=c_{6}=0$. Finally, $c_{0}=1+1+1+1+2+2=8$, so the $p_{0}$ equation of the system $\mathbf{A} x=b$ is $8 c_{0}-2 c_{1}-2 c_{3}-2 c_{5}-2 c_{7}=0$, which upon rearrangement yields

$$
\begin{equation*}
c_{0}=\frac{c_{1}+c_{3}+c_{5}+c_{7}}{4} \tag{А.34}
\end{equation*}
$$

Equation (A.34) is simply the standard finite difference approximation for Laplace's equation, and similar computations yield the same equations for the case when $p_{0}$ is on the boundary $\partial \Omega$.

## A.2.4 Galerkin's method

Another commonly-used finite element method is Galerkin's method. In many cases, it produces equations equivalent to the Rayleigh-Ritz equations. However, this method differs in that it is slightly more difficult to justify mathematically, even though it is more generally applicable, especially in situations where a variational principle is not available. We derive Galerkin's method by a close analogy with a slightly more general function-expansion method, which also uses expansion in terms of basis functions to solve differential equations.

As before, basis functions are denoted by $\left\{\phi_{i}, i=1,2,3, \ldots\right\}$; however, these functions are not, for the moment, necessarily of the type considered in Rayleigh-Ritz. Furthermore, representations of functions as (possibly infinite) linear combinations of these basis functions is assumed to be exact, so the set of basis functions (called the basis) will no longer be finite. Given two real-valued functions $f$ and $g$ on $\Omega$, define the inner product $\langle f, g\rangle$ by

$$
\begin{equation*}
\langle f, g\rangle=\int_{\Omega} f \cdot g \tag{A.35}
\end{equation*}
$$

The basis is required to be complete, in the sense that if a function $u$ satisfies $\left\langle u, \phi_{i}\right\rangle=0$ for all $i$, then $u=0$ uniformly on $\Omega$. For example, if $\Omega$ is a bounded interval of the real line, one can choose the $\phi_{i}$ to be Legendre polynomials or sinusoidal functions; both form complete bases.

Back to Laplace's equation now: Recall that this involves finding a real-valued function $u$ on $\bar{\Omega}$ such that

$$
\begin{equation*}
D_{1}^{2} u+D_{2}^{2} u=0 \tag{A.36}
\end{equation*}
$$

on $\Omega$ and $u=f$ on the boundary $\partial \Omega$ for some prescribed function $f$. Expanding the solution as an infinite series

$$
\begin{equation*}
u=\sum_{i=1}^{\infty} a_{i} \phi_{i}(x, y) \tag{A.37}
\end{equation*}
$$

over a complete basis, the problem reduces to the determination of the unknown coefficients. Using completeness, this is equivalent to

$$
\begin{equation*}
\int_{\Omega}\left(D_{1}^{2} u+D_{2}^{2} u\right) \phi_{i}=0, i=1,2,3, \ldots \tag{A.38}
\end{equation*}
$$

Expanding $u$ in its infinite series, the above equation becomes

$$
\begin{equation*}
\sum_{i=1}^{\infty} a_{i} \int_{\Omega}\left(D_{1}^{2} \phi_{i}+D_{2}^{2} \phi_{i}\right) \phi_{j}=0, j=1,2,3, \ldots \tag{A.39}
\end{equation*}
$$

Galerkin's method generalizes this procedure to the case when the basis is finite, and therefore not complete.

More specifically, let $\left\{\phi_{i}, i=1,2, \ldots, M\right\}$ and $\left\{\beta_{i}, i=1,2, \ldots, N\right\}$ now denote the finite element basis functions considered in $\S$ A.2.3, where, as before, $M$ nodes lie in the interior of $\Omega$ and $N$ nodes lie on the boundary. Since the finite element basis is finite, it cannot be a complete basis for the solution space (which is generally infinite-dimensional). However, by analogy with Equation (A.39), one can still require that the residual be orthogonal to the basis functions, producing

$$
\begin{equation*}
\sum_{i=1}^{M} a_{i} \int_{\Omega}\left(D_{1}^{2} \phi_{i}+D_{2}^{2} \phi_{i}\right) \phi_{j}=-\sum_{i=1}^{N} b_{i} \int_{\Omega}\left(D_{1}^{2} \beta_{i}+D_{2}^{2} \beta_{i}\right) \phi_{j} \tag{A.40}
\end{equation*}
$$

with $j=1,2, \ldots, M$. Integrating by parts and noting that each basis functions vanishes outside a bounded region, the equations become

$$
\begin{equation*}
-\sum_{i=1}^{M} a_{i} \int_{\Omega}\left(D_{1} \phi_{i} \cdot D_{1} \phi_{j}+D_{2} \phi_{i} \cdot D_{2} \phi_{j}\right)=\sum_{i=1}^{N} b_{i} \int_{\Omega}\left(D_{1} \beta_{i} \cdot D_{1} \phi_{j}+D_{2} \beta_{i} \cdot D_{2} \phi_{j}\right) \tag{A.41}
\end{equation*}
$$

again for $j=1,2, \ldots, M$. These equations are identical, up to a sign, to (A.28).
Let $u$ denote the approximate solution given by Galerkin's method. Galerkin's method only requires that the residual $D_{1}^{2} u+D_{2}^{2} u$ lies in the orthogonal complement of the span of the basis. Thus, without some other criterion to justify the equations, Galerkin's method does not actually guarantee that the approximate solution satisfies the differential equation in any sense. Notice the resemblance between the orthogonality condition and least-squares approximations: Recall that if $\phi$ is a function to be approximated, and $u$ is linear combination of basis functions $\phi_{i}$, then the orthogonality condition $\left\langle\phi-u, \phi_{i}\right\rangle=0$ indeed produces the least-squares approximation. But in this case, the exact solution $\phi$ is not available to us. Thus, Galerkin's method does not actually produce the least squares approximation. Orthogonalizing the residual does not minimize it. Indeed, since the basis is not complete, the error residual can be arbitrarily large while still being orthogonal to all the basis functions.

## Appendix B

## Odds \& Ends

This appendix develops a few more basic ideas of manifold theory for the interested reader. This is mainly "just for fun," and is meant to provide a slightly more detailed look at the definitions of Chapter 2 for the interested reader.

## B. 1 Charts without a manifold

As we have seen, the description of spaces that require more than one coordinate system can be effectively accomplished using the manifold abstraction. More concretely, we started with an arbitrary set of points $M$ and defined enough structure on it that it allowed us to define smooth maps and their differentials on the set. It can be shown, however, that all one needs are the charts themselves: The set $M$ itself is unnecessary, and can be constructed out of charts. However, this definition of a system of coordinate charts without an underlying manifold is equivalent to the usual one; this section is simply an interesting exercise in manipulating the mathematical definition of manifolds.

Let $\mathcal{S}=\left\{U_{i}\right\}$ be a collection of open subsets of $R^{n}$. Let $T$ be a mapping on $\mathcal{S} \times \mathcal{S}$ that assigns to each pair of open sets $\left(U_{i}, U_{j}\right)$ a triple $\left(V_{i}, V_{j}, f_{i j}\right)$ such that:

1. $V_{i}$ and $V_{j}$ are (possibly empty) open subsets of $U_{i}$ and $U_{j}$, respectively.
2. $f_{i j}$ is a smooth bijection from $V_{i}$ to $V_{j}$.
3. If $U_{i}=U_{j}$, then $V_{i}=V_{j}=U_{i}$ and $f_{i j}$ is the identity.
4. If $U_{i}, U_{j}, U_{k} \in \mathcal{S}$, then $f_{i k}=f_{j k} \circ f_{i j}$, and the domains $V_{i}, V_{j}$, and $V_{k}$ are mapped onto each other by these maps and their inverses.

Let us define a coordinate manifold to be a collection of open sets $\mathcal{S}$ with such a map $T$. It is clear that whenever we are given a manifold in the usual sense, it can be converted to a coordinate manifold. Conversely, given a coordinate manifold, one could construct an underlying space for it: Let $x \in U_{i}$ for some open set $U_{i} \in \mathcal{S}$. The pair ( $x, U_{i}$ ) is called a coordinate, and two coordinates $\left(x, U_{i}\right)$ and $\left(x^{\prime}, U_{j}\right)$ are equivalent if the open sets $V_{i}$ and $V_{j}$ assigned to $U_{i}$ and $U_{j}$ by $T$ are nonempty, and if $f_{i j}(x)=x^{\prime}$. Under this equivalence relation, the equivalence classes are called points, and the collection $M$ of all points constructed this way is the underlying space of the manifold.

What good is this definition? It is absolutely useless for all theoretical purposes: The underlying set $M$ in the usual definition lets us apply the tools of topology, which are very
important in the theory of manifolds. However, for all practical purposes, coordinate manifolds are actually enough: The programs in Chapters 2 and 3 only use local coordinates, and points on the manifold are simply used as identifiers. Furthermore, in usual computations involving manifolds, one carries out most calculations explicitly in specific charts anyway. This way of treating points on manifolds is very much in the spirit of the construction of tangent vectors.

## B. 2 Integration of differential forms on oriented manifolds

This section briefly sketches the construction of differential forms, which are "functions" that can be integrated on oriented manifolds. While they are of less importance in the theory of differential equations on manifolds, they are very much essential in the study of differential topology. However, those applications would take us too far afield and will not be discussed here. For more information, please see either Guillemin and Pollack [14] or Warner [28].

Recall the change of variables theorem (3.19):

$$
\begin{equation*}
\int_{y \in V_{2}} f(y) d y=\int_{x \in V_{1}} f(\phi(x))|\operatorname{det} D \phi(x)| d x \tag{B.1}
\end{equation*}
$$

where $f$ is a function on $B$ and $\phi: A \rightarrow B$ is a smooth bijection. In $\S 3.4 .1$, this theorem is used to define integrals of scalar-valued functions on compact Riemannian manifolds. Another possible approach, which we will briefly sketch here, involves assigning "determinant-like" functions to each tangent space of the manifold. Such an assignment is called a "differential form."

Let $V$ be a finite-dimensional vector space. A scalar-valued function $T$ on $V \times \ldots \times V$ is multilinear if it is linear in each of its components, and is alternating if exchanging any two arguments changes $T$ to $-T$. The degree of $T$ is the number of arguments $T$ has. It is a theorem of linear algebra that such functions, called alternating tensors, are always proportional to the determinant function on $V$ with respect to some basis.

Now, let $\omega$ be a function that assigns to each point $p \in M$ an alternating tensor $\omega_{p}$ on $T_{p} M$. One can show (although it is not done here) that the usual notions of smoothness also apply to these alternating tensor fields. A differential form on a manifold $M$ is then a smooth alternating tensor field on $M$.

Because alternating tensors are proportional to the determinant function, it is not very difficult to show that one could obtain a consistent definition of integration for differential forms. To do this, first choose a partition of unity so that the problem is reduced to a local one. Then, note that tensors transform naturally in the following way:

$$
\begin{equation*}
T^{\prime}\left(v_{1}, \ldots, v_{k}\right)=T\left(L v_{1}, \ldots, L v_{k}\right) \tag{B.2}
\end{equation*}
$$

where $T^{\prime}$ is a tensor on some vector space $W, T$ is a tensor on $V, L: V \rightarrow W$ is a linear transformation, and $k$ is the degree of $T^{\prime}$. Generalizing this to differential forms on manifolds, we can simply replace $L$ by the differential $d \phi$ of the transition map $\phi$. A little bit of linear algebra shows that this almost gives us the change of variables theorem:

$$
\begin{equation*}
\int_{y \in V_{2}} f(y) d y=\int_{x \in V_{1}} f(\phi(x)) \operatorname{det} D \phi(x) d x \tag{B.3}
\end{equation*}
$$

where $f$ det is a (local) differential form on $V_{2}$ and $(f \circ \phi)$ det is a differential form on $V_{1}$.
(Note that both their degrees have to agree with the dimension of the space, $n$, because of the dimensions of $D \phi$ as a matrix.) As one can see, this is the change of variables theorem except for the absolute value. Thus, if one could choose charts so that all the transition maps have positive determinants:

$$
\begin{equation*}
\operatorname{det} D \phi(x)>0, \tag{B.4}
\end{equation*}
$$

then we can define integrals consistently. Manifolds for which such atlases exist are called orientable manifolds, and we can thus define integration of differential forms of degree $n$ on compact orientable $n$-manifolds.

## B.2.1 Stokes's theorem

One of the most important things one can do with differential forms is to generalize Stokes's theorem to compact orientable manifolds. This is done through a map called the exterior derivative, which takes a differential form $\omega$ of degree $k$ to another differential form $d \omega$ of degree $k+1$. Defining $d$ takes a little bit of work and will not be done here. But to show how much Stokes's theorem is simplified, here is the statement of the theorem using differential forms:

$$
\begin{equation*}
\int_{M} d \omega=\int_{\partial M} \omega . \tag{B.5}
\end{equation*}
$$

This is actually so abstract that it does not say much, unless one has studied differential forms in some depth. However, note that the boundary operator on manifolds satisfies:

$$
\begin{equation*}
\partial(M \times N)=(\partial M \times N) \cup(M \times \partial N) \tag{B.6}
\end{equation*}
$$

just like the product rule. As there is a corresponding product rule for exterior derivatives, this shows that there is a rather deep duality between geometric objects on the one hand and algebraic structures (such as differential forms) on the other.

## Appendix C

## Complete Program Listings

This appendix contains the source code for all the programs written for this project. The programs are listed in alphabetical order, and as such are rather incomprehensible. They have been included primarily for completeness rather than for clarity.

## C. 1 Computational manifolds

The following programs implement ODE and PDE solvers on manifolds, as described in Chapters 2 and 3. All of this work is done in Scheme [9]. Interested readers may wish to begin with load-pde.scm and load-ode.scm to get a feel for the organization of the programs.

## C.1.1 basis-imb.sem

```
;;; The basis functions defined here are much like polynomial basis functions,
;;; only they exist directly on the imbedding representation of a manifold,
;;; instead of on the chart. Many of the procedures in 2d-poly-basis.scm are
;;; called here.
(declare (usual-integrations))
;;; Interface to the manifold code:
(define (pde:make-imbedded-poly-basis-function nodes i)
    (let ((basis (make-imbedded-basis-function
                                    nodes i (node:get-chart (car nodes)))))
        (node:add-basis-function (list-ref nodes i) basis)
        basis))
(define (operator:imbedded-poly-op left-op right-op combine)
    (lambda (chart nodes)
        (lambda (f g)
            (combine (left-op f) (right-op g)))))
```

;; Basic constructor:
(define (vector->imbedded $v$ chart)
(package-basis-function-methods

```
    '2d-imbeded-basis-function
    v
    (imbedded->function v chart)
    (make-2d-poly-adder v)
    (make-2d-poly-subtractor v)
    (make-2d-poly-multiplier v)
    (make-2d-poly-scalar-multiplier v)))
(define (make-imbedded-basis-function nodes center chart)
    (let* ((n (length nodes))
            (vals (make-vector n))
            (points (make-vector n)))
        (let loop ((nodes nodes) (i 0))
            (if (null? nodes)
                (vector->imbedded (poly:point-value->coeff vals points) chart)
                (let ((node (car nodes)))
                    (if (= i center)
                            (vector-set! vals i 1)
                            (vector-set! vals i 0))
                                (vector-set! points i (node:get-point node))
                            (loop (cdr nodes) (+ i 1)))))))
;;; A slightly different kind of constructor:
(define (function->imbedded f nodes)
    (let* ((n (length nodes))
                (vals (make-vector n))
                (points (make-vector n)))
            (let loop ((i 0) (nodes nodes))
            (if (null? nodes)
            (vector->poly (poly:point-value->coeff vals points))
            (let ((node (car nodes)))
                                    (vector-set! points i (node:get-point node))
                            (vector-set! vals i (f node))
                            (loop (+ i 1) (cdr nodes)))))))
;;; And its inverse:
(define (imbedded->function f chart)
    (lambda (x)
            (imbedded:evaluate f x chart)))
(define (imbedded:evaluate f x chart)
    (vector-first (poly:coeff->point-value
                            f (vector (chart:coords->point x chart)))))
```

;; The truly messy stuff: Integrals! This needs to run a lot faster. What
;; about doing away with the coordinate transformations?
(define (make-triangular-imbedded-integrator vertex-nodes)
; ; We assume that there are three vertex nodes, and that the triangle they
; f form is the boundary of the element:

```
(if (not (= (length vertex-nodes) 3))
    (error (string-append "Error: Elements must have three vertex nodes."
                                    " -- MAKE-TRIANGULAR-IMBEDDED-INTEGRATOR")))
(let ((p1 (car vertex-nodes))
        (p2 (cadr vertex-nodes))
        (p3 (caddr vertex-nodes)))
    ;; Find the absolute value of the Jacobian of the affine transformation
    ;; mapping the reference triangle {(0,0),(1,0),(0,1)} to this triangle.
    (let* ((A (list->matrix
                2}
                        (list
                (- (node:get-real-x p2) (node:get-real-x p1))
                (- (node:get-real-x p3) (node:get-real-x p1))
                (- (node:get-real-y p2) (node:get-real-y p1))
                (- (node:get-real-y p3) (node:get-real-y p1)))))
            (b (node:get-point p1))
            (jacobian (abs (det A))))
        (define (integrate f . rest)
            (let* ((f (apply basis:* (cons f rest)))
                    (degree (poly:degree f))
                    (reference (poly:make-sample-points degree))
                    (n (choose (+ degree 2) 2))
                    (real (make-vector n)))
            (do ((i 0 (+ i 1)))
                    ((>= i n))
                (vector-set! real i
                    (apply-affine-transformation
                            A b (vector-ref reference i))))
            (* jacobian
                    (inner-product
                        (poly:point-value->coeff
                        (poly:coeff->point-value (basis:get-rep f) real) reference)
                (make-reference-integrals degree)))))
    integrate)))
```


## C.1.2 basis-poly.scm

;; ; This file defines some procedures that help extend the polynomial basis
; ; ; functions used in fem.scm. Note that these functions limit the accuracy of
;;; computation because a polynomial may not stay a polynomial under coordinate
; ; transformations, and yet that is how we transform these guys between
;;; coordinate systems.
(declare (usual-integrations))
;; ; A little wrapper that let's us keep track of basis functions:
(define (pde:make-poly-basis-function nodes i)
(let ((basis (make-polynomial-basis-function nodes i)))

```
(node:add-basis-function (list-ref nodes i) basis)
```

basis))

```
;; This procedure turns an operator on \(M\), where \(M\) is represented as an
\(; ;\); imbedded submanifold of \(R^{\wedge} n\), into an operator on functions on \(M\). It
;;; depends heavily on the fact that it's working with polynomial interpolants.
(define (operator:pull-back-poly-op left-op right-op combine)
    (lambda (chart nodes)
        ;; Take a polynomial basis function, pull back to the canonical coordinates
        ;; of the ambient space, interpolate by polynomial, then apply operator to
        ; form a polynomial approximating the image of the original polynomial in
        ; ; the original chart under the differential operator.
        (let* ((pl (map node:get-point nodes))
                (pv (list->vector pl))
                (cl (map node:get-coords nodes))
                (cv (list->vector cl)))
        (lambda (f g)
            (let* ((f1 (basis-function->function f))
                (f2 (vector->poly (poly:point-value->coeff
                            (list->vector (map f1 cl)) pv)))
                            (g1 (basis-function->function g))
                (g2 (vector->poly (poly:point-value->coeff
                    (list->vector (map g1 cl)) pv)))
                        (h (basis-function->function
                            (combine (left-op f2) (right-op g2)))))
                (vector->poly (poly:point-value->coeff
                                    (list->vector (map h pl)) cv)))))))
;; Here's a problem: If we integrate purely in local coordinates, then the
;; ; integral is in fact using the *wrong* measure. In order to perform the
;;; correct integration, we need to put in the Jacobian of the coordinate
;; ; function. Since integrators are given nodes (not coordinates), and nodes
;; ; have charts attached to them, this could be done (very approximately):
(define (make-triangular-chart-integrator nodes)
    (let ((triangular-integrate (make-triangular-integrator nodes))
            (jacobian (abs (/ (apply triangle-area (map node:get-point nodes))
                            (apply triangle-area (map node:get-coords nodes)))))
        (lambda (f . rest)
            (* (apply triangular-integrate (cons f rest)) jacobian))))
(define (triangle-area a b c)
    (let ((x1 (- (vector-ref b 0) (vector-ref a 0 )))
            (y1 (- (vector-ref b 1) (vector-ref a 1)))
            (x2 (- (vector-ref c 0 ) (vector-ref a 0 )))
            (y2 (- (vector-ref c 1) (vector-ref a 1 ))))
        (abs (* 1/2 (- (* x1 y2) (* y1 x2))))))
```


## C.1.3 basis-real.scm

```
;;; This file defines a class of basis functions based on real functions (not
;;; just polynomials). At bottom, we still use polynomial basis functions, but
;;; these guys don't get truncated under changes of coordinates. The drawback
;;; is that we actually need to use numerical integration, which is less
;;; accurate and a lot slower.
(declare (usual-integrations))
```

;;; Constructor:
(define (pde:make-real-basis-function nodes i)
(let ((f (make-real-basis-function nodes i)))
(node:add-basis-function (list-ref nodes i) f)
f))
;;; Differential operators:
(define (operator:pull-back-real-op left-op right-op combine)
(lambda (chart nodes)
;; Take a basis function, pull back, apply operator, and then send it back
;; to the chart.
(let ((coord-map (chart:get-coord-map chart))
(inverse-map (chart:get-inverse-map chart)))
(lambda (f g)
(let ((f (proc->real (compose (basis:get-rep f) coord-map)))
(g (proc->real (compose (basis:get-rep g) coord-map))))
(let ( h (combine (left-op f) (right-op g))))
(proc->real (compose (basis:get-rep h) inverse-map))))))))
; ; ; This integrates with the wrong *measure*, though. What is required is to
; ; ; take into account the Jacobian of the coordinate charts. (See
;;; basis-poly.scm, where this is done very approximately.) Df course, this
;; ; particular approach assumes that the manifold is imbedded in some Euclidean
;; ; space, which can be restrictive for some applications. To fix this, we
; ; ; probably need some computational representation of Riemann metrics or
; ; ; differential forms on manifolds.
(define (trapezoidal-integrator-maker-on-charts count)
(let ((make-integrator (trapezoidal-integrator-maker count)))
(lambda (nodes)
(let* ((integrate (make-integrator nodes))
(g (chart:get-inverse-map (node:get-chart (car nodes))))
(dg (proc->real (function->jacobian g)))
(lambda (f . rest)
(apply integrate '(,dg ,f , ©rest)))))))
; ; Given that $F$ is an imbedding of a subset of the plane in a
;; ; higher-dimensional Euclidean space, how do we (efficiently) compute its
;;; Jacobian?

```
;;; This guy will currently work only if F goes from R^2 to R^2. Needs fixing.
(define (function->jacobian f)
    (lambda (x)
            (let ((M (jacobian-matrix f x)))
                    (abs (det M)))))
(define (jacobian-matrix f x)
    (let ((n (vector-length x))
                (m (vector-length (f x))))
        (let ((mat (make-matrix m n)))
            (do ((j (- n 1) (- j 1)))
                    ((< j 0) mat)
                (let ((v (((pdiff j) f) x)))
                    (do ((i O (+ i 1)))
                        ((>= i m))
                            (matrix-set! mat i j (vector-ref v i)))))))
```


## C.1.4 boundary.scm

```
;;; Manifolds with boundary are probably going to be useful for PDEs:
```

;;; Manifolds with boundary are probably going to be useful for PDEs:
(declare (usual-integrations))
(declare (usual-integrations))
;;; Boundary charts? What extra structures are needed? By convention, a
;;; Boundary charts? What extra structures are needed? By convention, a
;;; boundary chart maps the boundary to the half space {x_n >= 0}, so that the
;;; boundary chart maps the boundary to the half space {x_n >= 0}, so that the
;;; boundary is the space {x_n = 0}.
;;; boundary is the space {x_n = 0}.
;;; Of course, any changes made here propagate to tangent and product chart
;;; Of course, any changes made here propagate to tangent and product chart
;;; constructions... :(
;;; constructions... :(
;;; The problem is that the product of two smooth manifolds with boundary will
;;; The problem is that the product of two smooth manifolds with boundary will
;;; be a *topological* manifold with boundary, but points (p,q) where p and q
;;; be a *topological* manifold with boundary, but points (p,q) where p and q
;;; are in the respective boundaries of the component manifolds may not have a
;;; are in the respective boundaries of the component manifolds may not have a
;;; neighborhood that maps to the boundary of a Euclidean half-space.
;;; neighborhood that maps to the boundary of a Euclidean half-space.
;;; (Consider the product of the unit interval with itself: There are corners!)
;;; (Consider the product of the unit interval with itself: There are corners!)
;;; Make these regular domains imbedded inside manifolds and that avoids the
;;; Make these regular domains imbedded inside manifolds and that avoids the
;;; problem -- Can't make those constructions...
;;; problem -- Can't make those constructions...
(define (add-boundary-to-chart chart i . argl)
(let ((level 0))
;; Locally, the boundary should look like the set of all X such that
;; COORD-MAP(X)[i] = LEVEL, where LEVEL is by default 0.
(if (and (not (null? argl))
(real? (car argl)))
(set! level (car argl)))
;; Given the coordinate maps (x_0, ..., x_n), the boundary in the image of
;; the chart is the set {x_i = 0}.
(let ((coord-map (chart:get-coord-map chart))

```
```

            (in-domain? (chart:get-membership-test chart))
            (in-range? (chart:get-range-test chart)))
        (letrec
            ((range-boundary?
                (lambda (x)
                (and (in-range? x)
                    (almost-equal? level (vector-ref x i))))
            (domain-boundary?
                (lambda (p)
                (and (in-domain? p)
                    (range-boundary? (coord-map p))))))
            (chart:install-extra chart
                                    'boundary-structs
                                    (vector i level domain-boundary? range-boundary?))
    chart))))
(define (chart:get-boundary-structs chart)
(chart:get-extra chart 'boundary-structs))
(define (boundary-chart? chart)
(if (chart:get-boundary-structs chart)
\#t
\#f))
(define (chart:get-boundary-index chart)
(let ((result (chart:get-boundary-structs chart)))
(if result
(vector-ref result 0)
\#f)))
(define (chart:get-boundary-level chart)
(let ((result (chart:get-boundary-structs chart)))
(if result
(vector-ref result 1)
\#f))
(define (chart:get-domain-boundary-test chart)
(let ((result (chart:get-boundary-structs chart)))
(if result
(vector-ref result 2)
\#f)))
(define (chart:get-range-boundary-test chart)
(let ((result (chart:get-boundary-structs chart)))
(if result
(vector-ref result 3)
\#f)))
(define (chart:domain-boundary? p chart)
(let ((boundary? (chart:get-domain-boundary-test chart)))
(if boundary?
(boundary? p)
\#f)))
(define (chart:range-boundary? x chart)
(let ((boundary? (chart:get-range-boundary-test chart)))
(if boundary?

```
(boundary? x) \#f)))
```

;;; Make a chart for the boundary out of a chart-with-boundary:
(define (make-boundary-chart chart)
(let ((boundary-chart (chart:get-extra chart 'boundary-chart)))
(if boundary-chart
boundary-chart
(make-new-boundary-chart chart))))
(define (make-new-boundary-chart chart)
(let ((in-domain? (chart:get-domain-boundary-test chart))
(in-range? (chart:get-domain-boundary-test chart))
(index (chart:get-boundary-index chart))
(level (chart:get-boundary-level chart))
(dim (chart:dimension chart)))
(if (and in-domain? in-range? index level)
(let ((coord-map (chart:get-coord-map chart))
(inverse-map (chart:get-inverse-map chart))
(project (lambda (x)
(vector:drop-coord x index)))
(immerse (lambda (x)
(let ((y (vector:add-coord x index)))
(vector-set! y index level)
y))))
(let ((new-coord-map (compose project coord-map))
(new-inverse-map (compose inverse-map immerse))
(transition
(lambda (Bother)
(let ((other (chart:whose-boundary? Bother)))
(compose
(lambda (x)
(vector:drop-coord
x (chart:get-boundary-index other)))
(chart:make-transition-map chart other)
immerse)))))
(let ((boundary-chart
(make-chart (- dim 1) in-domain? in-range?
new-coord-map new-inverse-map transition)))
(chart:install-extra chart 'boundary-chart boundary-chart)
(chart:install-extra
boundary-chart 'whose-boundary? (delay chart))
boundary-chart)))
\#f)))
(define (chart:whose-boundary? chart)
(force (chart:get-extra chart 'whose-boundary?)))
;;; Now a manifold with boundary (this may end up being the empty set):

```
```

(define (make-boundary-manifold M)
(let ((charts (manifold:get-finite-atlas M)))
(if charts
(let loop ((charts charts) (result '()))
(if (null? charts)
(if (null? result)
\#f
(charts->manifold result))
(let ((boundary-chart (make-boundary-chart (car charts))))
(if boundary-chart
(loop (cdr charts) (cons boundary-chart result))
(loop (cdr charts) result)))))
(let ((find-chart-in-M (manifold:get-general-chart-finder M))
(minimize-in-M (manifold:get-general-chart-finder M)))
(letrec
((general-find-chart
(lambda (p . predicates)
(call-with-current-continuation
(lambda (return)
(find-chart-in-M
P
(lambda (chart)
(if (chart:domain-boundary? p chart)
(let ((new-chart (make-boundary-chart chart)))
(let valid? ((predicates predicates))
(if (null? predicates)
(return new-chart)
(if ((car predicates) new-chart)
(valid? (cdr predicates))
\#f))))
\#f))()))(
(find-minimizing-chart
(lambda (p f <)
(cadr (minimize-in-M
p
(lambda (chart)
(if (chart:domain-boundary? p chart)
(let ((new-chart (make-boundary-chart chart)))
(list new-chart (f new-chart)))
\#f))
(lambda (x y)
(or (and x y (< (cadr x) (cadr y)))
(and x (false? y))))))))
(local-atlas-finder
(lambda (p)
(map (lambda (chart) (make-boundary-chart chart))
(manifold:get-local-atlas M p)))))
(make-manifold (- (manifold:dimension M) 1)
general-find-chart
find-minimizing-chart
local-atlas-finder))))))

```

\section*{C.1.5 bug.sem}
```

(load "load-ode")
(define result
(shou-time
(lambda ()
(rigid-body-path singular-init 1.))))
(define e-list
(show-time
(lambda ()
(map (compose vector-first (make-rigid-body-energy 1. (sqrt 2) 2.) cadr)
result))))
(define e-errors
(let ((ref (car e-list)))
(show-time
(lambda ()
(map (lambda (val) (relative-error val ref)) e-list)))))

```

\section*{C.1.6 charts.scm}
;;; Charts:
(declare (usual-integrations))
;; Abstract charts need only contain the right maps. What they actually do is
;; ; up to the particular implementation.
; ; Might have been nice to make charts out of smooth maps, but that might be
;; more trouble than it's worth. It's too recursive, and the abstraction has
;; ; to bottom out somewhere. (Why? Because otherwise it wouldn't be
;; ; computable!) Charts will be made out of structures much like smooth
; ; functions. We'll try to merge these structures if it appears possible.
(define (make-chart dim in-domain? in-range? coord-map inverse-map transition)
    ; ; TRANSITION should be a function that, given another chart, returns a
    ; ; transition function to the other chart from this one. (Within reasonable
    ; ; ranges, of course.)
    (vector in-domain? in-range? coord-map inverse-map transition dim '()))
(define (make-simple-chart dim in-domain? in-range? coord-map inverse-map)
    (make-chart dim in-domain? in-range? coord-map inverse-map
                        (lambda (V)
                            (compose (chart:get-coord-map V) inverse-map))))
;;; Get the various maps out:
(define (chart:get-membership-test chart)
    ; Should return a function that tests whether a point is in the chart.
    (vector-ref chart 0 ))
```

(define (chart:get-range-test chart)
(vector-ref chart 1))
(define (chart:get-coord-map chart)
;; Should provide the mapping from the manifold to Euclidean coordinates.
(vector-ref chart 2))
(define (chart:get-inverse-map chart)
;; Should provide the inverse of the above.
(vector-ref chart 3))
(define (chart:get-transition-maker chart)
(vector-ref chart 4))
(define (chart:dimension chart)
(vector-ref chart 5))
(define (chart:install-extra chart tag datum)
(let ((result (assq tag (vector-ref chart 6))))
(if result
(set-cdr! result datum)
(vector-set! chart 6 (cons (cons tag datum) (vector-ref chart 6))))))
(define (chart:get-extra chart tag)
(let ((result (assq tag (vector-ref chart 6))))
(if result
(cdr result)
\#f)))
;;; Some useful wrappers for debugging purposes:
(define (domain-check f chart)
(lambda (p)
(if (not (chart:member? p chart))
(write-line '(warning! stepping out of domain!)))
(f p)))
(define (range-check g chart)
(lambda (x)
(if (not (chart:in-range? x chart))
(write-line '(warning! stepping out of range!)))
(g x)))
;;; Some methods that are bound to be handy:
(define (chart:member? x chart)
((chart:get-membership-test chart) x))
(define (chart:in-range? x chart)
((chart:get-range-test chart) x))
(define (chart:point->coords x chart)
((chart:get-coord-map chart) x))
(define (chart:coords->point x chart)
((chart:get-inverse-map chart) x))

```
```

(define (chart:make-transition-map U V)
((chart:get-transition-maker U) V))
;;; Turn the chart maps into smooth maps:
(define (chart:get-range U)
(make-euclidean-space (chart:dimension U)))
(define (chart:get-domain chart)
(let ((U (chart:get-extra chart 'chart-as-manifold)))
(if U
(force U)
(let ((U (charts->manifold (list chart))))
(chart:install-extra chart 'chart-as-manifold (delay U))
U))))
(define (chart:smooth-coord-map chart)
(make-smooth-map (chart:get-domain chart)
(chart:get-range chart)
(chart:get-coord-map chart)
(lambda (U V) (chart:get-coord-map chart))))
(define (chart:smooth-inverse-map chart)
(make-smooth-map (chart:get-range chart)
(chart:get-domain chart)
(chart:get-inverse-map chart)
(lambda (U V) (chart:get-inverse-map chart))))
;;; A faster distortion test to compute for the ODE integrator:
(define (chart:stable-coords? x chart)
(chart:in-range?
(chart:point->coords (chart:coords->point x chart) chart)
chart))

```

\section*{C.1.7 cotangent.scm}
; ; This file defines cotangent bundles as vector bundles (see vbundle.scm).
(declare (usual-integrations))
```

;;; Make some covectors:
(define (make-cotangent chart x v)
;; x is a point in the abstract manifold, and chart contains x. v is an
;; element of the dual tangent space at x, represented in the chart as an
;; n-vector.
(vector 'cotangent chart x v))
(define (cotangent? x)
(and (vector? x)
(> (vector-length x) 0)

```
```

        (eq? 'cotangent (vector-ref x 0))))
    (define (cotangent:get-chart v)
(vector-ref v 1))
(define (cotangent:get-anchor v)
(vector-ref v 2))
(define (cotangent:get-coords v)
(vector-ref v 3))
(define (cotangent:dimension v)
(vector-length (cotangent:get-coords v)))
(define (make-binary-cotangent-operation op)
(lambda (v w)
(let ((p (cotangent:get-anchor v))
(q (cotangent:get-anchor w)))
(if (equal? p q)
(let ((chart (cotangent:get-chart v)))
(make-tangent chart
p
(op (cotangent:get-coords v)
(chart:pull-back w chart))))
(error "Cannot add covectors tangent to different points.")))))
(define cotangent+ (make-binary-cotangent-operation vector:+))
(define cotangent- (make-binary-cotangent-operation vector:-))
(define (cotangent* a v)
(make-tangent (cotangent:get-chart v)
(cotangent:get-anchor v)
(vector:* a (cotangent:get-coords v))))
(define (cotangent:act ctv tv)
(let ((chart (cotangent:get-chart ctv)))
(vector:dot (cotangent:get-coords ctv) (chart:push-forward tv chart))))
;;; Pull back a covector along a chart:
(define (chart:pull-back ctv chart)
(let ((orig (cotangent:get-chart ctv))
(v (cotangent:get-coords ctv)))
(if (eq? chart orig)
v
(pull-back-in-coords
(chart:make-transition-map chart orig)
(chart:point->coords (cotangent:get-anchor ctv) chart)
v))))
;;; Pull back v from T*f(x) to T*x:
(define (pull-back-in-coords f x v)
(let ((n (vector-length x)))
(let ((w (make-vector n))
(df ((diff f) x)))

```
```

        (do ((i 0 (+ i 1)))
                ((>= i n) w)
            (vector-set! w i (vector:dot v (df (vector:basis n i 1))))))))
    ;;; Cotangent charts:
(define (make-cotangent-chart chart)
(let ((new-chart (chart:get-extra chart 'cotangent-chart)))
(if new-chart
(force new-chart)
(make-new-cotangent-chart chart))))
(define (make-new-cotangent-chart chart)
(let* ((dim (chart:dimension chart))
(2*dim (* 2 dim))
(in-M-domain? (chart:get-membership-test chart))
(in-M-range? (chart:get-range-test chart))
(M-map (chart:get-coord-map chart))
(M-inverse (chart:get-inverse-map chart))
(dim-vector? (make-euclidean-test dim))
(2*dim-vector? (make-euclidean-test 2*dim)))
(letrec
((in-domain?
(lambda (v)
(and (in-M-domain? (cotangent:get-anchor v))
(dim-vector? (cotangent:get-coords v)))))
(in-range?
(lambda (v)
(and (2*dim-vector? v)
(in-M-range? (vector-head v dim))))
(coord-map
(lambda (v)
(vector-append (M-map (cotangent:get-anchor v))
(chart:pull-back v chart))))
(inverse-map
(lambda (x)
(make-cotangent chart
(M-inverse (vector-head x dim))
(vector-end x dim)))
(transition
(lambda (Tother)
(let ((other (chart:get-base-chart Tother)))
(let ((f (chart:make-transition-map chart other))
(g (chart:make-transition-map other chart)))
(lambda (x)
(let ((anchor (f (vector-head x dim)))
(cotangent (vector-end x dim)))
(vector-append
anchor
(pull-back-in-coords g anchor cotangent)))))))))

```
```

        (let ((new-chart (make-chart 2*dim in-domain? in-range?
                        coord-map inverse-map transition)))
    (chart:install-extra new-chart 'base-chart (delay chart))
    (chart:install-extra chart 'cotangent-chart (delay new-chart))
    new-chart))))
    (define (chart:get-base-chart chart)
(let ((result (chart:get-extra chart 'base-chart)))
(if result
(force result)
\#f)))
;;; Here's how we make a cotangent bundle:
(define (make-cotangent-bundle M)
(let ((T*M (manifold:get-extra M 'cotangent-bundle)))
(if T*M
(force T*M)
(make-new-cotangent-bundle M))))
(define (make-new-cotangent-bundle M)
(let ((dim-M (manifold:dimension M)))
(let (CE
(let ((charts (manifold:get-finite-atlas M)))
(if charts
(charts->manifold (map (lambda (chart)
(make-cotangent-chart chart))
charts))
(let ((find-chart-in-M (manifold:get-general-chart-finder M))
(minimize-in-M (manifold:get-general-minimizer M)))
(letrec
((general-find-chart
(lambda (p . predicates)
(call-with-current-continuation
(lambda (return)
(find-chart-in-M
(cotangent:get-anchor p)
(lambda (chart)
(let ((new-chart
(make-cotangent-chart chart)))
(let valid? ((predicates predicates))
(if (null? predicates)
(return new-chart)
(if ((car predicates) new-chart)
(valid? (cdr predicates))
\#f)))))))))(
(find-minimizing-chart
(lambda (p f <)
(cadr (minimize-in-M
(cotangent:get-anchor p)
(lambda (chart)
(let ((new-chart
(make-cotangent-chart chart)))

```
```

                    (list new-chart (f new-chart))))
                        (lambda (x y)
                            (< (cadr x) (cadr y)))))))
                            (local-atlas-finder
        (lambda (p)
            (map (lambda (chart) (make-cotangent-chart chart))
                        (manifold:get-local-atlas
                        M (cotangent:get-anchor p))))))
    (make-manifold (* 2 dim-M)
general-find-chart
find-minimizing-chart
local-atlas-finder))))))
(proj cotangent:get-anchor)
(fiber
(lambda (p)
(make-fiber cotangent+ cotangent- cotangent*
(lambda (v)
(equal? p (cotangent:get-anchor v)))))))
(let ((T*M (make-vector-bundle M E proj fiber)))
(manifold:install-extra M 'cotangent-bundle (delay T*M))
T*M))))

```

\section*{C.1.8 fields.scm}
```

;;; This file defines some structures related to vector fields:
; ; ; Here's a trivial vector field on the circle:
(define (circle-field p)
(let ((x (vector-ref p 0))
(y (vector-ref $p$ 1)))
(imbedding->tangent circle $p$ (vector (-y) $x$ ))))
;;; Here is the nonlinear pendulum. This shouldn't be *that* hard to define, ; ; but it is. Why? What should we change about the system?
; ; ; Related to the problem of defining vector fields is the issue of
; ; ; efficiency. The definition problem can be solved by making more charts,
; ; ; but efficiency would suffer even more. How do we fix that?
(define (make-pendulum $g$ mass length)
(let* ( $k$ (/g length))
(-k (* -1 k) )
(chart-1 (make-spherical-chart 1 '(0 1) 0))
(chart-2 (make-spherical-chart 1 '(1 0) 0)) (find-chart (make-tangent-chart-finder (lambda (x)
(if (chart:member? $x$ chart-1)
chart-1
(hart-2)))))

```
(lambda (p)
```

;; p should be a point from the cylinder constructed above.
(let* ((x (tangent:get-anchor p))
(chart (if (chart:member? x chart-1) chart-1 chart-2)))
(let ((xdot (chart:push-forward p chart)))
(make-tangent (find-chart p)
p
(vector-append
xdot
(vector (* (if (chart:member? x chart-1) -k k)
(vector-ref x 1))))))))))

```
; ; This is useful for checking how well the integrator is doing:
(define (make-pendulum-energy-function g mass length)
    (lambda (p)
        (let ( \(x\) (tangent:get-anchor \(p\) ))
            (v (tangent->imbedded-velocity circle p)))
            (- (/ (* mass (vector:magnitude~2 v)) 2)
                    (* mass g length (vector-ref \(x\) 0)))))
;;; Spherical pendulum:
(define make-spherical-pendulum
    (let* ((C1 (make-tangent-chart (make-spherical-chart 2 '(2 0 1) 0)))
            (C2 (make-tangent-chart (make-spherical-chart 2 ' \(\left(\begin{array}{lll}1 & 0 & 2)\end{array}\right)\) pi)))
            (TS^2 (charts->manifold (list C1 C2))))
        (lambda (g mass length)
            (let* ( \((\mathrm{k}\) (/g length))
                (-k (-k)))
            (lambda (p)
                    (let* ((chart (manifold:find-best-chart TS~2 p))
                        (x (chart:point->coords p chart))
                        (phi (vector-ref x 0 ))
                                    (theta (vector-ref \(\times 1\) ))
                                    (phidot (vector-ref \(\times 2\) ))
                                    (thetadot (vector-ref x 3)))
                (make-tangent chart
                                    p
                            ; ; Ended up deriving these things using Lagrangian
                            ;; mechanics anyway; might as well automate it.
                    (if (eq? chart C1)
                        (vector phidot
                        thetadot
                            (* (+ 1 (* (cos phi) (square thetadot)))
                                    (sin phi))
                            (* -2 (cot phi) phidot thetadot))
                                (vector phidot
                                    thetadot
                                    (* (+ (* (sin phi) (square thetadot))
                                    (sin theta))
```

                    (cos phi))
    (+ (* -2 (cot phi) phidot thetadot)
(/ (cos theta) (sin phi)))))))))))

```
```

;;; And in phase space:
(define make-spherical-H-pendulum
(let* ((C1 (make-cotangent-chart (make-spherical-chart 2 '(2 0 1) 0)))
(C2 (make-cotangent-chart (make-spherical-chart 2 '(1 0 2) pi)))
(T*S^2 (charts->manifold (list C1 C2))))
(lambda (g mass length)
(let ((k1 (/ (* mass (square length))))
(k2 (* mass g length)))
(lambda (p)
(let* ((chart (manifold:find-best-chart T*S^2 p))
(x (chart:point->coords p chart))
(phi (vector-ref x 0))
(theta (vector-ref x 1))
(p_phi (vector-ref x 2))
(p_theta (vector-ref x 3)))
(make-tangent chart
p
(if (eq? chart C1)
(vector (* k1 p_phi)
(* (/ k1 (square (sin phi))) p_theta)
(+ (* k1 (square p_theta)
(/ (* (square (sin phi))
(tan phi))))
(* k2 (sin phi)))
0)
(vector (* k1 p_phi)
(* (/ k1 (square (sin phi))) p_theta)
(+ (* k1 (square p_theta)
(/ (* (square (sin phi))
(tan phi))))
(* k2 (cos phi) (sin theta)))
(* k2 (sin phi) (cos theta)))))))))))

```
;; An example of something defined using Lagrangian methods:
(define R~3 (make-euclidean-space 3))
(define TR^3 (make-tangent-bundle \(R^{\wedge} 3\) ))
(define (make-free-particle-lagrangian m)
    (let ( \(\mathrm{m} / 2\) (/m 2)))
        (make-real-map TR^3
                            (lambda (p)
                            (* m/2 (vector:magnitude^2 (tangent:get-coords p))))))
;; An example of something defined using Hamiltonian methods:
(define \(T * R^{\wedge} 3\) (make-cotangent-bundle \(R^{\wedge} 3\) ))
(define (make-free-particle-hamiltonian \(m\) )
    (let ( \(\mathrm{m} / 2\) (/m2)))
        (make-real-map T*R^3
                        (lambda (p)
```

;;; The Lagrangian for a rather simple potential:
(define TR~3 (make-tangent-bundle R^3))
(define falling-lagrangian
(make-real-map
TR-3 (lambda (p)
(- (* 1/2 (vector:magnitude^2 (tangent:get-coords p)))
(vector-third (tangent:get-anchor p))))))
;;; And the equivalent Hamiltonian:
(define T*R^3 (make-cotangent-bundle R^3))
(define falling-hamiltonian
(make-real-map
T*R^3 (lambda (p)
(+ (* 1/2 (vector:magnitude^2 (cotangent:get-coords p)))
(vector-third (cotangent:get-anchor p))))))
;;; And now for rigid bodies:
(define SO3 (make-rotational-group))
(define TSO3 (make-tangent-bundle SO3))
(define T*S03 (make-cotangent-bundle SO3))
;;; What is the easiest way to make a Lagrange top? Just write down the
;;; Lagrangian!
(define (antisymmetric-matrix->vector A)
(vector (matrix-ref A 2 1) (matrix-ref A 0 2) (matrix-ref A 1 0)))
(define (tangent->angular-velocity p)
(let* ((M (tangent:get-anchor p))
(chart (tangent:get-chart p))
(Mdot
(vector->matrix 3 3
(push-forward-in-coords
(compose matrix:flatten
(chart:get-inverse-map chart))
(chart:point->coords M chart)
(tangent:get-coords p)))))
(antisymmetric-matrix->vector (matrix:* Mdot (transpose M)))))
(define (make-free-rigid-body-angular-momentum A B C)
(make-simple-map
TS03
(make-euclidean-space 3)
(lambda (p)
(let* ((M (tangent:get-anchor p))
(w (apply-linear-transformation
(transpose M) (tangent->angular-velocity p)))
(w-prime (apply-1inear-transformation (transpose M) w)))
(vector (* A (vector-ref w-prime 0))
(* B (vector-ref w-prime 1))

```
```

(define (make-free-rigid-body-lagrangian A B C)
(make-real-map
TS03
(lambda (p)
(let* ((M (tangent:get-anchor p))
(w (apply-linear-transformation
(transpose M) (tangent->angular-velocity p)))
(w-prime (apply-linear-transformation (transpose M) w)))
(* 1/2
(+ (* A (square (vector-ref w-prime 0)))
(* B (square (vector-ref w-prime 1)))
(* C (square (vector-ref w-prime 2)))))))))
(define (free-rigid-body-field-maker a b c)
(let* ((charts (manifold:get-finite-atlas TSO3))
(fields (list rigid-field-0 rigid-field-1
rigid-field-2 rigid-field-3))
(charts\&fields (map cons charts fields)))
(lambda (chart)
(let ((field (assq chart charts\&fields)))
(if field
(lambda (x)
((cdr field) a b c (vector-head x 3) (vector-tail x 3)))
(error "Unknown chart! -- FREE-RIGID-BODY"))))))
(define (scmutils-rigid-body-field-maker a b c)
(let* ((charts (manifold:get-finite-atlas TSO3))
(fields (map (make-sysder a b c)
(list t-rigid-body-0
t-rigid-body-1
t-rigid-body-2
t-rigid-body-3)))
(charts\&fields (map cons charts fields)))
(lambda (chart)
(let ((field (assq chart charts\&fields)))
(if field
(cdr field)
(error "Unknown chart! -- FREE-RIGID-BODY"))))))
(define (make-sysder a b c)
(lambda (t-rigid-body)
(show-time
(lambda ()
(let* ((sysder (lambda (a b c)
(lagrangian->state-derivative
(t-rigid-body a b c))))
(compiled (compile-sysder 3 sysder))
(field (compiled a b c)))
(lambda (x)
(state->manifold (field (manifold->state x)))))))))
(define (make-rigid-body-energy a b c)
(let* ((charts (cons euler-angles (manifold:get-finite-atlas SO3)))
(energies (list (t-rigid-body a b c)
(t-rigid-body-0 a b c)
(t-rigid-body-1 a b c)
(t-rigid-body-2 a b c)

```
```

                                    (t-rigid-body-3 a b c)))
        (charts&energies (map cons charts energies)))
    (lambda (tangent)
    (let* ((chart (tangent:get-chart tangent))
                (energy (assq chart charts&energies)))
            (if energy
                ((cdr energy)
                (manifold->state
                (vector-append
                    (chart:point->coords (tangent:get-anchor tangent) chart)
                    (tangent:get-coords tangent))))
            (error "Unknown chart! -- FREE-RIGID-BODY"))))))
    (define (make-rigid-body-momentum a b c)
;; This version is more accurate than MAKE-RIGID-BODY-ANGULAR-MOMENTUM.
(let* ((charts (cons euler-angles (manifold:get-finite-atlas SO3)))
(momenta (list (state->L-space a b c)
(state->L-space-0 a b c)
(state->L-space-1 a b c)
(state->L-space-2 a b c)
(state->L-space-3 a b c)))
(charts\&momenta (map cons charts momenta)))
(lambda (tangent)
(let* ((chart (tangent:get-chart tangent))
(momentum (assq chart charts\&momenta)))
(if momentum
((cdr momentum)
(manifold->state
(vector-append
(chart:point->coords (tangent:get-anchor tangent) chart)
(tangent:get-coords tangent))))
(error "Unknown chart! -- FREE-RIGID-BODY"))))))
(define (manifold->state x)
(let ((psi (vector-ref x 0))
(theta (vector-ref x 1))
(phi (vector-ref x 2))
(psidot (vector-ref x 3))
(thetadot (vector-ref x 4))
(phidot (vector-ref x 5)))
(->state O (vector theta phi psi) (vector thetadot phidot psidot))))
(define (state->manifold state)
(let ((q (state->q state))
(qdot (state->qdot state))
(t (state->t state)))
(let ((theta (vector-ref q 0))
(phi (vector-ref q 1))
(psi (vector-ref q 2))
(thetadot (vector-ref qdot 0))
(phidot (vector-ref qdot 1))
(psidot (vector-ref qdot 2)))
(vector psi theta phi psidot thetadot phidot))))

```

\section*{C.1.9 hamilton.scm}
```

;;; This file implements Hamiltonian mechanics. Given a sufficiently efficient
;;; implementation of cotangent bundles, this should run faster than using
;;; Lagrangians.
(declare (usual-integrations))
;;; The Hamiltonian should be a smooth map from the cotangent bundle of some
;;; manifold into the real line.
(define (hamiltonian->v.field H)
(let ((T*M (smooth-map:get-domain H))
(R (smooth-map:get-range H)))
(lambda (p)
(let ((U (manifold:find-best-chart T*M p)))
(make-tangent U p
(hamilton-in-coords
(smooth-map:make-transition
H U (car (manifold:get-finite-atlas R)))
(chart:point->coords p U)))))))
;;; Derive Hamilton's equations for f at x (in coordinates):
(define (hamilton-in-coords f x)
(let* ((2n (vector-length x))
(v (make-vector 2n))
(n (/ 2n 2)))
(do ((i n (+ i 1))
(j O (+ j 1)))
((>= j n) v)
(vector-set! v i (- (vector-first (((pdiff j) f) x))))
(vector-set! v j (vector-first (((pdiff i) f) x))))))

```

\section*{C.1.10 imbedding.scm}
```

;;; This file defines some procedures that are useful for working with
;;; imbedding representations of manifolds:
(declare (usual-integrations))
;;; For tangent vectors:
(define (imbedding->tangent M p v)
(let ((U (manifold:find-best-chart M p)))
(make-tangent U p (push-forward-in-coords (chart:get-coord-map U) p v))))
(define (tangent->imbedding v)
(let* ((U (tangent:get-chart v))
(p (tangent:get-anchor v)))
Clist p
(push-forward-in-coords (chart:get-inverse-map U)

```
```

(define tangent->imbedded-velocity
(compose cadr tangent->imbedding))
;;; For cotangent vectors:
(define (imbedding->cotangent M p v)
(let ((U (manifold:find-best-chart M p)))
(make-cotangent U p (pull-back-in-coords (chart:get-inverse-map U)
(chart:point->coords P U)
v))))
(define (cotangent->imbedding v)
(let* ((U (cotangent:get-chart v))
(p (cotangent:get-anchor v)))
(list p
;; Need to project the pulled-back functional onto the imbedded
;; surface because it's represented in the standard basis of the
;; ambient space, not the basis of the tangent space to M imbedded
;; inside R^n.
(project-onto-basis
(make-imbedded-basis U p)
(pull-back-in-coords (chart:get-coord-map U)
p
(cotangent:get-coords v))))))
(define (make-imbedded-basis chart x)
(let ((dim (chart:dimension chart)))
(let loop ((i 0) (vlist '()))
(if (< i dim)
(loop (+ i 1) (cons (tangent->imbedded-velocity
(make-tangent chart x (vector:basis dim i 1)))
vlist))
vlist))))

```

\section*{C.1.11 job-ode.scm}
```

(load "load-ode")

```
(show-time
    (lambda ()
        (let ( (segment-size 1.)
            (count 101)
            (filename "rigid-man.data")
            (charts (manifold:get-finite-atlas TSO3)))
            (let loop ( i 0 ) ( \(x\) singular-init) (t 0.\()\) )
            (if (< i count)
                (begin
                    (write-line '(step, it \(\mathrm{t}=\mathrm{t})\) )
```

(let ((results
(show-time
(lambda ()
(rigid-body-path x (+ t segment-size) t)))))
(let ((port (open-output-file filename \#t)))
(for-each
(lambda (l)
(let ((t (car l))
(p (cadr l)))
(display t port)
(let loop ((i 0) (charts charts))
(if (null? charts)
(display "No chart!" port)
(let ((chart (car charts)))
(if (chart:member? p chart)
(begin
(display " " port)
(display i port)
(display " " port)
(display (chart:point->coords p chart)
port))
(loop (+ i 1) (cdr charts))))))
(newline port)))
(let ((l (sort results (lambda (x y) (< (car x) (car y))))))
(if (> t 0)
(cdr 1)
1)))
(close-output-port port))
(loop (+ i 1) (cadar results) (caar results)))))))))

```

\section*{C.1.12 job-pde.scm}
```

(load "load-pde")
;;; Construct a domain:
(define make-test-domain
(pde:make-domain-without-overlaps
disc
make-vertices
make-no-extra-nodes
planar-triangulate
'(rectangular 20 10)
'(spherical 10 20)))

```
```

;;; Construct the elements:
(make-test-domain
imbedded-poly-laplacian
make-triangular-imbedded-integrator
pde:make-imbedded-poly-basis-function)
;;; Make a matrix!
(define mat (combine-equations-without-overlap disc 0-function test-function))
;;; Print some stuff out to file:
(if \#f
(begin
(write-line '(writing matrix to file...))
(let ((port (open-output-file "mat")))
(print-matrix mat port)
(close-output-port port))))
(write-line '(done!))
(if \#f
(let ((port (open-output-file "err")))
(write-line '(max err = ,(max-error disc x-coord-map v)) port)
(write-line '(min err = ,(min-error disc x-coord-map v)) port)
(write-line '(avg err = ,(avg-error disc x-coord-map v)) port)
(newline port)
(write-line '(computed actual) port)
(for-each
(lambda (node)
(let ((index (node:get-id node)))
(if (number? index)
(write-line '(,(vector-ref v index)
,(x-coord-map node))
port))))
(sort (append-map (lambda (node)
(if (number? (node:get-id node))
(list node)
'())
(manifold:get-nodes disc))
(lambda (n1 n2)
(< (node:get-id n1) (node:get-id n2)))))
(close-output-port port)))

```

\section*{C.1.13 lagrange.scm}
;;; This file defines the structures necessary to support Lagrangian vector
;;; fields on configuration spaces of classical mechanical systems.
```

(declare (usual-integrations))
;;; The Lagrangian should be a smooth map from the tangent bundle of some
;;; manifold into the real line.
;;; This is very slow, as every evaluation of the field involves a matrix
;;; inversion. Which is why Hamiltonians are *better*, even for comuptational
;;; purposes!
(define (lagrangian->v.field L)
(let ((TM (smooth-map:get-domain L))
(R (smooth-map:get-range L)))
(lambda (p)
(let ((U
(if (tangent? p)
(make-tangent-chart (tangent:get-chart p))
(manifold:find-best-chart TM p))))
(let ((f (smooth-map:make-transition
L U (car (manifold:get-finite-atlas R))))
(x (chart:point->coords p U)))
(let ((v (vector-tail x (/ (vector-length x) 2))))
(let ((E-L (euler-lagrange-in-coords f x)))
(let ((A (car E-L))
(B (cadr E-L))
(c (caddr E-L)))
(let ((accel (matrix:solve-linear-system
A
(vector:+ (apply-linear-transformation B v) c))))
(make-tangent U p (vector-append v accel))))))))))

```
```

;;; Derive the Euler-Lagrange equations for f at x (in coordinates) in the form

```
;;; Derive the Euler-Lagrange equations for f at x (in coordinates) in the form
;;; A*xdotdot = B*xdot + c.
;;; A*xdotdot = B*xdot + c.
(define (euler-lagrange-in-coords f x)
(define (euler-lagrange-in-coords f x)
    (let* ((n (/ (vector-length x) 2))
    (let* ((n (/ (vector-length x) 2))
            (A (make-matrix n n))
            (A (make-matrix n n))
            (B (make-matrix n n))
            (B (make-matrix n n))
            (c (make-vector n 0)))
            (c (make-vector n 0)))
        (do ((in (+ i 1))
        (do ((in (+ i 1))
            (p O (+ p 1)))
            (p O (+ p 1)))
            ((>= p n))
            ((>= p n))
            ;; First, compute the hessian of f with respect to the velocity part of
            ;; First, compute the hessian of f with respect to the velocity part of
            ;; the independent variable:
            ;; the independent variable:
            (matrix-set! A p p (vector-first (((pdiff i) ((pdiff i) f)) x)))
            (matrix-set! A p p (vector-first (((pdiff i) ((pdiff i) f)) x)))
            (do ((j (+ i 1) (+ j 1))
            (do ((j (+ i 1) (+ j 1))
                    (q (+ p 1) (+ q 1)))
                    (q (+ p 1) (+ q 1)))
                    ((>= q n))
                    ((>= q n))
                    (let ((val (vector-first (((pdiff j) ((pdiff i) f)) x))))
                    (let ((val (vector-first (((pdiff j) ((pdiff i) f)) x))))
                        (matrix-set! A p q val)
                        (matrix-set! A p q val)
                (matrix-set! A q p val)))
                (matrix-set! A q p val)))
            ;; Next, compute the rest of the terms involving the partials of the
```

            ;; Next, compute the rest of the terms involving the partials of the
    ```
```

;; Lagrangian with respect to the positions (note the minus sign):
(do ((j 0 (+ j 1)))
((>= j n))
(let ((val (- (vector-first (((pdiff j) ((pdiff i) f)) x)))))
(matrix-set! B p j val)))
;; And then there's the term due to the derivative of the Lagrangian with
;; respect to the position variables:
(vector-set! c p (vector-first (((pdiff p) f) x))))

```
(list A B c)) )
```

;;; In many mechanics problems, it's natural to check conservation laws:
(define (check-vector-conservation-law quantity ref-point)
(let ((ref (quantity ref-point)))
(lambda (chart tangent)
(vector:distance (quantity (tangent:get-anchor tangent)) ref))))

```

\section*{C.1.14 linear.scm}
;;; This file defines (again) some useful vector algebra procedures. And
;; ; SQUARE is *always* useful...
(declare (usual-integrations))
;;; Vector operations. For completeness, we provide procedures that operate on
;; ; objects of the same *shape* (made up of lists of lists of vectors, etc.),
;; ; which *may* be useful for working with product structures.
(define (generalize-binary-operation combine-vectors
                    combine-numbers
                    combine-structs
                        null
                        error-string)
    (let ( report-error
        (let ((string (string-append "Objects have different sizes or are of"
                        " the wrong type. -- "
                        error-string)))
        (lambda ()
            (error string))))
        (generalized-op
        (lambda (v w)
            (if (list? v)
                (if (list? w)
                    (if (null? v)
                        (if (null? w)
                        null
                        (report-error))
                            (if (null? w)
                                    (report-error)
                                    (combine-structs (generalized-op (car v) (car w))
```

                                    (generalized-op (cdr v) (cdr w)))))
                                    (report-error))
    (if (vector? v)
(if (vector? w)
(combine-vectors v w)
(report-error))
(if (number? v)
(if (number? w)
(combine-numbers v W)
(report-error))
(report-error)))))))
generalized-op))
;;; Vector addition:
(define (vector:binary+ v1 v2)
(if (= (vector-length v1) (vector-length v2))
(let* ((len (vector-length v1))
(v (make-vector len)))
(let loop ((i 0))
(if (< i len)
(begin
(vector-set! v i (+ (vector-ref v1 i) (vector-ref v2 i)))
(loop (+ i 1)))
v)))
(error "Cannot add vectors of different dimensions. -- VECTOR:+")))
(define (vector:+ v . vlist)
(let loop ((v v) (vlist vlist))
(if (null? vlist)
v
(loop (vector:binary+ v (car vlist)) (cdr vlist)))))
(define vector:general+
(generalize-binary-operation vector:binary+ + cons '() "VECTOR:+"))
;;; Vector subtraction:
(define (vector:binary- v1 v2)
(if (= (vector-length v1) (vector-length v2))
(let* ((len (vector-length v1))
(v (make-vector len)))
(let loop ((i 0))
(if (< i len)
(begin
(vector-set! v i (- (vector-ref v1 i) (vector-ref v2 i)))
(loop (+ i 1)))
v)))
(error "Cannot subtract vectors of different lengths. -- VECTOR:-")))
(define (vector:- v . vlist)
(if (null? vlist)
(vector:* -1 v)
(let loop ((v v) (vlist vlist))
(if (null? vlist)
v
(loop (vector:binary- v (car vlist)) (cdr vlist))))))

```
```

(define vector:general-
(generalize-binary-operation vector:binary- - cons '() "VECTOR:-"))
;;; Scalar multiplication:
(define (vector:* a v)
(let* ((len (vector-length v))
(r (make-vector len)))
(let loop ((i 0))
(if (< i len)
(begin
(vector-set! w i (* a (vector-ref v i)))
(loop (+ i 1)))
w))))
(define (vector:general* a v)
(if (list? v)
(if (null? v)
'()
(cons (vector:general* a (car v)) (vector:general* a (cdr v))))
(if (vector? v)
(vector:* a v)
(if (number? v)
(* a v)
(error "Object is not a vector! -- VECTOR:*")))))
;;; Standard euclidean structures:
(define (vector:dot v w)
(let ((len (vector-length v)))
(if (not (= len (vector-length w)))
(error "Vectors do not have the same dimension. -- VECTOR:DOT")
(let loop ((i 0) (sum 0.))
(if (< i len)
(loop (+ i 1) (+ sum (* (vector-ref v i) (vector-ref w i))))
sum)))))
(define vector:general-dot
(generalize-binary-operation vector:dot * + 0 "VECTOR:DOT"))
;;; Solve linear systems of equations:
(define (matrix:solve-linear-system A b)
(let* ((m (matrix-row-count A))
(n (matrix-column-count A))
(mat (make-matrix m (+ n 1))))
(do ((i O (+ i 1)))
((>= i m))
(do ((j O (+ j 1)))
((>= j n))
(matrix-set! mat i j (matrix-ref A i j))))
(do ((i O (+ i 1)))
((>= i m))
(matrix-set! mat i n (vector-ref b i)))

```
```

(rref mat)
(let ((result (make-vector n)))
(do ((j O (+ j 1)))
((>= j n))
(vector-set! result j (matrix-ref mat j n)))
result)))

```

\section*{C.1.15 load-main.scm}
```

;;; Figure out if ScmUtils is loaded (by checking if a few key procedures are
;;; defined):
(define *using-scmutils?*
(let ((procs '(derivative vector:scalar*vector)))
(and mit-scheme?
(not (memq \#f (map
(lambda (proc)
(environment-bound? (the-environment) proc))
procs))()))
(newline)
(display
(if *using-scmutils?*
"*** It looks like we're running ScmUtils..."
"*** ScmUtils doesn't seem to be running. Going numerical..."))
(newline)
(let ((preload )("misc"
"lshared"))
(core '("charts"
"manifold"
"vbundle"
"product"
"boundary"
"ranges"
"smooth"
"spaces"))
(numdiff '("misc-math"
"fem/matlib"
"linear"
"richardson"))
(scmutils '("stubs")))
(for-each load preload)
(for-each load (if *using-scmutils?* scmutils numdiff))
(for-each load core))

```

\section*{C.1.16 load-ode.scm}

\footnotetext{
; ; ; Load the PDE stuff separately (so that the parts of the program *not* under ;; ; development can still be used without these definitions).
}
```

(let (Code-core
'("ode"
"ode-fast"
"lagrange"
"hamilton"
"fields"
"rigid-fields"
"ode-examples"))
(ode-scmutils
'("rigid")))
(load "load-main")
(if (not *using-scmutils?*)
(begin
(newline)
(display "*** Warning: ODE code works better with ScmUtils!")
(newline))
(for-each load ode-scmutils))
(for-each load ode-core))

```

\section*{C.1.17 load-pde.scm}
;; ; Load the PDE stuff separately (so that the parts of the program *not* under
;; development can still be used without these definitions).
(let ( pde-core
    ' ("pde-aux"
        "pde-charts"
        "pde-cmpgrd"
        "pde-debug"
        "pde-elements"
        "pde-main"
        "pde-mergers"
        "pde-nodes"
        "pde-ops"
        "pde-tools"
        "pde-config"
        "basis-imb"
        "basis-poly"
        "basis-real"
        "pde-examples"))
    (fem-stuff
    '("basis"
        "2d-poly-basis"
        "2d-real-basis"
        "2d-trapezoid"
        "delaunay"
        "delaux"
        "dyntable"
        "edge"
```

        "fem"
        "relax"
        "sparse"
        "util-too"))
        (fem-dir "fem/"))
    (load "load-main")
    (if *using-scmutils?*
        (begin
            (newline)
            (display "*** Warning: PDE code does not work well with ScmUtils!")
            (newline)))
    (for-each (lambda (file)
(load (string-append fem-dir file)))
fem-stuff)
(for-each load pde-core))

```

\section*{C.1.18 lshared.scm}
```

(declare (usual-integrations sqrt))
;;; Some useful procedures for manipulating stereographic projections and such:
(define (vector:drop-coord v i)
;; Project v onto the orthogonal complement of the ith standard basis vector:
(let* ((n (vector-length v))
(n (make-vector (- n 1))))
(let loop ((j 0) (k 0))
(if (< j n)
(if (= j i)
(loop (+ j 1)k)
(begin
(vector-set! w k (vector-ref v j))
(loop (+ j 1) (+ k 1))))
w))))
(define (vector:add-coord v i)
;; Do the opposite:
(let* ((n (+ (vector-length v) 1))
(w (make-vector n)))
(let loop ((j 0) (k 0))
(if (< j n)
(if (= j i)
(begin
(vector-set! w j 0.)
(loop (+ j 1) k))
(begin
(vector-set! w j (vector-ref v k))
(loop (+ j 1) (+ k 1))))
w))))
(define (vector:basis dim i val)

```
```

    (let ((v (make-vector dim 0.)))
    (vector-set! v i val)
    v))
    ;;; Useful to return the last COUNT elements of a vector:
(define (vector-end v count)
(vector-tail v (- (vector-length v) count)))
;;; Based on vector:dot:
(define (vector:magnitude^2 v)
(vector:dot v v))
(define (vector:magnitude v)
(sqrt (vector:magnitude`2 v)))
(define (vector:distance 2 v w)
(vector:magnitude^2 (vector:- v w)))
(define (vector:distance v w)
(sqrt (vector:distance~2 v w)))
;;; Least-squares approximation:
(define (project-onto-basis vlist vector)
;; Just do least-squares...
(let* ((trans (list->vector vlist))
(basis (transpose trans))
;; Taking this transpose, of course, is where we implicitly use the
;; metric structure of the ambient Euclidean space.
(result (matrix:solve-linear-system (matrix:* trans basis)
(apply-linear-transformation
trans vector))))
(apply-linear-transformation basis result)))
;;; Make an identity matrix:
(define (make-identity-matrix n)
(let ((mat (make-matrix n n)))
(do ((i O (+ i 1)))
((>= i n) mat)
(matrix-set! mat i i 1))))
;;; Compute the square root of the trace of a matrix multiplied by its own
;;; transpose:
(define (matrix:magnitude A)
(let ((m (matrix-row-count A))
(n (matrix-column-count A)))

```
```

    (let next-row ((i 0) (sum 0))
        (if (< i m)
            (let next-col ((j 0) (sum sum))
                (if (< j n)
                    (next-col (+ j 1) (+ sum (square (matrix-ref A i j))))
                    (next-row (+ i 1) sum)))
            (sqrt sum)))))
    ;;; Find the maximum element of a matrix:
(define (matrix:max A)
(let ((m (matrix-row-count A))
(n (matrix-column-count A)))
(let next-row ((i 0) (max 0))
(if (< i m)
(let next-col ((j 0) (max max))
(if (< j n)
(let ((mag (magnitude (matrix-ref A i j))))
(if (> mag max)
(next-col (+ j 1) mag)
(next-col (+ j 1) max)))
(next-row (+ i 1) max)))
max))))
;;; Do a least-squares approximation:
(define (least-squares mat)
(let* ((m (matrix-row-count mat))
(n+1 (matrix-column-count mat))
(n (- n+1 1))
(out (make-matrix n n+1)))
(write-line '(preparing normal equations...))
(do ((i 0 (+ i 1)))
((>= i n))
(do ((j 0 (+ j 1)))
((> j n))
(let loop ((k 0) (sum 0))
(if (< k m)
(loop (+ k 1) (+ sum (* (matrix-ref mat k i)
(matrix-ref mat k j))))
(matrix-set! out i j sum)))))
(write-line '(solving normal equations using lu-decomposition...))
(lu-solve mat)))
;;; Here's a matrix deconstructor:
(define (matrix:flatten A)
(let* ((m (matrix-row-count A))
(n (matrix-column-count A))
(size (* m n))
(v (make-vector size 0)))

```
```

        (do ((i O (+ i 1)))
            ((>= i m))
        (do (( j 0 (+ j 1)))
                ((>= j n))
            (vector-set! v (+ (* i n) j) (matrix-ref A i j))))
    v))
    (define (vector->matrix m n v)
(let ((A (make-matrix m n)))
(do ((i 0 (+ i 1)))
((>= i m))
(do ((j O (+ j 1)))
((>= j n))
(matrix-set! A i j (vector-ref v (+ (* i n) j))))
A))

```

\section*{C.1.19 manifold.scm}

Some obsolete comments:
;;
;; ; How do we represent manifolds as computational objects? How can we perform
;;; geometric operations in a coordinate-free way? Can we efficiently encode
;; charts and mappings? How can we use the inverse function theorem (or the
; implicit function theorem) to automagically construct charts for a
manifold?
;;
;;; And how much set theory do we need to implement? This probably depends on
;; ; what we want to do with the system. How can we implement, for example, the
;;; axiom of choice?
;;;
;; ; Regarding the question of using the inverse function theorem: We can
;;; probably accomplish this by some kind of numerical differentiation and ;; using a first-order approximation of the function to define charts.
;; ;
;; Do this quickly and try out ideas. Don't waste time on completeness or
;;; generality.
;; \(;\) December 1996 (from Neal's and Holly's machines):
;;
; ; ; Notes on PDEs have been moved to pde.scm, while notes on ODEs are now in
;;; ode.scm.
(declare (usual-integrations))
;; We need to make some simple Euclidean spaces:
(define (make-new-euclidean-space dim)
; ; Just need one big happy chart:
(let* ((test (make-euclidean-test dim))
(chart (make-simple-chart dim test test identity identity)))
(charts->manifold (list chart))))
;; Will need this a lot in charts, so this will help speed it up a bit:
```

(define make-euclidean-space (simple-memoize make-new-euclidean-space 26))

```
;;; Manifolds:
; Again, abstract manifolds need only have the right access methods. This
allows potentially infinite atlases (if the required methods can be
computed efficiently). It's not even clear the atlases need to be
externally accessible.
To construct a manifold, we need a procedure FIND-CHART that looks up a
chart satisfying some given predicate. Most other procedures can be
constructed out of this, but it should be arranged so that these procedures
can be replaced, if necessary.
;;; I guess we need to require atlases to be *locally finite*. Most manifolds
;; ; we construct will be compact (or products of compact manifolds with
;;; Euclidean spaces), so it shouldn't matter anyway.
(define (package-manifold-maps dimension
                                    general-chart-finder
                                    general-chart-minimizer
                                    find-chart
                                    find-another-chart
                                    find-least-distorted
                                    get-local-atlas)
(vector dimension
; ; Find a chart containing a given point and satisfying a given list
; ; of predicates:
general-chart-finder
    ; Find a chart containing a given point and minimizing a given
    ; ; function (given an ordering on the function's output):
    general-chart-minimizer
    ; ; Find a chart containing a given point:
find-chart
    ; ; Find a chart containing a given point and not in a given list:
find-another-chart
; Find the least distorted chart containing the given tangent
; ; vector:
find-least-distorted
; Find a (finite) set of charts containing a given point. The
; ; procedure is allowed to return (), if \(p\) is not in the manifold.
; ; Note that everything else is, theoretically, implementable from
; ; this. However, this would be too slow (even for us).
get-local-atlas
; ; Extra junk:
'())
; ; ; Here's an easier way to make manifolds: GENERAL-FIND-CHART finds a chart
;;; satisfying a given predicate, and FIND-MINIMIZING-CHART finds a chart that
```

;;; minimizes a function that always returns either a real *or* \#f (\#f means
;;; the chart should be thrown out).
;;; As usual, if something cannot be found, \#f is returned.
(define (make-manifold dim
general-find-chart
find-minimizing-chart
get-local-atlas)
(letrec
((find-chart
(lambda (p)
(general-find-chart p)))
(find-another-chart
(lambda (p charts)
(general-find-chart
P
(lambda (chart)
(not (memq chart charts)))))
(find-least-distorted
(lambda (tangent)
(car (find-minimizing-chart
(tangent:get-anchor tangent)
(lambda (chart)
(local-distortion chart tangent))
<))))
(package-manifold-maps dim
general-find-chart
find-minimizing-chart
find-chart
find-another-chart
find-least-distorted
get-local-atlas)))
;;; Get the various methods:
(define (manifold:dimension M)
(vector-ref M 0))
(define (manifold:get-general-chart-finder M)
(vector-ref M 1))
(define (manifold:get-general-minimizer M)
(vector-ref M 2))
(define (manifold:get-chart-finder M)
;; Return a function that finds a chart containing a given point.
(vector-ref M 3))
(define (manifold:get-second-opinion M)
(vector-ref M 4))
(define (manifold:get-least-distorted M)
(vector-ref M 5))

```
```

(define (manifold:get-local-atlas M p)
((vector-ref M 6) p))
(define (manifold:install-extra M tag datum)
(let ((result (assq tag (vector-ref M 7))))
(if result
(set-cdr! result datum)
(vector-set! M 7 (cons (cons tag datum) (vector-ref M 7))))))
(define (manifold:get-extra M tag)
(let ((result (assq tag (vector-ref M 7))))
(if result
(cdr result)
\#f)))
(define (manifold:reset-extra! M)
(vector-set! M 7 '()))
;;; Some things that are bound to be handy:
(define (manifold:member? M x)
(if ((manifold:get-chart-finder M) x)
\#t
\#f))
(define (manifold:find-chart M x)
;; If the manifold has only one chart, always return it without checking.
;; This kludge make smooth functions on tangent bundles of Euclidean spaces
;; work with ScmUtils.
(let ((atlas (manifold:get-finite-atlas M)))
(if (and atlas (null? (cdr atlas)))
(car atlas)
((manifold:get-chart-finder M) x))))
(define (manifold:find-another-chart M x . charts)
((manifold:get-second-opinion M) x charts))
(define (manifold:find-least-distorted M tangent)
((manifold:get-least-distorted M) tangent))
(define manifold:find-best-chart
(if *using-scmutils?*
manifold:find-chart
(lambda (M x)
((manifold:get-least-distorted M)
(make-tangent (manifold:find-chart M x)
x
(make-vector (manifold:dimension M) 1))))))
;;; An easy way to construct a large class of manifolds:
(define (charts->manifold charts)
(if (null? charts)

```
```

    (error "No charts given. -- CHARTS->MANIFOLD"))
    (let ((dim (chart:dimension (car charts))))
(let loop ((charts (cdr charts)))
(if (not (null? charts))
(if (= dim (chart:dimension (car charts)))
(loop (cdr charts))
(error (string-append "Not all charts have the same dimension!"
" -- CHARTS->MANIFOLD"))))

```
    (letrec
        ( (general-chart-finder
            (lambda (p . predicates)
                    (let loop ((charts charts))
                        (if (null? charts)
                        \#f
                            (let ((chart (car charts)))
                                (if (chart:member? p chart)
                                (let valid? ((predicates predicates))
                                (if (null? predicates)
                        chart
                        (if ((car predicates) chart)
                            (valid? (cdr predicates))
                            (loop (cdr charts)))))
                            (loop (cdr charts)))))))
                (find-minimizing-chart
                (lambda ( \(p \mathrm{f}\) <)
                    (let loop ( (charts charts) (result \#f) (min \#f))
                        (if (null? charts)
                        (if result
                        (list result min)
                        \#f)
                            (let ((chart (car charts)))
                            (if (chart:member? p chart)
                            (let ((val (f chart)))
                                    (if result
                                    (if (< val min)
                                    (loop (cdr charts) chart val)
                                    (loop (cdr charts) result min))
                                    (loop (cdr charts) chart val)))
                                    (loop (cdr charts) result min))))))
            (get-local-atlas
                (lambda (p)
                (let loop ((charts charts) (result '()))
                        (if (null? charts)
                        result
                        (let ((chart (car charts)))
                            (if (chart:member? p chart)
                                    (loop (cdr charts) (cons chart result))
                                    (loop (cdr charts) result)))))))
        (let ( \(M\) (make-manifold dim
                        general-chart-finder
                        find-minimizing-chart
                        get-local-atlas)))
            (manifold:install-extra M 'finite-atlas charts)

\section*{M) ) )}
```

(define (manifold:get-finite-atlas M)
(manifold:get-extra M 'finite-atlas))
;; There are various tools for constructing new manifolds out of old ones,
;;; such as vector bundles and product manifolds. However, the constructors do
;;; not know about each other: A product of two vector bundles should be a
;;; vector bundle, etc. I guess if such structures are ever needed, we can
;;; create extra operations.
;;; Note that the product of two manifolds with boundary may have corners on
;;; its boundary, so it may not be a smooth manifold with boundary. For
;;; example, the product of the unit interval with itself has corners. (Is
;;; this a problem for manifolds with n > 1?)

```

\section*{C.1.20 misc-math.scm}
```

(declare (usual-integrations))
;;; Some definitions that are always useful:
(define (square z)
(* z z))
(define (compose f . rest)
(let loop ((f f) (l rest))
(if (null? 1)
f
(loop (binary-compose f (car l)) (cdr l)))))
(define (binary-compose f g)
(lambda (first . rest)
(f (apply g (cons first rest)))))
(define pi (* 4 (atan 1)))
(define -pi (- pi))
(define (identity x)
x)
(define (relative-error val ref)
(if (zero? ref)
val
(/ (- val ref) ref)))

```

\section*{C.1.21 misc.scm}
```

(declare (usual-integrations))

```
;; Sometimes useful (stolen from nscmutils):
(define (make-comparator tol)
```

    (lambda (a b)
    (< (magnitude (- a b))
        (* . 5 tol
                (+ (magnitude a) (magnitude b) 2.0)))))
    (define almost-equal? (make-comparator 1e-10))
(define (almost-zero? x)
(almost-equal? x 0))
;;; Useful in making product manifolds:
(define (all-pairs l1 12)
(let loop1 ((l1 l1) (result '()))
(if (null? 11)
result
(let ((obj (car 11)))
(let loop2 ((12 12) (result result))
(if (null? 12)
(loop1 (cdr 11) result)
(loop2 (cdr 12) (cons (list obj (car 12)) result))))))))
;;; How do you tell if an object is a vector in R^n?
(define (make-euclidean-test dim)
(lambda (v)
(and (vector? v)
(= (vector-length v) dim)
(let loop ((i 0))
(if (< i dim)
(and (real? (vector-ref v i))
(loop (+ i 1)))
\#t))))(
;;; Always useful to memoize things:
(define (simple-memoize proc size)
;; Memoize a function whose argument is a non-negative integer:
(let ((cache (make-vector size 'undefined)))
(lambda (n)
(if (>= n size)
(proc n)
(let ((val (vector-ref cache n)))
(if (eq? val 'undefined)
(let ((val (proc n)))
(vector-set! cache n val)
val)
val))))()

```
;;; Useful in our implementation of spherical coordinates:
(define (list-integers i)
    (let loop ((result ' ()) (i i))
        (if (< i 0)
            result
```

(loop (cons i result) (- i 1)))))

```
```

;;; Surprisingly enough, this is not in MIT Scheme:
(define (cot z)
(/ (tan z)))
;;; PARTIAL in ScmUtils doesn't do the right thing for functions of vector
;;; arguments, so we can't just use that and stub out the numerical equivalent:
(define (pdiff i)
(lambda (f)
(let ((df (diff f)))
(lambda (x)
(let ((v (make-vector (vector-length x) 0)))
(vector-set! v i 1)
((df x) v))))))

```

\section*{C.1.22 ode-examples.scm}
;; ; This file defines some examples of things we can do on manifolds. First, ;;; load some files that need to be compiled:
;;; Let's make a torus!
(define circle (make-sphere 1))
(define torus (product-manifold circle circle))
; ; ; Now we need the tangent bundle of the circle, or this wouldn't make very
;;; much sense...
(define TS^1 (make-tangent-bundle circle))
;; ; The tangent bundle of the circle is trivial, because the circle is a Lie
;;; group!
(define cylinder TS^1)
i; Here's something to integrate:
(define circle-path
    (v.field->flow circle
                                    (v.field->local-field-maker circle-field)
                                    (make-rk4-integrator (* 2 pi 1e-3))
                                    local-distortion))
(define (real-circ \(t\) )
    (vector \((\cos t)(\sin t)))\)
;; ; Try this:
; (vector:distance (cadar (circle-path (vector 10 ) (* 2 pi))) (vector 10 ))
```

;;; Make a nonlinear pendulum:
(define pend-field (make-pendulum 1 1 1))
(define pend-path
(v.field->flow cylinder
(v.field->local-field-maker pend-field)
(make-rk4-integrator 1e-3)
local-distortion))
(define pend-energy (make-pendulum-energy-function 1 1 1))
(define pend-init (imbedding->tangent circle (vector 0 1) (vector 0 0)))
;;; Try this:
;(define pend-results (pend-path pend-init 1))
;;; We should check things like the conservation of energy. Also, do it with
;;; other methods (such as the imbedding) so that we have something to compare.
;;; The spherical pendulum isn't much harder to make:
(define S~2 (make-sphere 2))
(define TS^2 (make-tangent-bundle S^2))
(define make-sphere-tangent
(let* ((chart (make-tangent-chart (make-spherical-chart 2 '(2 0 1) 0)))
(coords->point (chart:get-inverse-map chart)))
(lambda (latitude longitude dlat dlong)
(let ((p (coords->point (vector latitude longitude dlat dlong))))
;(write-line '(initial position: ,(tangent:get-anchor p)))
;(write-line '(initial velocity: ,(tangent->imbedded-velocity p)))
p))))
(define spherical-init (make-sphere-tangent (* 3/4 pi) 0 0 1))
(define spherical-field (make-spherical-pendulum 1 1 1))
(define spherical-path
(v.field->flow TS^2
(v.field->local-field-maker spherical-field)
(make-rk4-integrator 1e-3)
local-distortion))
;;; An example of something defined using Lagrangian methods:
(define free-L-field (lagrangian->v.field (make-free-particle-lagrangian 1)))
;;; An example of something defined using Hamiltonian methods:
(define free-H-field (hamiltonian->v.field (make-free-particle-hamiltonian 1)))
;;; Let's try the spherical pendulum again:
(define spherical-inclusion

```
```

    (smooth-map:diff (make-simple-map S^2 R^3 identity)))
    (define spherical-lagrangian
(smooth-map:compose falling-lagrangian spherical-inclusion))
(define spherical-L-field
;; Note that this works only in numerical mode -- Due to some structural
;; problems, ScmUtils won't do the right thing on smooth functions on tangent
;; bundles that are *not* derived from maps on the base space.
(lagrangian->v.field spherical-lagrangian))
(define spherical-L-init
(imbedding->tangent S^2 (vector 1 0 0) (vector 0 1 0)))
(define spherical-L-path
(v.field->flow TS`2
(v.field->local-field-maker spherical-L-field)
(make-rk4-integrator 1e-3)
local-distortion))
;;; And again...
(define T*S^2 (make-cotangent-bundle S^2))
(define spherical-inclusion*
(let* ((chart (car (manifold:get-finite-atlas R^3)))
(f (lambda (v)
(apply make-cotangent
(cons chart (cotangent->imbedding v))))))
(make-simple-map T*S^2 T*R^3 f)))
(define spherical-hamiltonian
(smooth-map:compose falling-hamiltonian spherical-inclusion*))
(define spherical-H-field
(if \#t
(make-spherical-H-pendulum 1 1 1)
(hamiltonian->v.field spherical-hamiltonian)))
(define spherical-H-init
(imbedding->cotangent S^2 (vector 1 0 0) (vector 0 1 .5)))
(define spherical-H-path
(v.field->flow T*S^2
(v.field->local-field-maker spherical-H-field)
(make-rk4-integrator 1e-3)
(check-vector-conservation-law
(smooth-map:get-point-function spherical-hamiltonian)
spherical-H-init)))
(define (spherical-H-angular-momentum cv)
(let ((p (cadr (cotangent->imbedding cv)))
(q (cotangent:get-anchor cv)))
(let ((px (vector-ref p 0))
(py (vector-ref p 1))
(x (vector-ref q 0))

```
```

            (y (vector-ref q 1)))
            (- (* x py) (* y px)))))
    (define (print-spherical-H-state pair port)
(let ((p (cadr (cotangent->imbedding (cadr pair))))
(q (cotangent:get-anchor (cadr pair))))
(let ((t (car pair))
(px (vector-ref p 0))
(py (vector-ref p 1))
(pz (vector-ref p 2))
(x (vector-ref q 0))
(y (vector-ref q 1))
(z (vector-ref q 2)))
(display t port)
(for-each (lambda (val)
(display " " port)
(display val port))
(list x y z px py pz))
(newline port))))
;;; Here's a test of rigid bodies and Euler angles.
(define euler-angles (make-euler-angles 0 1 (/ -pi 2) (/ pi 2)))
(define rigid-body-energy
(make-free-rigid-body-lagrangian 1. (sqrt 2) 2.))
(define rigid-body-momentum
(make-free-rigid-body-angular-momentum 1. (sqrt 2) 2.))
(define rigid-body-init
(make-tangent euler-angles
(chart:coords->point (vector 0 1 0) euler-angles)
(vector .1 .1 .1)))
(define singular-init
(chart:coords->point (vector 0. 1. 0. -. 01 -. 1-.01)
(make-tangent-chart euler-angles)))
(define bad-init
(chart:coords->point
\#(-1.309711193385365 .12149475001297212 1.1518832285401293
-.19763392062291368 .09649536172931211 .18861249508985967)
(list-ref (manifold:get-finite-atlas TSO3) 0)))
(define make-rigid-body-field
(free-rigid-body-field-maker 1 (sqrt 2) 2))
(define energy+momentum
(let ((E (smooth-map:get-point-function rigid-body-energy))
(L (smooth-map:get-point-function rigid-body-momentum)))
(lambda (p)
(vector-append (E p) (L p)))))
(define (correct->traditional-order v)
(let ((psi (vector-ref v 0))
(theta (vector-ref v 1))
(phi (vector-ref v 2))

```
```

            (psidot (vector-ref v 3))
            (thetadot (vector-ref v 4))
            (phidot (vector-ref v 5)))
        (vector 0. theta phi psi thetadot phidot psidot)))
    (define scmutils-energy+momentum
(let ((energy (make-rigid-body-energy 1. (sqrt 2) 2.))
(momentum (make-rigid-body-momentum 1. (sqrt 2) 2.)))
(lambda (state)
(let ((L (momentum state))
(E (energy state)))
(vector E (vector-first L) (vector-second L) (vector-third L))))))
(define rigid-body-path
(v.field->flow
TSO3
make-rigid-body-field
(if *using-scmutils?*
(begin
(set! *ode-integration-method* 'bulirsch-stoer)
(make-scmutils-integrator . }01\mathrm{ 1e-12))
(make-rk4-integrator .01))
(check-vector-conservation-law
(if *using-scmutils?*
scmutils-energy+momentum
energy+momentum)
singular-init)))

```

\section*{C.1.23 ode-fast.scm}
```

;;; This is like ode.scm, only the integrator has its distortion checks

```
;;; This is like ode.scm, only the integrator has its distortion checks
;;; disabled.
;;; disabled.
(declare (usual-integrations))
(declare (usual-integrations))
(define (fast-v.field->flow M make-local-field next-step)
(define (fast-v.field->flow M make-local-field next-step)
    (lambda (p t-final . aux)
    (lambda (p t-final . aux)
    ;; Reset the error-reporting mechanism:
    ;; Reset the error-reporting mechanism:
    (set! *point-of-failure* #f)
    (set! *point-of-failure* #f)
    ;; AUX lets the user specify an initial time (optional).
    ;; AUX lets the user specify an initial time (optional).
    (let next-point ((p p)
    (let next-point ((p p)
                (t (if (not (null? aux)) (car aux) 0.))
                (t (if (not (null? aux)) (car aux) 0.))
                (result '()))
                (result '()))
    (if (< t t-final)
    (if (< t t-final)
            ;; Find all the charts containing this point and try each of them:
            ;; Find all the charts containing this point and try each of them:
            (let ((charts (manifold:get-local-atlas M p)))
            (let ((charts (manifold:get-local-atlas M p)))
            (if (null? charts)
            (if (null? charts)
                (begin
                (begin
                (write-line '(failure after ,(length result) steps))
                (write-line '(failure after ,(length result) steps))
                (write-line '(failed at time = ,t seconds))
                (write-line '(failed at time = ,t seconds))
                (set! *point-of-failure* p)
```

                (set! *point-of-failure* p)
    ```
```

            (error (ode-integrator-error 2))))
    (let next-chart ((charts charts))
    (if (null? charts)
            ;; No more charts: Panic!
            (begin
                (write-line '(failure after ,(length result) steps))
                (write-line '(failed at time = ,t seconds))
                        (set! *point-of-failure* p)
                (error (ode-integrator-error 1)))
            ;; Take a step forward in the next chart:
            (let* ((chart (car charts))
                (not-in-range
                    (compose not (chart:get-range-test chart)))
                (v.field (make-local-field chart))
                (make-field (field-protector v.field)))
            (let ((new
                    ;; This hack provides an escape mechanism from the
                    ;; local integrator: Check if it tries to access the
                    ;; field at a point outside the current chart.
                    (call-with-current-continuation
                    (lambda (return)
                                    (next-step
                                    (chart:point->coords p chart)
                                    (make-field
                                    chart
                                    (list (list not-in-range return))
                                    '())
                                    (lambda () (return #f)))))))
            (if new
                (let ((x (integrator:get-new-x new))
                    (dt (integrator:get-dt new)))
                    (next-point (chart:coords->point x chart)
                    (+ t dt)
                            (cons (list t p) result)))
                    (next-chart (cdr charts))))))))
    result)))(

```

\section*{C.1.24 ode.scm}
; ; ; What do we do when a orbit makes a transition between charts? This depends ; ; ; on the representation of points; GJS used an explicit imbedding. Whatever ; ; ; the representation, we can use hashing to look up charts given a point (and ;;; this operation should be abstracted anyway).
; ; What about using imbeddings? What would this do for either ODEs (the paths ; ; ; should just stay on the manifold) or PDEs? This is not very general.
;;; Since we can only get global existence under limited circumstances (e.g. a ; ; smooth vector field is complete if the manifold is compact), we might as ; ; well assume that the underlying manifold (or configuration space, since
```

;;; local existence works for second-order equations as well) is compact, thus
;;; allowing us to assume that the manifold has a finite atlas. Given that, we
;;; can evolve the integral curve in all charts at the same time, and pick the
;;; "best" solution at each step. (Or even any solution at all.)
;;; Actually, this idea only requires a locally finite atlas.
(declare (usual-integrations))
;;; Solve ODEs with 4th-order Runge-Kutta:
;;; (Needs automatic step-size control.)
;;; Something to agree on:
(define integrator:package-result cons)
(define integrator:get-new-x cdr)
(define integrator:get-dt car)
;;; Local integrators are assumed to take, in order, the following things:
;;; A STATE vector, a vector field, and FAIL (which is a thunk that does
;;; something in the case of an error).
;;; This procedure turns real vector fields into constructors for local fields;
;;; this exists (mostly) for compatibility issues.
(define (v.field->local-field-maker v.field)
(lambda (chart)
(let ((f (chart:get-inverse-map chart)))
(lambda (x)
(chart:push-forward (v.field (f x)) chart))))
;;; RK4 on manifolds. Still needs QC.
(define (make-rk4-integrator dt)
(lambda (x v fail)
(let ((dt/2 (/ dt 2.))
(dt/6 (/ dt 6.)))
(let* ((F1 (v x))
(F2 (v (vector:+ x (vector:* dt/2 F1))))
(F3 (v (vector:+ x (vector:* dt/2 F2))))
(F4 (v (vector:+ x (vector:* dt F3)))))
(integrator:package-result
dt
(vector:+
(vector:* dt/6
(vector:+ F1
(vector:* 2. F2)
(vector:* 2. F3)
F4))
x))()))

```
;;; If ScmUtils is loaded, just use the integrator there (QCRK4 is the default,
```

;;; I believe):
(define (make-scmutils-integrator dt tol)
(lambda (x v fail)
(let* ((w) (lambda (x)
(vector-append (vector 1) (v (vector-tail x 1)))))
(result (ode-advancer w (vector-append (vector 0) x) dt tol)))
(integrator:package-result
(vector-ref result 0)
(vector-tail result 1)))))
;;; We now need ODE solvers. When dealing with ODEs (as opposed to PDEs),
;;; there is one particular problem we need to address: When do we switch
;;; charts, if integrating locally?
;;; Here's an integrator integrates one step at a time. It understands how and
;;; when to switch between charts.
;;; This procedure uses continuations to handles faults like stepping out of
;;; the chart on a intermediate step, or detecting really bad error conditions.
;;; This simplifies the local integrator. (Tail recursion is *cool*.)
;;; We should modify this continuation hack to allow QC-RK-4 to punt charts
;;; based on local error analysis.
;;; PRE-CHECKS and POST-CHECKS are predicate-continuation pairs that check for
;;; errors before and after the computation of the vector field.
(define (field-protector v.field)
(lambda (chart pre-checks post-checks)
(lambda (x)
(let loop ((l post-checks))
(if (null? 1)
(let ((v (v.field x)))
(let loop ((l post-checks))
(if (null? 1)
v
(let ((pred-cont (car l)))
(if ((car pred-cont) v)
((cadr pred-cont) v)
(loop (cdr l)))))))
(let ((pred-cont (car l)))
(if ((car pred-cont) x)
((cadr pred-cont) x)
(loop (cdr 1)))))))))
;;; As an optimization, it might make sense for vector fields to work directly
;;; with charts. That is, a vector field is a constructor that, given a chart,
;;; constructs a local vector field on the chart. This fits in more nicely
;;; with a Lagrangian or Hamiltonian description of mechanics.
;;; There are some problems here:
;;; If we switch charts *before* we step off, then how do we know which chart
;;; to switch to? The program could sit there switching charts forever unless
;;; we build some memory into this.

```
```

;;; On the other hand, if we switch charts *after* we step off, then the
;;; inverse mapping fails. So what we need is a "compact refinement" of a
;;; covering of charts...
;;; We can always simultaneously evolve the point in several charts. But is
;;; that too slow?
(define ode-integrator-error
(let ((errors
(vector "*** Warning: Cannot find a good chart. Using any chart..."
"Error: I'm stuck! (Out of charts!) -- V.FIELD->FLOW"
"Error: Is this point in the manifold at all?")))
(lambda (i)
(vector-ref errors i))))
(define *point-of-failure* \#f)
(define (integrator-failure-point)
*point-of-failure*)
;;; M is the manifold on which we integrate, MAKE-LOCAL-FIELD is a function
;;; that takes a chart and returns a (local) vector field on that chart,
;;; NEXT-STEP is the local integrator, and DISTORTION lets the user rank how
;;; undesirable a particular result is.
(define (v.field->flow M make-local-field next-step distortion)
(lambda (p t-final . aux)
;; Reset the error-reporting mechanism:
(set! *point-of-failure* \#f)
;; AUX lets the user specify an initial time (optional).
(let next-point ((p p)
(t (if (not (null? aux)) (car aux) 0.))
(result '()))
(if (<= t t-final)
;; Find all the charts containing this point and try each of them:
(let ((charts (manifold:get-local-atlas M p)))
;(write-line '(one more point!))
(let next-chart ((charts charts) (min \#f) (best \#f))
(if (null? charts)
;; If we have found a pretty good answer, use it for the next
;; step and save the previous time step. Otherwise panic and
;; use any chart we can find. If we can't even find a chart,
;; then the previous step was really bad, too, so the program
;; just dies.
(if best
(next-point (cadr best)
(+ t (car best))
(cons (list t p) result))

```
```

    (let ((chart (manifold:find-best-chart M p)))
    (if chart
                (let* ((v.field (make-local-field chart))
                        (make-field (field-protector v.field))
                        (new (next-step
                            (chart:point->coords p chart)
                            (make-field chart '() '())
                            (lambda () (return #f)))))
                (newline)
                (display (ode-integrator-error 0))
                (neuline)
                (next-point (chart:coords->point
                                    (integrator:get-new-x new) chart)
                                    (+ t (integrator:get-dt new))
                                    (cons (list t p) result)))
        (begin
            (write-line
                    '(failure after ,(length result) steps))
            (write-line '(failed at time = ,t seconds))
            (set! *point-of-failure* p)
            (error (ode-integrator-error 1))))))
    ;; Take a step forward in the next chart:
(let* ((chart (car charts))
(not-in-range
(compose not (chart:get-range-test chart)))
(v.field (make-local-field chart))
(make-field (field-protector v.field)))
;(write-line '(one more chart!))
(let ((new

```
            ; ; This hack provides an escape mechanism from the
            ;; local integrator: Check if it tries to access the
            ; ; field at a point outside the current chart.
            (call-with-current-continuation
            (lambda (return)
                (next-step
                (chart:point->coords p chart)
                (make-field
                chart
                (list (list not-in-range return))
                , ()
                (lambda () (return \#f)))))))
    ; ; If nothing went wrong, check if the current chart does
    ; ; better, and keep the result if it does. Otherwise
    ; ; keep the old results.
    (if new
        (let ((x (integrator:get-new-x new)))
            (if (chart:in-range? \(x\) chart)
                (let* ((q (chart:coords->point \(x\) chart))
                (dt (integrator:get-dt new))
                (e (distortion
```

    chart
    (make-tangent chart q
                                    (v.field x)))))
        ;(write-line '(t = ,t e = ,e))
        (if min
            (if (< e min)
                (next-chart (cdr charts)
                    e
                    (list dt q))
                            (next-chart (cdr charts) min best))
            (next-chart (cdr charts) e (list dt q))))
        (next-chart (cdr charts) min best)))
        (next-chart (cdr charts) min best)))))))
    ```
    result))) (
```

;;; Local integrators, on the other hand, are more general. But they have the
;;; following problems:
;;;
1. They are less efficient.
;;
;;; 2. When to switch charts?
;;;
;;; 3. Distortion of the vector field may produce locally bad solutions.
Some of these problems (namely 2) can be solved by requiring that manifolds
have charts ( }U,V,f)\mathrm{ where the closure of U and V are both compact, and f
extends to a diffeomorphism between those compact sets. This lets us
switch charts without having a measure of how badly the vector field's
doing. (Namely, switch when we step out of a chart!)
Using the imbedding to integrate: It's probably faster, in many cases more
intuitive, and avoids the issue of switching between charts. The problems
are:

1. It's not as useful in abstract manifolds. We don't know if all
manifolds important for applications will be represented by imbeddings).
;;
;;; 2. The trajectories may not stay on the manifold.
;;; Some of these problems can be resolved by having a uniform way to attach
;;; special structures to charts. It's ugly, but it'll be needed for solving
;;; PDEs.
```

\section*{C.1.25 pde-aux.scm}
;; This file defines some auxiliary data structures for the PDE code.
(declare (usual-integrations))
```

;;; Linear equations:

```
(define (make-equation node constant terms)
    (vector constant terms node))
(define (equation:get-constant equation)
```

    (vector-ref equation 0))
    (define (equation:get-terms equation)
(vector-ref equation 1))
(define (equation:get-node equation)
(vector-ref equation 2))
(define (equation:get-id equation)
(node:get-id (equation:get-node equation)))
(define (null-equation? equation)
(let loop ((terms (equation:get-terms equation)))
(if (null? terms)
(let ((node (equation:get-node equation)))
(list (node:get-point node) (node:boundary? node)))
(if (almost-zero? (term:get-coeff (car terms)))
(loop (cdr terms))
\#f))))

```
; ; ; Terms in linear equations:
(define (make-term node value)
    (vector node value))
(define (term:get-node term)
    (vector-ref term 0 ))
(define (term:get-id term)
    (node:get-id (term:get-node term)))
(define (term:get-coeff term)
    (vector-ref term 1))

\section*{C.1.26 pde-charts.scm}
;;; This file defines the structures necessary on charts to facilitate PDE
;;; integration.
(declare (usual-integrations))
; ; ; When these charts are constructed, the space has already been discretized.
;;; It only remains to compute the appropriate coefficients with respect to the
;;; given differential operator.
(define (make-pde-chart chart extra-nodes discretize complex)
    ; ; NODES should be the list of nodes used to discretize this chart.
    ; ; DISCRETIZE should be a procedure that takes a differential operator and
    ; ; whatever extra arguments it requires to produce a list of equations. Note
    ; ; that it doesn't need a list of nodes because that state can already be
    ; ; encapsulated in the proceure.
    (chart:install-extra
        chart 'pde-chart
```

    (vector (complex->vertices complex)
            extra-nodes
                        discretize
            complex
            (concat-node-list extra-nodes)
            '())
    chart)
    (define (pde-chart? chart)
(if (chart:get-extra chart 'pde-chart)
\#t
\#f))
(define (chart:get-vertices chart)
(let ((result (chart:get-extra chart 'pde-chart)))
(if result
(vector-ref result 0)
\#f)))
(define (chart:get-extra-nodes chart)
;; This returns a list of lists, where sublists contain extra nodes for
;; corresponding faces in the face list of the complex.
(let ((result (chart:get-extra chart 'pde-chart)))
(if result
(vector-ref result 1)
\#f)))
(define (chart:get-nodes chart)
(let ((result (chart:get-extra chart 'pde-chart)))
(if result
(append (vector-ref result 0) (vector-ref result 4))
\#f)))
(define (chart:get-discretizer chart)
(let ((result (chart:get-extra chart 'pde-chart)))
(if result
(vector-ref result 2)
\#f)))
(define (chart:set-discretizer! chart discretize)
(let ((result (chart:get-extra chart 'pde-chart)))
(if result
(vector-set! result 2 discretize)
\#f)))
(define (chart:get-complex chart)
(let ((result (chart:get-extra chart 'pde-chart)))
(if result
(vector-ref result 3)
\#f)))
(define (chart:get-elements chart)
(let ((result (chart:get-extra chart 'pde-chart)))
(if result
(vector-ref result 5)
\#f)))

```
```

(define (chart:set-elements! chart elements)
(let ((result (chart:get-extra chart 'pde-chart)))
(if result
(vector-set! result 5 elements)
\#f))
(define (chart:in-an-element? p chart)
(and (chart:member? p chart)
(let ((x (chart:point->coords p chart)))
(let loop ((elements (chart:get-elements chart)))
(if (null? elements)
\#f
(if (element:member? (car elements) x)
\#t
(loop (cdr elements)))))))
(define (chart:coords->elements x chart)
(if (chart:in-range? x chart)
(let loop ((elements (chart:get-elements chart)) (result '()))
(if (null? elements)
result
(if (element:member? (car elements) x)
(loop (cdr elements) (cons (car elements) result))
(loop (cdr elements) result)))
'()))
(define (chart:point->elements p chart)
(chart:coords->elements (chart:point->coords p chart) chart))
(define (chart:node->elements node chart)
(chart:point->elements (node:get-point node) chart))
(define (chart:coords->any-element x chart)
(if (chart:in-range? x chart)
(let loop ((elements (chart:get-elements chart)))
(if (null? elements)
\#f
(if (element:member? (car elements) x)
(car elements)
(loop (cdr elements)))))
\#f))
(define (chart:point->any-element p chart)
(chart:coords->any-element (chart:point->coords p chart) chart))
(define (chart:node->any-element node chart)
(chart:point->any-element (node:get-point node) chart))
;;; A useful routine:
(define (concat-node-list node-list)
(let ((all-nodes (apply append node-list)))
(for-each (lambda (node)
(node:set-id! node \#f))
all-nodes)
(let loop ((l all-nodes) (result '()))

```
```

(if (null? 1)
result
(let ((node (car l)))
(if (node:get-id node)
(loop (cdr l) result)
(begin
(node:set-id! node \#t)
(loop (cdr l) (cons node result))))))))

```
```

;;; This actually doesn't do anything, except it gives us the flexibility of
;; using different discretization algorithms on different charts. For now,
;;; all we have is Galerkin's method.
(define (chart:discretize-pde chart source extra-args)
(let ((discretize (chart:get-discretizer chart)))
(if discretize
(apply discretize '(,chart ,source ,@extra-args))
(error "Error: Cannot discretize this chart."))))

```

\section*{C.1.27 pde-cmpgrd.scm}
;;; Following the ideas and algorithms in Chesshire and Henshaw's paper, let's
;; ; try to duplicate (as much as possible) what their program, CMPGRD, does.
;; ; Note that they use finite differences and this program uses finite
;;; elements. So their interpolation kluge doesn't seem nearly as natural
;; ; here; one is very tempted to interpolate using finite element basis
;; ; functions, which doesn't seem to work.
(declare (usual-integrations))
;; Classify the nodes as interpolation, discretization, or exterior, as done
;;; in the paper.
(define (cmpgrd:combine-equations domain equations)
    (let* ((cv (list->vector (cons 'foo! (manifold:get-finite-atlas domain))))
            (chart-count (- (vector-length cv) 1))
            (coords (make-vector (+ chart-count 1) \#f)))
        ; ; Make the array one longer than it should so that the array indexing
        ;; conforms to the pseudocode in the paper... (Bletch!)
        (vector-set! cv 0 chart-count)
        (vector-set! coords 0 chart-count)
        ; First, construct and initialize each entry in the array.
        (write-line '(cmpgrd step 0: initializing array...))
        (do ( \(k 1(+k\) 1)))
            ((> k chart-count))
        (vector-set! coords \(k\)
                                    (make-vector (length (chart:get-nodes (vector-ref cv k)))
                                    chart-count)))
```

    ;; Next, follow the steps in the paper.
    ;; Step 1: Assign IDs to *all* nodes.
    (write-line '(cmpgrd step 1: assigning ids to all nodes...))
    (cmpgrd:step-1/assign-ids cv)
    ;; Step 2: Mark exterior nodes. (Is this really necessary?)
    (write-line '(cmpgrd step 2: marking exterior nodes...))
    (cmpgrd:step-2/mark-exterior-nodes cv coords)
    ;; Step 3: Find interpolating nodes.
    (write-line '(cmpgrd step 3: finding interpolation nodes...))
    (cmpgrd:step-3/find-interpolating-nodes cv coords)
    ;; Step 4: Mark necessary interpolation nodes.
    (write-line '(cmpgrd step 4: marking necessary interpolation nodes...))
    (cmpgrd:step-4/mark-necessary-interpolation-nodes cv coords)
    ;; Step 5: Delete interpolation points.
    (write-line '(cmpgrd step 5: deleting extra interpolation points...))
    (cmpgrd:step-5/delete-interpolation-points cv coords)
    ;; Step 6: Fix the entries in the table.
    (write-line '(cmpgrd step 6: fixing up table...))
    (cmpgrd:step-6/fix-table cv coords)
    ;; Finally, generate the appropriate constraints and produce a matrix:
    (write-line '(cmpgrd final step: generating matrix...))
    (cmpgrd:final-step/generate-matrix cv coords domain equations)))
    ;;; Step 1: Assign IDs to *all* nodes, sequentially (beginning with 0) within
;;; each chart.
(define (cmpgrd:step-1/assign-ids cv)
(let ((n (vector-ref cv 0)))
(do ((k 1 (+ k 1)))
((> k n))
(do ((nodes (chart:get-nodes (vector-ref cv k)) (cdr nodes))
(i 0 (+ i 1)))
((null? nodes))
(let ((node (car nodes)))
(node:set-id! node i))))))
;;; Step 2: Mark exterior nodes.
(define (cmpgrd:step-2/mark-exterior-nodes cv coords)
(let ((n (vector-ref cv 0)))
(do ((k 1 (+ k 1)))
((> k n))
(for-each
(lambda (node)

```
```

        (if (node:boundary? node)
            (let ((p (node:get-point node)))
                (do ((k-prime 1 (+ k-prime 1)))
                        ((> k-prime n))
            (if (not (= k-prime k))
                                (let* ((nodes (chart:get-nodes (vector-ref cv k-prime)))
                                    (node (car nodes)))
                                    (let loop ((nodes (cdr nodes))
                                    (dist (vector:distance
                                    p (node:get-point node)))
                                    (id (node:get-id node)))
                            (if (null? nodes)
                                (vector-set! (vector-ref coords k-prime) id 0)
                                (let* ((node (car nodes))
                                    (new-d (vector:distance
                                    p (node:get-point node))))
                                    (if (< new-d dist)
                                    (loop (cdr nodes) new-d (node:get-id node))
                                    (loop (cdr nodes) dist id))))))))))
    (chart:get-nodes (vector-ref cv k))))))

```
```

;;; Step 3: Find interpolating nodes.

```
;;; Step 3: Find interpolating nodes.
(define (cmpgrd:step-3/find-interpolating-nodes cv coords)
(define (cmpgrd:step-3/find-interpolating-nodes cv coords)
    (let ((n (vector-ref cv 0)))
    (let ((n (vector-ref cv 0)))
        (let loop ((count 1))
        (let loop ((count 1))
            (let ((change-count 0))
            (let ((change-count 0))
                (do ((k 1 (+ k 1)))
                (do ((k 1 (+ k 1)))
                            ((> k n))
                            ((> k n))
                    (let ((v (vector-ref coords k)))
                    (let ((v (vector-ref coords k)))
                        (for-each
                        (for-each
                        (lambda (node)
                            (let ((i (node:get-id node))
                                (p (node:get-point node)))
                                ;; In the paper, l = k-prime.
                        (let ((valid-point? #f))
                        (do ((l (vector-ref v i) (- l 1)))
                        ((or (zero? l) valid-point?))
                        (cond ((=k l)
                        (if (node:local-boundary? node)
                        (begin
                                    (set! change-count (+ change-count 1))
                                    (vector-set! v i (- (vector-ref v i) 1)))))
                                    ((not (chart:in-an-element? p (vector-ref cv 1)))
                                    (set! change-count (+ change-count 1))
                                    (vector-set! v i (- (vector-ref v i) 1)))
                                    (else (set! valid-point? #t)))))))
                        (chart:get-nodes (vector-ref cv k)))))
            (if (> change-count 0)
                        (begin
```

```
(write-line '(,change-count changes. number of tries = ,count))
```

(loop (+ count 1)))))))

```
;;; Step 4: Mark necessary interpolation nodes.
(define (cmpgrd:step-4/mark-necessary-interpolation-nodes cv coords)
    (let ((n (vector-ref cv 0)))
        (do ((k 1 (+ k 1)))
            ((>kn))
            (let ((v (vector-ref coords k)))
                (for-each
                    (lambda (node)
                            (let* ((i (node:get-id node))
                            (l (vector-ref v i)))
                            (if (and (< 1 k) (> 1 0))
                            (let ((w (vector-ref coords 1)))
                                    (for-each
                                    (lambda (needed)
                                    (let ((j (node:get-id needed)))
                                    (vector-set! w j (- (abs (vector-ref w j))))))
                                    (chart:needed-nodes node (vector-ref cv 1)))))))
                    (chart:get-nodes (vector-ref cv k)))))))
;;; Step 5: Delete interpolation points.
(define (cmpgrd:step-5/delete-interpolation-points cv coords)
    (let ((n (vector-ref cv 0)))
        (do ((k 1 (+k 1)))
            ((>kn))
            (let* ((v (vector-ref coords k))
                (i-nodes (cmpgrd:step-5/get-interpolation-nodes cv v k)))
                (for-each
                    (lambda (node)
                        (let ((i (node:get-id node)))
                        (if (> (vector-ref v i) 0)
                                (vector-set! v i 0))))
                    i-nodes)
                (for-each
                    (lambda (node)
                        (if (not (or (node:boundary? node)
                            (node:local-boundary? node)))
                            (vector-set! v (node:get-id node) k)))
                    i-nodes)
                (for-each
                    (lambda (node)
                        (let ((l (vector-ref v (node:get-id node))))
                            (if (> l k)
                                (let ((w (vector-ref coords l)))
                                (for-each
                        (lambda (needed)
                            (let ((j (node:get-id needed)))
                            (vector-set! w j (- (abs (vector-ref w j))))))
                                (chart:needed-nodes node (vector-ref cv l)))))))
            i-nodes))))(
```

```
(define (cmpgrd:step-5/get-interpolation-nodes cv v k)
    (let loop ((nodes (chart:get-nodes (vector-ref cv k))) (result '()))
        (if (null? nodes)
            result
            (let ((val (vector-ref v (node:get-id (car nodes)))))
                (if (not (or (= val k) (zero? k)))
                        (loop (cdr nodes) (cons (car nodes) result))
                        (loop (cdr nodes) result))))))
;;; Step 6: Fix the entries in the table.
(define (cmpgrd:step-6/fix-table cv coords)
    (let ((n (vector-ref cv 0)))
        (do ((k 1 (+ k 1)))
            ((>k n))
            (let ((v (vector-ref coords k)))
                (for-each
                    (lambda (node)
                        (let* ((i (node:get-id node))
                            (abs-val (abs (vector-ref v i))))
                    (cond ((= abs-val k)
                            (vector-set! v i abs-val))
                        ((> abs-val 0)
                            (vector-set! v i (- abs-val))))))
                (chart:get-nodes (vector-ref cv k)))))))
;;; Final step: Generate the interpolation equations and produce the final
;;; matrix:
(define (cmpgrd:final-step/generate-matrix cv coords domain equations)
    ;; First, loop through the charts and pick up the nodes.
    (let ((chart-count (vector-ref cv 0))
            (m 0)
            (n 0)
            (mat #f)
            (constraints '()))
        (let next-chart ((k 1) (result '()))
            (if (<= k chart-count)
                        (let ((chart (vector-ref cv k))
                            (v (vector-ref coords k)))
                    (let next-node ((nodes (chart:get-nodes chart))
                            (result result))
                    (if (null? nodes)
                        (next-chart (+ k 1) result)
                        (let* ((node (car nodes))
                            (val (vector-ref v (node:get-id node))))
                    (if (< val 0)
                        (let* (Cother (vector-ref cv (- val)))
                                    (eq (chart:pointwise-constraint node other)))
                            (if eq
                                    (next-node (cdr nodes) (cons eq result))
                                    (next-node (cdr nodes) result)))
                                    (next-node (cdr nodes) result))))))
```

```
        (set! constraints result)))
;; Next, create the matrix. The IDs need to be reset first:
(let loop ((nodes (manifold:get-nodes domain)) (count 0))
    (if (null? nodes)
            (set! m count)
            (let ((node (car nodes)))
                (if (node:boundary? node)
                    (begin
                        (node:set-id! node 'boundary-node!)
                            (loop (cdr nodes) count))
                (begin
                            (node:set-id! node count)
                            (loop (cdr nodes) (+ count 1))))))
(set! n (+ m 1))
(set! mat (make-sparse-matrix m n))
;; Finally, copy the equations into the matrix while replacing equations
;; corresponding to interpolation nodes with the corresponding constraint
;; equation.
(let ((ev (make-vector m #f)))
    ;; Need to keep track of equations:
    (for-each
        (lambda (equation)
            (vector-set! ev
                        (node:get-id (equation:get-node equation))
                            equation))
        equations)
        ;; Constraints can overwrite equations:
        (for-each
        (lambda (constraint)
                (vector-set! ev
                    (node:get-id (equation:get-node constraint))
                    constraint))
        constraints)
        ;; Now just copy!
        (do ((i O (+ i 1)))
            ((>= i m) mat)
        (let ((eq (vector-ref ev i)))
            (if eq
                                    (begin
                            (sparse-matrix-set! mat i m (equation:get-constant eq))
                            (let next-term ((terms (equation:get-terms eq)))
                            (if (not (null? terms))
                            (let* ((term (car terms))
                                    (j (term:get-id term))
                                    (val (term:get-coeff term)))
                                    (sparse-matrix-set! mat i j val)
                            (next-term (cdr terms))))))
                    (write-line '(*** warning: row ,i of matrix is null!))))))))
```

```
;;; With the roles of the nodes figured out, here's the real work: Generate the
;;; appropriate constraints. First, we need to figure out which nodes to
;;; interpolate from:
(define (chart:needed-nodes node chart)
    ;; Instead of using basis functions to interpolate, maybe we should follow
    ;; Chesshire & Henshaw's suggestion and create higher-order *interpolating
    ;; equations* (constraint equations, in our language) by extending the
    ;; constraint to elements neighboring the one containing the given point.
    ;; For now, let's just use the finite element interpolation and test the rest
    ;; of the Chesshire-Henshaw code.
    (element:get-nodes (chart:node->any-element node chart)))
```


## C.1.28 pde-collect.scm

;; Use this file to collect data for theis work. Based on pde-test.scm

```
(load "pde-test")
```

;; Define the test procedures:
(define test-1
(pde:experiment-too 'pde:make-domain-without-overlaps
'combine-equations-without-overlap))
(define test-2
(pde:experiment-too 'pde:make-domain-with-small-overlaps
'combine-equations-without-overlap))
(define test-3
(pde:experiment-too 'pde:make-domain-with-overlaps
'combine-equations-with-overlap))
(define test-4
(pde:experiment-too 'pde:make-domain-with-larger-overlaps
'combine-equations-with-overlap))
(define test-5
(pde:experiment-too 'pde:make-simple-domain
'combine-equations-with-overlap))
(define test-6
(pde:experiment-too 'pde:make-simple-domain
'combine-equations-using-cmpgrd))
;; ; Run the experiments (sorted by size, not test):
(test-1 '(rectangular 10 5) '(spherical 5 10) 100001.7 "Data/disc/test1a")
(test-2 '(rectangular 10 5) '(spherical 5 10) 100001.7 "Data/disc/test2a")
(test-3 '(rectangular 10 5) '(spherical 5 10) 100001.7 "Data/disc/test3a")

```
(test-4 '(rectangular 10 5) '(spherical 5 10) 10000 1.7 "Data/disc/test4a")
(test-5 '(rectangular 10 5) '(spherical 5 10) 10000 1.7 "Data/disc/test5a")
(test-6 '(rectangular 10 5) '(spherical 5 10) 10000 1.7 "Data/disc/test6a")
(test-1 '(rectangular 20 10) '(spherical 10 20) 10000 1.7 "Data/disc/test1b")
(test-2 '(rectangular 20 10) '(spherical 10 20) 10000 1.7 "Data/disc/test2b")
(test-3 '(rectangular 20 10) '(spherical 10 20) 10000 1.7 "Data/disc/test3b")
(test-4 '(rectangular 20 10) '(spherical 10 20) 10000 1.7 "Data/disc/test4b")
(test-5 '(rectangular 20 10) '(spherical 10 20) 10000 1.7 "Data/disc/test5b")
(test-6 '(rectangular 20 10) '(spherical 10 20) 10000 1.7 "Data/disc/test6b")
(test-1 '(rectangular 40 15) '(spherical 20 30) 10000 1.7 "Data/disc/test1c")
(test-2 '(rectangular 40 15) '(spherical 20 30) 10000 1.7 "Data/disc/test2c")
(test-3 '(rectangular 40 15) '(spherical 20 30) 10000 1.7 "Data/disc/test3c")
(test-4 '(rectangular 40 15) '(spherical 20 30) 10000 1.7 "Data/disc/test4c")
(test-5 '(rectangular 40 15) '(spherical 20 30) 10000 1.7 "Data/disc/test5c")
(test-6 '(rectangular 40 15) '(spherical 20 30) 10000 1.7 "Data/disc/test6c")
(test-1 '(rectangular 60 30) '(spherical 37 61) 10000 1.7 "Data/disc/test1d")
(test-2 '(rectangular 60 30) '(spherical 37 61) 10000 1.7 "Data/disc/test2d")
(test-3 '(rectangular 60 30) '(spherical 37 61) 10000 1.7 "Data/disc/test3d")
(test-4 '(rectangular 60 30) '(spherical 37 61) 10000 1.7 "Data/disc/test4d")
(test-5 '(rectangular 60 30) '(spherical 37 61) 10000 1.7 "Data/disc/test5d")
(test-6 '(rectangular 60 30) '(spherical 37 61) 10000 1.7 "Data/disc/test6d")
```


## C.1.29 pde-collect1.scm

;; ; Just the last two tests, which take more memory, from pde-collect.scm.

```
(load "pde-test")
```

```
(define test-5
    (pde:experiment-too 'pde:make-simple-domain
                        'combine-equations-with-overlap))
(define test-6
    (pde:experiment-too 'pde:make-simple-domain
                        'combine-equations-using-cmpgrd))
(test-5 '(rectangular 60 30) '(spherical 37 61) 10000 1.7 "Data/disc/test5d")
(test-6 '(rectangular 60 30) '(spherical 37 61) 10000 1.7 "Data/disc/test6d")
```


## C.1.30 pde-collect2.scm

```
;;; Even more tests:
(load "pde-test")
;;; Define the test procedures:
(define test-7
    (pde:experiment-too 'pde:make-simple-domain
                        'combine-equations-with-overlap1))
```

```
(define test-8
    (pde:experiment-too 'pde:make-simple-domain
                                    'combine-equations-with-overlap2))
```

```
;;; Run the experiments (sorted by size, not test):
```

;;; Run the experiments (sorted by size, not test):
(test-7 '(rectangular 10 5) '(spherical 5 10) 10000 1.7 "Data/disc/test7a")
(test-7 '(rectangular 10 5) '(spherical 5 10) 10000 1.7 "Data/disc/test7a")
(test-8 '(rectangular 10 5) '(spherical 5 10) 10000 1.7 "Data/disc/test8a")
(test-8 '(rectangular 10 5) '(spherical 5 10) 10000 1.7 "Data/disc/test8a")
(test-7 '(rectangular 20 10) '(spherical 10 20) 10000 1.7 "Data/disc/test7b")
(test-7 '(rectangular 20 10) '(spherical 10 20) 10000 1.7 "Data/disc/test7b")
(test-8 '(rectangular 20 10) '(spherical 10 20) 10000 1.7 "Data/disc/test8b")
(test-8 '(rectangular 20 10) '(spherical 10 20) 10000 1.7 "Data/disc/test8b")
(test-7 '(rectangular 40 15) '(spherical 20 30) 10000 1.7 "Data/disc/test7c")
(test-7 '(rectangular 40 15) '(spherical 20 30) 10000 1.7 "Data/disc/test7c")
(test-8 '(rectangular 40 15) '(spherical 20 30) 10000 1.7 "Data/disc/test8c")

```
(test-8 '(rectangular 40 15) '(spherical 20 30) 10000 1.7 "Data/disc/test8c")
```


## C.1.31 pde-collect3.scm

```
;;; Even more tests:
(load "pde-test")
;;; Define the test procedures:
(define test-7
    (pde:experiment-too 'pde:make-simple-domain
                                    'combine-equations-with-overlap1))
(define test-8
    (pde:experiment-too 'pde:make-simple-domain
                                'combine-equations-with-overlap2))
;;; Run the experiments (sorted by size, not test):
(test-7 '(rectangular 40 15) '(spherical 20 30) 10000 1.7 "Data/disc/test7c")
(test-8 '(rectangular 40 15) '(spherical 20 30) 10000 1.7 "Data/disc/test8c")
```


## C.1.32 pde-collect4.scm

;; J Just the last two tests, which take more memory, from pde-collect.scm.
(load "pde-test")
;; Define the test procedures:
(define test-7
(pde:experiment-too 'pde:make-simple-domain
'combine-equations-with-overlap1))
(define test-8
(pde:experiment-too 'pde:make-simple-domain
'combine-equations-with-overlap2))

```
;;; Run the experiments (sorted by size, not test):
(test-7 '(rectangular 60 30) '(spherical 37 61) 10000 1.7 "Data/disc/test7d")
(test-8 '(rectangular 60 30) '(spherical 37 61) 10000 1.7 "Data/disc/test8d")
```


## C.1.33 pde-config.scm

;; This file defines procedures that need to be easily modifiable:
;;; Methods for generating equations from triangulated domains:
(define combine-equations-without-overlap
(pde:equation-maker merge-equations))
(define combine-equations-with-overlap
(pde: equation-maker (append-constraint-equations make-ordered-boundary-constraints)))
(define combine-equations-with-overlap1
(pde: equation-maker (append-constraint-equations make-all-constraints)))
(define combine-equations-with-overlap2
(pde:equation-maker (append-constraint-equations make-all-ordered-constraints)))
(define combine-equations-using-CMPGRD
(pde:equation-maker cmpgrd:combine-equations))
;;; Methods for triangulating domains:
(define pde:make-domain-without-overlaps
(pde:domain-maker generate-node-lists exact-overlap))
(define pde:make-domain-with-small-overlaps
(pde:domain-maker copy-between-node-lists exact-overlap))
(define pde:make-domain-with-overlaps
(pde:domain-maker make-nodes-for-each-chart reduce-overlap))
(define pde:make-domain-with-larger-overlaps
(pde:domain-maker make-nodes-for-each-chart extended-overlap))
(define pde:make-simple-domain
(pde:domain-maker make-nodes-for-each-chart do-nothing-to-complex))

## C.1.34 pde-debug.scm

;; This file is based on fem/debug.scm.
(declare (usual-integrations))

```
;;; Drawing a domain requires some care with element edges:
(define (draw-domain domain)
    (draw-charts-in-domain (manifold:get-finite-atlas domain) domain))
;;; Drawing a chart is much simpler:
(define (draw-chart chart)
    (let ((edges (complex->edges (chart:get-complex chart)))
            (nodes (chart:get-nodes chart)))
        (apply 2d-draw
                        (append
                        (list (chart:get-nodes chart) edges node:get-coords drav-line)
                        (bounding-box nodes node:get-coords)))))
(define (draw-chart-in-domain chart domain)
    (draw-charts-in-domain (list chart) domain))
(define (draw-charts-in-domain charts domain)
    (let ((edges (append-map (compose complex->edges chart:get-complex) charts))
                    (nodes (append-map chart:get-nodes charts)))
        (apply 2d-draw
                    (append
                            (list nodes edges node:get-point draw-edge-in-chart)
                            (bounding-box (manifold:get-nodes domain) node:get-point)))))
;;; Actually draw something:
(define *planar-device* 'undefined)
(define (make-2d-draw colors)
    (let ((background (vector-ref colors 0))
            (cursor (vector-ref colors 1))
            (line (vector-ref colors 2))
            (boundary (vector-ref colors 3))
            (loc-bound (vector-ref colors 4))
            (node (vector-ref colors 5))
            (bnode (vector-ref colors 6))
            (gnode (vector-ref colors 7))
                (border-frac .05)
                (cross-frac .005))
        (lambda (nodes edges get-coords draw-edge
                        x-left y-bottom x-right y-top)
            ;; Report the dimensions:
            (write-line '(x range: ,x-left to ,x-right))
            (write-line '(y range: ,y-bottom to ,y-top))
            ;; Open a graphics window, if there isn't one already:
            (if (eq? *planar-device* 'undefined)
                    (set! *planar-device* (make-graphics-device 'x))
                    (graphics-clear *planar-device*))
```

```
(let ((dev *planar-device*)
    (bx (* border-frac (- x-right x-left)))
    (by (* border-frac (- y-top y-bottom)))
    (ex (* cross-frac (- x-right x-left)))
    (ey (* cross-frac (- y-top y-bottom))))
    (if (zero? bx) (set! bx .5))
    (if (zero? by) (set! by .5))
    (if (zero? ex) (set! ex cross-frac))
    (if (zero? ey) (set! ey cross-frac))
    ;; Set up the window:
    (write-line '(setting up...))
    (graphics-set-coordinate-limits
    dev (- x-left bx) (- y-bottom by) (+ x-right bx) (+ y-top by))
    (graphics-operation dev 'set-foreground-color line)
    (graphics-operation dev 'set-background-color background)
    (graphics-operation dev 'set-mouse-color cursor)
    (graphics-clear dev)
    (graphics-enable-buffering dev)
    ;; First, draw the edges (the graphics are cooler if one sorts the
    ;; edges first):
    (write-line '(drawing ,(length edges) edges...))
    (let ((color 'line))
        (for-each
            (lambda (e)
            (let* ((org-e (car e))
                                    (dest-e (cadr e))
                                    (org (get-coords org-e))
                                    (dest (get-coords dest-e)))
                    (cond ((and (node:boundary? org-e) (node:boundary? dest-e))
                                (if (not (eq? color 'boundary))
                                (begin
                            (set! color 'boundary)
                            (graphics-operation
                                    dev 'set-foreground-color boundary))))
                                    ((and (node:local-boundary? org-e)
                            (node:local-boundary? dest-e))
                (if (not (eq? color 'local-boundary))
                        (begin
                                    (set! color 'local-boundary)
                                    (graphics-operation
                                    dev 'set-foreground-color loc-bound))))
                                    Celse
                                    (if (not (eq? color 'line))
                        (begin
                            (set! color 'line)
                            (graphics-operation
```

```
                dev 'set-foreground-color line)))))
```

```
            (draw-edge dev e)))
        edges))
    ;; Next, draw the nodes:
    (write-line '(drawing ,(length nodes) nodes...))
    (let ((color 'node))
    (graphics-operation dev 'set-foreground-color node)
    (for-each
        (lambda (n)
            (let* ((coords (get-coords n))
                        (x (vector-ref coords 0))
                    (y (vector-ref coords 1)))
                (cond ((node:boundary? n)
                                (if (not (eq? color 'bnode))
                                (begin
                                (set! color 'bnode)
                                    (graphics-operation
                                    dev 'set-foreground-color bnode)))
                                (draw-boundary-node dev x y ex ey))
                        ((node:constrained? n)
                                (if (not (eq? color 'gnode))
                                (begin
                                    (set! color 'gnode)
                                    (graphics-operation
                                    dev 'set-foreground-color gnode)))
                                    (draw-glued-node dev x y ex ey))
                                    (else
                                (if (not (eq? color 'node))
                        (begin
                                (set! color 'node)
                                    (graphics-operation
                                    dev 'set-foreground-color node)))
                                    (draw-node dev x y ex ey)))))
        nodes))
    (write-line '(flushing buffers...))
    (graphics-disable-buffering dev))
'done)))
;;; Different color schemes for different purposes:
(define *standard-colors*
    (vector "black" ;; Background.
```

```
    "white" ;; Cursor.
    "blue" ;; Regular edges.
    "red" ;; Boundary edges.
    "purple" ;; Local boundaries.
    "white" ;; Regular node color.
    "orange" ;; Boundary node color.
    "green")) ;; "Glued" node color.
(define *boring-colors*
    (vector "white" ;; Background.
        "black" ;; Cursor.
        "black" ;; Regular edges.
        "black" ;; Boundary edges.
        "black" ;; Local boundaries.
        "black" ;; Regular node color.
        "black" ;; Boundary node color.
        "black")) ;; "Glued" node color.
(define *print-colors*
    (vector "white" ;; Background.
        "black" ;; Cursor.
        "blue" ;; Regular edges.
        "red" ;; Boundary edges.
        "yellow" ;; Local boundaries.
        "black" ;; Regular node color.
        "red" ;; Boundary node color.
        "black")) ;; "Glued" node color.
(define 2d-draw (make-2d-draw *standard-colors*))
;;; Methods for drawing nodes:
(define (draw-node dev x y ex ey)
    ;; An 'x' is a regular node:
    (graphics-draw-line dev (- x ex) (- y ey) (+ x ex) (+ y ey))
    (graphics-draw-line dev (- x ex) (+ y ey) (+ x ex) (- y ey)))
(define (draw-boundary-node dev x y ex ey)
    ;; A square is a boundary node:
    (let ((x-ex (- x ex))
        (x+ex (+ x ex))
        (y-ey (- y ey))
        (y+ey (+ y ey)))
        (graphics-move-cursor dev x-ex y-ey)
        (graphics-drag-cursor dev x-ex y+ey)
        (graphics-drag-cursor dev x+ex y+ey)
        (graphics-drag-cursor dev x+ex y-ey)
        (graphics-drag-cursor dev x-ex y-ey)))
(define (draw-glued-node dev x y ex ey)
    ;; A triangle is a node glued to another chart:
```

```
    (let ((x-ex (- x ex))
        (x+ex (+ x ex))
        (y-ey (- y ey))
        (y+ey (+ y ey)))
    (graphics-draw-line dev x-ex y-ey x y+ey)
    (graphics-draw-line dev x y+ey x+ex y-ey)
    (graphics-draw-line dev x+ex y-ey x-ex y-ey)))
;;; Getting rid of the window is sometimes useful, too!
(define (close)
    (if (not (eq? *planar-device* 'undefined))
        (begin
            (graphics-close *planar-device*)
            (set! *planar-device* 'undefined))))
;;; In order to get edges drawn correctly, we need to walk around the
;;; parametrized path in the chart and then map it into the manifold (the
;;; straight line in the manifold seldom works correctly):
(define draw-edge-in-chart
    (let* ((step-count 40)
                (dt (exact->inexact (/ step-count))))
        (lambda (dev edge)
            (let* ((org (node:get-coords (car edge)))
                        (dir (vector:- (node:get-coords (cadr edge)) org))
                        (f (chart:get-inverse-map (node:get-chart (car edge)))))
            (let loop ((i 0) (p (f org)) (q (f (vector:+ (vector:* dt dir) org)))
                        (if (< i step-count)
                        (let ((i+1 (+ i 1)))
                        (graphics-draw-line dev
                                    (vector-ref p 0) (vector-ref p 1)
                                    (vector-ref q O) (vector-ref q 1))
                                    (loop i+1 q (f (vector:+ (vector:* (* i+1 dt) dir) org))))
                    'done))))))
;;; Draw a straight line:
(define (draw-line dev edge)
    (let ((org (node:get-coords (car edge)))
            (dest (node:get-coords (cadr edge))))
        (graphics-draw-line dev
                                    (vector-ref org 0) (vector-ref org 1)
                                    (vector-ref dest 0) (vector-ref dest 1))))
;;; What we have so far actually doesn't work! Even though we get more
;;; equations than unknowns, the rank is smaller than the number of unknowns
;;; (i.e. the system is underdetermined). What can be going wrong?
;;; 1. Something is wrong in applying FEM, most notably FEM-DISCRETIZE in
;;; pde-tools.scm.
;;; 2. Something is wrong in assembling the equations after that, as in
;;; COMBINE-EQUATIONS.
```

;;; 3. Something is wrong in generating constraints.

```
;;; 4. Something is wrong in collecting the equations and constraints.
;;; 5. Something is wrong in *controlling* when to generate constraints.
;;; (5) is the hardest one to fix, but it's most likely what's wrong with the
;;; approach.
;;; We can do a end-to-end test of 1, 2, and 4 by extracting the local matrices
;;; and comparing them with the local results. (There shouldn't be any bugs in
;;; FEM, right?) It turns out that the specific example we have does pass the
;;; test. So we need to figure out if there are problems with the constraints
;;; (items 3 and 5).
```

(define (extract-local-matrix mat chart)
(let ( icount 0)
(ncount-1 (- (matrix-column-count mat) 1))
(result \#f)
(indices '()))
(let loop ((nodes (chart:get-nodes chart)) (count 0) (ilist '()))
(if (null? nodes)
(begin
(set! icount count)
(set! result (make-matrix icount (+ icount 1)))
(set! indices (sort ilist <)))
(let ((id (node:get-id (car nodes))))
(if (number? id)
(loop (cdr nodes) (+ count 1) (cons id ilist))
(loop (cdr nodes) count ilist)))))
(do ((ilist indices (cdr ilist))
(m O (+ m 1)))
((null? ilist) result)
(let ( i (car ilist)))
(do ((jlist indices (cdr jlist))
(n 0 (+ n 1)))
((null? jlist))
(let ( $(\mathrm{j}$ (car jlist)))
(matrix-set! result m n (matrix-ref mat $i j)$ ))
(matrix-set! result m icount (matrix-ref mat i ncount-1)))))
(define (compute-local-matrix source chart)
(sparse->matrix (assemble-equations
source (list->vector (chart:get-nodes chart)))))
(define (compare-matrices A B)
(let ( (m (matrix-row-count A))
(n (matrix-column-count A)))
(do ( $\mathrm{i} 0(+\mathrm{i} 1)$ )
( $>=\mathrm{im}$ ) )
(do ((j $0(+j 1)))$
( $(>=\mathrm{j} n)$ )
(let ((diff (- (matrix-ref A i j) (matrix-ref Bim)))
(if (not (zero? diff))
(write-line
'( (, i , j)

```
(- ,(matrix-ref A i j),(matrix-ref B i j))
=> ,diff)))))))
```

```
;;; Test if a column of a matrix is all almost zero.
(define (matrix:null-column? mat j)
    (let ((m (matrix-row-count mat)))
        (let loop ((i 0))
                (if (< i m)
                    (if (almost-zero? (matrix-ref mat i j))
                            (loop (+ i 1))
                            #f)
                    #t)))
(define (matrix:null-row? mat i)
    (let ((n (matrix-column-count mat)))
            (let loop ((j 0))
                (if (< j n)
                    (if (almost-zero? (matrix-ref mat i j))
                            (loop (+ j 1))
                            #f)
                            #t))))
(define (matrix:find-null-columns mat)
    (let ((n (matrix-column-count mat)))
        (let loop ((j O) (results '()))
                (if (< j n)
                    (if (matrix:null-column? mat j)
                            (loop (+ j 1) (cons j results))
                            (loop (+ j 1) results))
                    results))))
(define (matrix:find-null-rows mat)
    (let ((m (matrix-row-count mat)))
        (let loop ((i 0) (results '()))
                (if (< i m)
                        (if (matrix:null-row? mat i)
                            (loop (+ i 1) (cons i results))
                            (loop (+ i 1) results))
                    results))))
;;; Compute some errors:
(define (process-absolute-errors process nodes f v)
        (apply process
            (append-map
                        (lambda (node)
                    (let ((index (node:get-id node)))
                            (if (and (number? index) (not (node:constrained? node)))
                            (list (abs (- (f node) (vector-ref v index))))
                                '()))
                nodes)))
(define (max-error domain f v)
        (process-absolute-errors max (manifold:get-nodes domain) f v))
(define (min-error domain f v)
```

```
    (process-absolute-errors min (manifold:get-nodes domain) f v))
(define (avg-error domain f v)
    (/ (process-absolute-errors + (manifold:get-nodes domain) f v)
            (vector-length v)))
(define (process-relative-errors process nodes f v)
    (apply process
                (append-map
                (lambda (node)
                    (let ((index (node:get-id node)))
                        (if (number? index)
                                    (list (relative-error (vector-ref v index) (f node)))
                                    '()))
                nodes)))
(define (max-relative-error domain f v)
    (process-relative-errors max (manifold:get-nodes domain) f v))
(define (min-relative-error domain f v)
    (process-relative-errors min (manifold:get-nodes domain) f v))
(define (avg-relative-error domain f v)
    (/ (process-relative-errors + (manifold:get-nodes domain) f v)
            (vector-length v)))
;;; Somewhat more useful information about how bad the solutions are (if the
;;; number of nodes is reasonably small):
(define (display-results domain f v)
    (for-each
        (lambda (node)
            (let ((id (node:get-id node)))
            (if (number? id)
                    (write-line '((id = ,id)
                                    (computed = ,(vector-ref v (node:get-id node)))
                                    (actual = ,(f node))
                            (x = ,(node:get-x node) y = ,(node:get-y node))))))
        (manifold:get-nodes domain)))
```

;; This is so things can be drawn in MATLAB or Maple: Draw a rectangle that
;; ; bounds the manifold, divide it into an MxN grid, drop the nodes and average
;;; the values. Output is a matrix.
(define (manifold->grid m $n$ domain $f=$ diff)
(let ((nodes (manifold:get-nodes domain)))
(if (null? nodes)
(error "Manifold contains no nodes! -- MANIFOLD->GRID"))
;; Save nodal values:
(for-each
(lambda (node)
(let ((index (node:get-id node)))
(if (number? index)

```
        (node:set-value! node (vector-ref v index)))))
nodes)
(let* ((p (node:get-point (car nodes)))
        (x-min (vector-first p))
        (x-max (vector-first p))
        (y-min (vector-second p))
        (y-max (vector-second p)))
    ;; Then find the bounding rectangle:
    (for-each
        (lambda (node)
        (let* ((p (node:get-point node))
                        (x (vector-first p))
                            (y (vector-second p)))
            (cond ((> x x-max) (set! x-max x))
                ((< x x-min) (set! x-min x)))
            (cond ((> y y-max) (set! y-max y))
                    ((< y y-min) (set! y-min y)))))
    (cdr nodes))
    ;; Next, create the matrix and start averaging:
    (let ((mat (make-matrix m n))
        (count (make-matrix m n))
        (dx (/ (- x-max x-min) m))
        (dy (/ (- y-max y-min) n)))
    ;; Collect the sums and count the number of nodes in each box:
    (for-each
        (lambda (node)
            (let* ((p (node:get-point node))
                                    (x (- (vector-first p) x-min))
                                    (y (- (vector-second p) y-min))
                                    (i (inexact->exact (floor (/ x dx))))
                            (j (inexact->exact (floor (/ y dy)))))
                ;; Silly fence-post conditions:
            (if (= i m) (set! i (- m 1)))
            (if (= j n) (set! j (- n 1)))
                (matrix-set! count i j (+ 1 (matrix-ref count i j)))
                (matrix-set! mat i j
                                    (+ (diff (node:get-value node) (f node))
                                    (matrix-ref mat i j)))))
        nodes)
        ;; Normalize:
        (do ((i O (+ i 1)))
            ((>= i m) mat)
        (do ((j O (+ j 1)))
            ((>= j n))
                (if (> (matrix-ref count i j) 0)
                    (matrix-set!
                            mat i j
```

```
(/ (matrix-ref mat i j) (matrix-ref count i j)))))))))
```

```
;;; This lets save the results of a computation, if not the finite element
;;; stuff itself:
(define (node-states domain f v)
    (let ((nodes (manifold:get-nodes domain)))
        (if (null? nodes)
            #f
            (let* ((n (real-node-count nodes))
                    (mat (make-matrix n 4)))
                (for-each
                    (lambda (node)
                        (let ((id (node:get-id node)))
                        (if (number? id)
                            (let ((p (node:get-point node)))
                                (matrix-set! mat id O (vector-first p))
                                (matrix-set! mat id 1 (vector-second p))
                                (matrix-set! mat id 2 (vector-ref v id))
                                (matrix-set! mat id 3 (f node))))))
                        nodes)
                        mat))))
(define (real-node-count nodes)
    (let loop ((count 0) (nodes nodes))
        (if (null? nodes)
            count
            (let ((id (node:get-id (car nodes))))
                (if (number? id)
                    (loop (+ count 1) (cdr nodes))
                    (loop count (cdr nodes))))))
```


## C.1.35 pde-elements.scm

```
;;; This file defines tools for dealing with elements.
```

;;; This file defines tools for dealing with elements.
(declare (usual-integrations))
;;; A template for element constructors:
(define (pde:element-maker L
make-integrator
make-basis-function)
; ; The operator abstraction in the FEM program is a bit artificial (see
;; ELEMENT-MAKER). Note that the interpolation of variable coefficients
;; *there* may not be necessary, as the operator is already given a PDE-chart
; ; along with a list of the nodes.
(lambda (chart)
;; This ugly hack sort of works (for now). Really ought to just make
;; ELEMENT-MAKER to do the right thing (whatever that is).
(element-maker

```
```

(lambda (nodes)
(operator:set-context! L chart nodes)
L)
make-integrator
make-basis-function)))

```
;; ; We need to know when a node belongs to an element. Note that this only
;; works with simplices. For more complicated shapes, we would require more
;; structure (i.e. a list of *faces* of the boundary of the convex element),
;; ; which can only be supplied by TESSELATE during domain construction.
(define (element:member? element p)
    (let ((vertices (map node:get-coords (element:get-vertex-nodes element))))
        (in-simplex? p vertices)))
;;; Some procedures that help with ELEMENT:MEMBER?.
(define (in-simplex? point vertices)
    (in-convex-domain?
        point vertices (choose-sublists vertices (vector-length point))))
(define (in-convex-domain? point vertices faces)
    (not (memq \#f (map (lambda (face)
                                    (same-side? point
                                    (find-another-vertex face vertices)
                                    (car face)
                                    (cdr face)))
                    faces))) (
(define (find-another-vertex face vertices)
    (let loop ((vertices vertices))
        (if (null? vertices)
            \#f
            (let ((vertex (car vertices)))
                    (if (member vertex face)
                        (loop (cdr vertices))
                                vertex)))))
(define (same-side? p q origin basis)
    ; S See if \(P\) and \(Q\) lie on the same side of the \(n\)-1-dimensional hyperplane
    ; defined by translating the span of BASIS from 0 to ORIGIN.
    (let ((basis (map (lambda (v) (vector:- v origin)) basis))
            (p (vector:- p origin))
            (q (vector:- q origin)))
        (let ((val (* (det (list->vector (cons p basis)))
                        (det (list->vector (cons q basis))))))
        (or ( \(>=\operatorname{val} 0\) )
                        (almost-zero? val)))))

\section*{C.1.36 pde-examples.scm}
```

;;; examples. A good example to use first is Laplace's equation, since the
;;; answers are easy to check. Another option is to define something on the
;;; torus or the sphere -- How does one define the Laplacian in that case?
;;; Laplacian = (d + d*) }2= = d*d + dd*. On 0-forms (smooth functions) d* = 0
;;; so it's just d*d. How does one compute the adjoint locally?
;;; Laplace's equation in a compact region of the plane (with boundary),
;;; covered by more than one coordinate system.
;;; First, let the domain be the unit closed disc, with spherical coordinates
;;; on the boundary:
(define disc
(make-ball 2 make-spherical-sphere))
;;; Define some Laplacian operators (on different basis functions):
(define poly-disc-laplacian
(make-operator
disc
(operator:pull-back-poly-op
poly-gradient
poly-gradient
(lambda (v w) (basis:scalar* -1 (basis:dot v w))))))
(define imbedded-poly-1aplacian
(make-operator
disc
(operator:imbedded-poly-op
poly-gradient
poly-gradient
(lambda (v w) (basis:scalar* -1 (basis:dot v w))))))
(define real-disc-laplacian
(make-operator
disc
(operator:pull-back-real-op
real-gradient
real-gradient
(lambda (v w) (basis:scalar* -1 (basis:dot v w))))))
;;; Some other things that are useful as test solutions:
(define x-coord-map
(compose vector-first node:get-point))
(define y-coord-map
(compose vector-second node:get-point))
(define (test-function node)
(let ((x (x-coord-map node))
(y (y-coord-map node)))
(- (square x) (square y))))

```
;; Useful definitions to have (for debugging purposes):
```

(define atlas (manifold:get-finite-atlas disc))
(define c1 (car atlas))
(define c2 (cadr atlas))
(define c3 (caddr atlas))

```
; ; ; The null equation corresponds to the node in row 228 of MAT. It is the
\(; ;\) node at the center of the disc. Why are its coefficients 0 ? We should
; ; ; never get null equations because the diagonal terms should at least be
;;; non-zero.
;; ; It may appear that the problem lies with the fact that differential
;;; operator operates on (inexact, interpolated) pull-backs of basis functions
;;; back onto the imbedded-manifold.
;;; But why should this cause problems on C3, which is really a subset of \(\mathrm{R}^{\wedge} 2\) ?
;;; What really must be happening is that Galerkin's method doesn't work unless
;;; one performs the integration by parts, which is not exactly kosher because
;; the basis functions are only piecewise-differentiable. So a variational
;; ; principle must be really what's at work in finite elements...
;; Using first-order operators and inner products seems to get rid of the
;; ; problem. This is because while the basis functions are \(C^{\wedge} 2\) over elements,
;;; they are only C~O across edges. Thus, differential operator we can safely
;; ; apply to basis functions have at most order 1, and it is necessary to split
;;; the operator using integration by parts.
;;; Most of the error is probably coming from truncation errors in computing
;;; the differential operator, since we cut off higher-order terms when
;; ; applying coordinate transformations. The other possible source of error is
;; the constraint. First thing to try, then, is to implement a nice
;;; multidimensional numerical integrator.

\section*{C.1.37 pde-gentest.scm}
; ; ; Useful for generating the test cases in pde-thesis.scm:
(define (generate-test-case ilist rectangular spherical count sor-fact string) (for-each
            (newline)
            (display
                    (string-append
                    "(test-" (number->string i) " "
                "'(rectangular " (number->string (cadr rectangular)) " "
                (number->string (caddr rectangular)) ") "
                "'(spherical " (number->string (cadr spherical)) " "
                (number->string (caddr spherical)) ") "
                (number->string count) " "
                (number->string sor-fact) " "
                "\"Data/thesis/test" (number->string i) string "\")")))
        ilist)
    (newline))
;;; Generate the test cases:
```

(define (generate-test-set indices)
(newline)
(display "*** Test cases:")
(newline)
(for-each
(lambda (args)
(apply generate-test-case (cons indices args)))
(list (list '(rectangular 10 5) '(spherical 5 10) 10000 1.9 "a")
(list '(rectangular 14 7) '(spherical 7 14) 11000 1.9 "b")
(list '(rectangular 18 9)'(spherical 9 18) 12000 1.9 "c")
(list '(rectangular 22 11) '(spherical 11 22) 13000 1.9 "d")
(list '(rectangular 26 13) '(spherical 13 26) 14000 1.9 "e")
(list '(rectangular 30 15) '(spherical 15 30) 15000 1.9 "f")
(list '(rectangular 34 17) '(spherical 17 34) 16000 1.9 "g")
(list '(rectangular 38 19) '(spherical 19 38) 17000 1.9 "h")
(list '(rectangular 42 21) '(spherical 21 42) 18000 1.9 "i")
(list '(rectangular 46 23) '(spherical 23 46) 19000 1.9 "j")
(list '(rectangular 50 25) '(spherical 25 50) 20000 1.9 "k"))))

```

\section*{C.1.38 pde-main.scm}
;;; This program relies on the existence of local triangulation algorithms for ;;; arbitrary dimensions. That problem does appear to be solved: See Barber,
;;; Dobkin, Huhdanpaa, "The Quickhull Algorithm for Convex Hulls." The paper,
;; as well as the software itself, are available at the University of
;;; Minnesota's Geometry Center, "http://www.geom.umn.edu/software/qhull/".
; ; S See pde-main.scm.old for lots and lots of comments and design notes. (It's
;; ; a bit incoherent, so I've deleted them from this version.)
;;; NOTE:
;;; Yet another possible approach is to look through the differential topology
;; ; literature and see if there exists a (constructive) proof that manifolds
;; ; are triangulable or CW complexes. One can probably show that manifolds are
;;; CW complexes by Morse theory; is this computable? Does "computational
;;; Morse theory" exist?
;;; If we require the manifolds to be CW complexes or simplicial compexes to ;; begin with, all these problems would be solved.
(declare (usual-integrations))
(load "pde-mergers")
;; ; Given a domain with constructed elements, a source function, and a boundary
;; ; value function, produce the appropriate discretized equation. The nodes
; ; ; are left with indices that specify their corresponding row in the matrix.
(define (pde:equation-maker merge-equations)
(lambda (domain source boundary-value . extra-args)
; \(\operatorname{EXTRA-ARGS}\) gives us finer control over the discretization.
; ; DOMAIN should be a manifold that already has PDE structures constructed. ; ; Hence, it contains information about the operator (through the elements ; ; in its discretized charts).
```

    ;; BOUNDARY-VALUE is irrelevant for domains without boundary. Just specify
    ;; anything (but do put in something).
    (let* (M domain)
            (charts (manifold:get-finite-atlas M))
            (nodes (list->vector (append-map chart:get-nodes charts)))
            (ncount (vector-length nodes)))
    ;; CHART:DISCRETIZE-PDE should return a list of linear equations. First,
    ;; set the boundary values:
    (write-line '(,ncount nodes generated...))
    (write-line '(setting boundary values...))
    (do ((i O (+ i 1)))
        ((>= i ncount))
        (let ((node (vector-ref nodes i)))
            (if (node:boundary? node)
                (node:set-value! node (boundary-value node)))))
    ;; Next, compute the local equation systems:
    (write-line '(computing ,(length charts) local systems of equations...))
    (let ((equations (append-map
                        (lambda (chart)
                            (chart:discretize-pde chart source extra-args))
                        charts)))
        ;; Compute constraints:
        (write-line '(merging local equations...))
        (show-time
            (lambda ()
                (merge-equations domain equations)))))))
    ;;; This procedure creates a constructor that, given a manifold M, creates the
;;; data structures necessary for solving PDEs. The arguments must agree with
;;; the manifold on a contract that lets procedures obtain chart information.
(define (pde:domain-maker generate-node-lists process-complex)
(lambda (M
make-vertices
make-extra-nodes
tesselate
. argl)
;; First, make the bounding nodes of the convex domain, and then
;; triangulate and make the extra nodes:
(let ((atlas (manifold:get-finite-atlas M)))
(if (not atlas)
(error "Error: Can only do FEM with finite atlases."))
(write-line '(tesselating domain...))

```
```

;; Do something more complicated here to reduce the overlap:
(let loop ((charts atlas)
(node-lists (generate-node-lists make-vertices atlas argl)))
(if (not (null? charts))
;; TESSELATE should return a list of lists, where each list
;; contains the elemental faces of a given dimension (in some given
;; polytope). In the planar case, this reverses the convention in
;; fem.scm: The list should be sorted by dimension in *descending*
;; order.
(let* ((chart (car charts))
(nodes (car node-lists))
(complex (process-complex (tesselate nodes) (cdr charts)))
(extra-nodes (make-extra-nodes complex)))
;; By default, use FEM-DISCRETIZE. Can replace with others.
(make-pde-chart chart extra-nodes fem-discretize complex)
(loop (cdr charts) (cdr node-lists)))))
;; Construct elements. We don't need to explicitly mark boundaries
;; because manifolds should already have such structures defined.
(lambda (operator make-integrator make-basis-function)
(let ((element-maker (pde:element-maker operator
make-integrator
make-basis-function)))
(write-line '(constructing elements...))
(for-each
(lambda (chart)
;; Construct the elements:
(write-line
'(making ,(length (complex->faces (chart:get-complex chart)))
elements...))
(let* ((make-element (element-maker chart))
(new-elements (show-time
(lambda ()
(map make-element
(complex->faces
(chart:get-complex chart))
(chart:get-extra-nodes chart)))))
(chart:set-elements! chart new-elements)))

```
            atlas))()))
; ; ; This procedure uses the implicit ordering of the charts to remove extra
; ; ; nodes in overlaps and to duplicate enough nodes so that the meshes can be
;;; "glued" together.
(define (make-nodes-for-each-chart make-nodes charts extra-args)
```

    (map (lambda (chart) (apply make-nodes (cons chart extra-args))) charts))
    (define (generate-node-lists make-nodes charts argl)
;; Generate a list of nodes for each chart, then loop over the charts. Note
;; that the earlier a chart is in the list, the less likely its nodes are to
;; survive.
(let next-chart ((charts charts)
(lists (make-nodes-for-each-chart make-nodes charts argl))
(result '())
(reversed '())
(count 0))
(if (null? charts)
(copy-overlap-nodes count result reversed)
(next-chart (cdr charts)
(cdr lists)
(cons (remove-overlap-nodes (car lists) (cdr charts))
result)
(cons (car charts) reversed)
(+ count 1)))))
(define (copy-between-node-lists make-nodes charts argl)
;; Same as GENERATE-NODE-LISTS, but doesn't call REMOVE-OVERLAP-NODES.
(let ((node-lists (make-nodes-for-each-chart make-nodes charts argl)))
(copy-overlap-nodes (length charts)
(reverse node-lists)
(reverse charts))))
;;; Take out all nodes in NODES that belong to any of the charts in CHARTS.
(define (remove-overlap-nodes nodes charts)
(let next-node ((nodes nodes) (result '()))
(if (null? nodes)
result
(let* ((node (car nodes))
(p (node:get-point node)))
(let next-chart ((charts charts))
(if (null? charts)
(next-node (cdr nodes) (cons node result))
(if (chart:member? p (car charts))
(next-node (cdr nodes) result)
(next-chart (cdr charts)))))))))
;;; For each node list in LISTS, take each node and see if it's in one of the
;;; charts that come after the node's own chart in list-order. If so, make a
;;; copy of that node and put it in the corresponding chart. Note that the
;;; order of node lists is reversed.
(define (copy-overlap-nodes count lists charts)
(let ((v (make-vector count '())))
(let next-list ((lists lists) (charts charts) (i 0) (result '()))
(if (null? lists)
result
(let next-node ((nodes (car lists)))
(if (null? nodes)
(next-list (cdr lists) (cdr charts) (+ i 1)

```
```

                            (cons (append (vector-ref v i) (car lists)) result))
    (let ((node (car nodes)))
(if (or (node:local-boundary? node)
(node:boundary? node))
(let ((p (node:get-point node)))
(let next-chart ((charts (cdr charts))
(j (+ i 1))
(l (cdr lists)))
(if (null? charts)
(next-node (cdr nodes))
(let ((chart (car charts)))
(if (chart:member? p chart)
(let ((other (close-node p (car l))))
(if other
(node:set-constraint! other node)
(vector-set! v j
(cons
(node:copy node chart)
(vector-ref v j))))))
(next-chart (cdr charts) (+ j 1) (cdr l))))))
(next-node (cdr nodes)))))))))

```
(define close-node
    (let* ((close-enuf? (make-comparator .01))
                (too-close? (lambda (p q)
                            (close-enuf? (vector:distance \(p q\) ) 0))))
        (lambda (p 1)
            (let loop ((1))
                (if (null? 1)
                    \#f
                        (if (too-close? p (node:get-point (car 1)))
                        (car 1)
                            (loop (cdr 1)))))))
;;; After filtering out nodes, local boundary information becomes useless...
(define (exact-overlap complex charts)
    (kill-extra-nodes complex charts)
    (resurrect-only-connected-nodes complex charts)
    (keep-only-live-nodes complex charts))
(define (remove-overlap complex charts)
    (kill-extra-nodes complex charts)
    (resurrect-some-nodes complex charts)
    (keep-only-live-nodes complex charts))
(define (reduce-overlap complex charts)
    (kill-extra-nodes complex charts)
    (resurrect-some-nodes complex charts)
    (resurrect-some-nodes complex charts)
    (keep-only-live-nodes complex charts))
(define (absolutely-no-overlap complex charts)
    (kill-extra-nodes complex charts)
    (keep-only-live-nodes complex charts))
(define (extended-overlap complex charts)
    (extend-local-boundary complex charts)
    complex)
```

(define (extend-local-boundary complex charts)
(write-line '(extending local boundary...))
(let loop ((edges (complex->edges complex)) (keep '()))
(if (null? edges)
(for-each
(lambda (node)
(node:set-local-boundary! node \#t))
keep)
(let ((n1 (caar edges))
(n2 (cadar edges)))
(if (node:local-boundary? n1)
(if (node:local-boundary? n2)
(loop (cdr edges) keep)
(loop (cdr edges) (cons n2 keep)))
(if (node:local-boundary? n2)
(loop (cdr edges) (cons n1 keep))
(loop (cdr edges) keep)))))))
(define (do-nothing-to-complex complex charts)
complex)
(define (kill-extra-nodes complex charts)
;; Figure out which nodes to keep by looking at the overlaps:
(write-line '(processing ,(length (complex->vertices complex)) nodes...))
(let next-node ((nodes (complex->vertices complex)))
(if (not (null? nodes))
(let ((node (car nodes)))
(let ((p (node:get-point node)))
(let next-chart ((charts charts))
(if (null? charts)
(next-node (cdr nodes))
(let ((chart (car charts)))
(if (chart:member? p chart)
(let ((node (car nodes)))
(node:kill! node)
(node:set-local-boundary! node \#f)
(next-node (cdr nodes)))
(next-chart (cdr charts)))))))))))
(define (resurrect-some-nodes complex charts)
;; We actually need to keep more nodes than this, because we need *some*
;; overlap, though not too much. We also need more sophisticated ways of
;; checking whether a node should be kept.
(write-line '(figuring out overlaps...))
(let loop ((faces (complex->faces complex)) (keep '()))
(if (null? faces)
(for-each
(lambda (face)
(for-each
(lambda (node)

```
```

            (if (not (node:active? node))
                (begin
                (node:set-local-boundary! node #t)
                (node:resurrect! node))))
            face))
        keep)
        (if (save-face? (car faces) charts)
            (loop (cdr faces) (cons (car faces) keep))
            (loop (cdr faces) keep)))))
    (define (resurrect-only-connected-nodes complex charts)
;; Only keep nodes that are connected to live ones:
(write-line '(figuring out overlaps...))
(let loop ((faces (complex->faces complex)) (keep '()))
(if (null? faces)
(for-each
(lambda (face)
(for-each
(lambda (node)
(if (not (node:active? node))
(begin
(node:set-local-boundary! node \#t)
(node:resurrect! node)))
face))
keep)
(if (at-least-one-live-node? (car faces) charts)
(loop (cdr faces) (cons (car faces) keep))
(loop (cdr faces) keep)))))
(define (keep-only-live-nodes complex charts)
;; Figure out which faces/edges/etc. to keep:
(write-line '(processing complex...))
(let loop ((complex complex) (result '()))
(if (null? complex)
(reverse result)
(let inner-loop ((faces (car complex)) (okay-faces '()))
(if (null? faces)
(loop (cdr complex) (cons okay-faces result))
(let* ((face (car faces))
(list? (list? face)))
(if (or (and list? (not (memq \#f (map node:active? face))))
(and (not list?) (node:active? face)))
(inner-loop (cdr faces) (cons face okay-faces))
(inner-loop (cdr faces) okay-faces))))))))

```
;; ; We need to check if the particular element covers a region of the manifold
;; ; that isn't covered by another chart. It is hard to do this in general, but
; ; we can use a probabilistic algorithm and take advantage of the fact that
;; ; elements are convex:
(define save-face?
```

    ;; KLUGE-FACTOR defines how many random points to try. It should scale up
    ;; with the dimension/size of the element, but for now it's constant.
    (let ((kluge-factor 40))
        (lambda (face charts)
        (or (at-least-one-live-node? face charts)
            ;; Go through a complicated (and probabilistic) test:
            (let ((vertices (map node:get-coords face))
                (chart (node:get-chart (car face))))
                (let ((m (length face))
                    (n (chart:dimension chart)))
                    (let loop ((k 0))
                        (if (< k kluge-factor)
                        ; Contruct a random point in the element:
                        (let ((v (make-random-probability-vector m))
                        (w (make-vector n 0)))
                        (do ((i O (+ i 1))
                                    (vertices vertices (cdr vertices)))
                                ((>= i m))
                                (let ((coeff (vector-ref v i))
                                (x (car vertices)))
                                (do ((j O (+ j 1)))
                                    ((>= j n))
                                    (vector-set! w j (+ (vector-ref w j)
                                    (* coeff (vector-ref x j)))))))
                                ;; Now test it:
                        (let ((p (chart:coords->point w chart)))
                        (let next-chart ((charts charts))
                                (if (null? charts)
                                    #t
                                    (if (chart:member? p (car charts))
                                    (loop (+ k 1))
                                    (next-chart (cdr charts)))))))
                    #f))())())
    (define (make-random-probability-vector n)
(let ((v (make-vector n)))
(let loop ((i 0) (sum 0))
(if (< i n)
(let ((val (random 1.)))
(vector-set! v i val)
(loop (+ i 1) (+ val sum)))
(do ((j 0 (+ j 1)))
((>= j n) v)
(vector-set! v j (/ (vector-ref v j) sum)))))))
;;; Here is a simpler variant:
(define (at-least-one-live-node? face charts)
(memq \#t (map node:active? face)))

```
```

;;; A useful procedure that gets all the nodes out of the domain:
(define (manifold:get-nodes domain)
(append-map chart:get-nodes (manifold:get-finite-atlas domain)))

```

\section*{C.1.39 pde-mergers.scm}
```

;;; This file defines some ways of putting together equations from different
;;; charts.
(declare (usual-integrations))
;;; Just adding linear combinations of old equations to the matrix won't do
;;; anything new. We also need to eliminate old equations -- How do we do
;;; this? In the case when two nodes overlap, the choice is simple: Just
;;; eliminate the two original equations, and replace one of the old nodes with
;;; the other. But what do we do when the node lies *inside* an element?
;;; Let's leave that for future work, and instead implement a method based on
;;; "copying" nodes from other charts to guarantee that nodes overlap exactly
;;; in interactions. Then this reduces to something like the exact case, and
;;; we know how to combine equations in this case.
;;; By the way, is it important to enforce constraints on boundary nodes? The
;;; current design makes this impossible to do, but one could probably fix it.
(define (merge-equations domain equations)
(let ((nodes (manifold:get-nodes domain))
(count 0)
(mat \#f))
;; First, assign IDs to nodes, and create the matrix:
(write-line '(creating matrix...))
(let loop ((nodes nodes) (i 0))
(if (null? nodes)
(begin
(set! count i)
(set! mat (make-sparse-matrix count (+ count 1))))
(let ((node (car nodes)))
(cond ((node:boundary? node)
(node:set-id! node 'boundary-node!)
(loop (cdr nodes) i))
((node:get-constraint node)
(node:set-id! node 'constrained-node!)
(loop (cdr nodes) i))
(else
(node:set-id! node i)
(loop (cdr nodes) (+ i 1))))))
;; Next, start filling in equations while keeping track of constraints:
(write-line '(copying equations...))

```
```

(let next-eq ((equations equations))
(if (null? equations)
(begin
(write-line '(done!))
mat)
(let* ((eq (car equations))
(i (node:get-real-id (equation:get-node eq))))
(sparse-matrix-set! mat i count
(+ (equation:get-constant eq)
(sparse-matrix-ref mat i count)))
(let next-term ((terms (equation:get-terms eq)))
(if (null? terms)
(next-eq (cdr equations))
(let* ((term (car terms))
(j (node:get-real-id (term:get-node term)))
(val (term:get-coeff term)))
(sparse-matrix-set! mat i j (+ (sparse-matrix-ref mat i j)
val))
(next-term (cdr terms))))))))))

```
;;; A slightly different way to merge equations that requires overlaps: Append
;;; linear constraints that force nodal values in overlapping regions to agree
;; ; with the interpolated value in the other chart. Note that if one does this
;; f for *all* nodes, it might "stiffen" the solution over the overlap and force
;;; it to be approximately linear. Thus, one should try to avoid having too
;;; many constrained nodes, or to somehow reduce the overlap.
(define (append-constraint-equations make-constraints)
    (lambda (domain equations)
; ; First, set IDs and clear hidden states:
(write-line '(setting node ids...))
(let loop ((id 0) (nodes (manifold:get-nodes domain)))
    (if (not (null? nodes))
            (let ((node (car nodes)))
                (node:set-constraint! node \#f)
                (if (node:boundary? node)
                        (begin
                        (node:set-id! node 'boundary-node!)
                        (loop id (cdr nodes)))
                    (begin
                        (node:set-id! node id)
                                (loop (+ id 1) (cdr nodes))))))
; ; Next, generate constraints:
(write-line '(generating constraints...))
(with-values
            (lambda () (make-constraints domain))
            (lambda (c-count clists)
            (let* ((eq-count (length equations))
```

            (m (+ eq-count c-count))
            (n (+ eq-count 1)))
    (write-line '(constructing a matrix of dimension (,m ,n)...))
(let ((mat (make-sparse-matrix m n)))
;; First, copy the equations:
(write-line '(copying ,eq-count equations...))
(for-each
(lambda (eq)
(let ((i (equation:get-id eq)))
(sparse-matrix-set!
mat i eq-count (equation:get-constant eq))
(for-each
(lambda (term)
(sparse-matrix-set! mat i (term:get-id term)
(term:get-coeff term)))
(equation:get-terms eq))))
equations)
;; Next, copy the constraints:
(write-line '(copying ,c-count constraints...))
(let next-clist ((i eq-count) (clists clists))
(if (null? clists)
mat
(let next-constraint ((clist (car clists)) (i i))
(if (null? clist)
(next-clist i (cdr clists))
(let ((constraint (car clist)))
(sparse-matrix-set!
mat i eq-count (equation:get-constant constraint))
(for-each
(lambda (term)
(sparse-matrix-set! mat i (term:get-id term)
(term:get-coeff term)))
(equation:get-terms constraint))
(next-constraint (cdr clist) (+ i 1)))))))))))))
;;; Here's one way to make constraints:
(define (make-ordered-boundary-constraints domain)
(let* ((charts (manifold:get-finite-atlas domain))
(result-1 (charts->constraints charts node:local-boundary?))
(result-2 (charts->constraints
(reverse charts) node:local-boundary?)))
(values (+ (car result-1) (car result-2))
(append (cadr result-1) (cadr result-2)))))
(define (charts->constraints charts good-node?)
(let next-chart ((charts charts)
(count 0)
(clists '()))
(if (null? charts)

```
```

    (list count clists)
    ;; Go through each node in the chart and check for constraints:
    (let ((chart (car charts)))
    (let next-node ((nodes (chart:get-nodes chart))
                        (count count)
                        (clist '()))
        (if (null? nodes)
            (next-chart (cdr charts) count (cons clist clists))
            (let ((node (car nodes)))
                (if (and (good-node? node)
                        (not (node:get-constraint node))
                            (not (node:boundary? node)))
                                (let ((eq (make-constraint node (cdr charts))))
                        (if eq
                            (next-node (cdr nodes) (+ count 1) (cons eq clist))
                                    (next-node (cdr nodes) count clist)))
                                    (next-node (cdr nodes) count clist)))))))))
    (define (make-constraint node charts)
(let loop ((charts charts))
(if (null? charts)
\#f
(let ((eq (chart:pointwise-constraint node (car charts))))
(if eq
eq
(loop (cdr charts)))))))
;;; A slightly different approach that generates *more* constraints:
(define make-all-ordered-constraints
(let ((exists? (lambda (node) \#t)))
(lambda (domain)
(let* ((charts (manifold:get-finite-atlas domain))
(result-1 (charts->constraints charts exists?))
(result-2 (charts->constraints (reverse charts) exists?)))
(values (+ (car result-1) (car result-2))
(append (cadr result-1) (cadr result-2)))))))
;;; Finally, something that generates a lot of constraints:
(define (make-all-constraints domain)
(let ((constraints
(append-map
(lambda (pair)
(let ((chart-1 (car pair))
(chart-2 (cadr pair)))
(append (constrain-all-nodes chart-1 chart-2)
(constrain-all-nodes chart-2 chart-1))))
(pairs (manifold:get-finite-atlas domain)))))
(values (length constraints) (list constraints))))
(define (constrain-all-nodes chart-1 chart-2)
(append-map
(lambda (node)

```
```

(if (node:boundary? node)
'()
(let ((eq (chart:pointwise-constraint node chart-2)))
(if eq
(list eq)
(())))
(chart:get-nodes chart-1)))
;;; Some extensions to charts:
;;; Here's one problem with this approach to constraints, though: Consider the
;;; case when we *do* have a mesh over a simple subset of the plane. Let's try
;;; to apply this constraint idea to this case: Cut the mesh along some line
;;; formed by the edges, so that we cut the meshed region }\mathbb{R}\mathrm{ into two subregions
;;; R1 and R2. Let's try to paste R1 and R2 together using constraints. What
;;; we notice is that when we identify two nodes (by adding a constraint
;;; equation), we are actually requiring that the nodal value satisfies *two*
;;; separate equations, one for R1 and one for R2, instead of satisfying the
;;; *sum* of those two equations. What this indicates is that this method
;;; actually *requires* overlaps.
;;; Furthermore, note that building constraints by putting in (indeterminate)
;;; Dirichlet boundary conditions along chart boundaries won't work. Consider
;;; the ideal case, where two charts overlap by exactly their boundary: Since
;;; the Dirichlet problem is well-posed for Laplace's equation, we can put
;;; *any* "boundary value" on this overlap and still get a solution of the
;;; equation over the whole domain. Clearly, this idea *may* work if one uses
;;; von Neumann conditions rather than Dirichlet conditions, but how to do that
;;; nicely is not clear. Perhaps using Lagrange multipliers for a constrained
;;; minimization...
(define (chart:pointwise-constraint node chart)
;; The coefficients of a linear constraint for some node x should simply be
;; the value at p of the basis function centered at }x\mathrm{ . This linearity
;; depends only on the fact that the solution is approximated by a linear
;; combination of basis functions.
(if (chart:member? (node:get-point node) chart)
(let* ((x (chart:point->coords (node:get-point node) chart))
(element (chart:coords->any-element x chart)))
(if element
(let loop ((nodes (element:get-nodes element))
(i 0)
(const 0)
(terms (list (make-term node -1))))
(if (null? nodes)
(begin
(node:set-constraint! node chart)
(make-equation node const terms))
(let ((neighbor (car nodes))
(coeff (evaluate-basis-function
(element:get-basis-function element i) x)))
(if (node:boundary? neighbor)
(loop (cdr nodes)
(+ i 1)
(- const (* (node:get-value neighbor) coeff))
terms)

```
```

                    (loop (cdr nodes)
                                    (+ i 1)
                                    const
                                    (cons (make-term neighbor coeff) terms))))))
    \#f))
\#f))

```

\section*{C.1.40 pde-nodes.scm}
```

;;; This file defines nodes for solving PDEs on manifolds. These are more
;;; complicated than nodes in the old FEM program because they need to keep
;;; track of corresponding structures on manifolds.
(declare (usual-integrations))
;;; The constructor now takes a chart. (X should be *coordinates* in the
;;; chart, not the point on the manifold.)
(define (make-node x chart)
(let ((p (chart:coords->point x chart))
(b? (and (boundary-chart? chart)
(chart:range-boundary? x chart))))
(vector x ;; 0. Coordinates.
0. ;; 1. Value.
37 ;; 2. ID.
b? ;; 3. Boundary?
'() ;; 4. Elements.
,() ;; 5. Local IDs.
p ;; 6. Point on manifold.
chart ;; 7. Chart.
'() ;; 8. Basis functions.
'() ;; 9. Extra structures.
\#f ;; A. Is this node constrained to another chart?
\#f ;; B. Is this node on the boundary of the chart?
\#t))) ;; C. Is this guy still alive?

```
;;; Same old methods:
(define (node:get-x node) (vector-ref (node:get-coords node) 0))
(define (node:get-y node) (vector-ref (node:get-coords node) 1))
(define (node:get-z node) (vector-ref (node:get-coords node) 2))
(define (node:get-coords node) (vector-ref node 0))
(define (node:get-value node) (vector-ref node 1))
(define (node:set-value! node val) (vector-set! node 1 val))
(define (node:get-id node) (vector-ref node 2))
(define (node:set-id! node id) (vector-set! node 2 id))
(define (node:boundary? node) (vector-ref node 3))
(define (node:get-elements node) (vector-ref node 4))
(define (node:get-local-ids node) (vector-ref node 5))
(define (node:add-element node element index)
    (vector-set! node 4 (cons element (vector-ref node 4)))
    (vector-set! node 5 (cons index (vector-ref node 5))))
```

;;; Some new additions:
(define (node:get-real-x node) (vector-ref (node:get-point node) 0))
(define (node:get-real-y node) (vector-ref (node:get-point node) 1))
(define (node:get-real-z node) (vector-ref (node:get-point node) 2))
(define (node:get-point node)
(vector-ref node 6))
(define (node:get-chart node)
(vector-ref node 7))
(define (node:get-basis-functions node)
(vector-ref node 8))
(define (node:add-basis-function node basis-function)
(vector-set! node 8 (cons basis-function (vector-ref node 8))))
(define (node:install-extra node tag datum)
(let ((result (assq (vector-ref node 9) tag)))
(if result
(set-cdr! result datum)
(vector-set! node 9 (cons (cons tag datum) (vector-ref node 9))))))
(define (node:get-extra node tag)
(let ((result (assq (vector-ref node 9) tag)))
(if result
(cdr result)
\#f)))
(define (node:constrained? node)
(if (node:get-constraint node)
\#t
\#f))
(define (node:get-constraint node)
(vector-ref node 10))
(define (node:set-constraint! node chart)
(vector-set! node 10 chart))
;;; Copy a node to a different chart:
(define (node:copy node chart)
(let ((new-node (make-node (chart:point->coords (node:get-point node) chart)
chart)))
(node:set-constraint! new-node node)
new-node))
;;; Recursively find the ID of the node to which a given node is constrained:
(define (node:get-real-id node)
(let ((constraint (node:get-constraint node)))
(if constraint
(node:get-real-id constraint)
(node:get-id node))))
;;; (Some of these properties may be obsolete.)

```
```

(define (node:local-boundary? node)
(and (vector-ref node 11)
(not (node:boundary? node))))
(define (node:set-local-boundary! node flag)
(vector-set! node 11 flag))
(define (node:active? node)
(vector-ref node 12))
(define (node:kill! node)
(vector-set! node 12 \#f))
(define (node:resurrect! node)
(vector-set! node 12 \#t))

```

\section*{C.1.41 pde-ops.scm}
;; \(;\) Let's define some differential operators so we have something to test.
(declare (usual-integrations))
;;; Differential operators (ours only act on scalar functions, not sections of
;;; vector bundles):
(define (make-operator M make-local-form)
    (vector M make-local-form \#f '()))
(define (operator:get-manifold L)
    (vector-ref L 0))
;;; Some extra structures for working with FEM. "Context" is a kluge that lets
;; operators learn about the particular element they are working in, etc.
(define (operator:get-local-form operator)
    (let ((context (operator:get-context operator)))
        (if context
                        (apply (vector-ref operator 1) context)
                \#f)) )
(define (operator:set-context! operator . contextual-data)
    (vector-set! operator 2 contextual-data))
(define (operator:get-context operator)
    (vector-ref operator 2))
;;; This might come in handy:
(define (operator:install-extra \(L\) tag datum)
    (let ((result (assq tag (vector-ref L 4))))
        (if result
                        (set-cdr! result datum)
                        (vector-set! L 4 (cons (cons tag datum) (vector-ref L 4))))))
(define (operator:get-extra L tag)
    (let ((result (assq tag (vector-ref L 4))))
```

(if result
(cdr result)
\#f)))

```

\section*{C.1.42 pde-test.scm}
```

;;; See pde-test.scm.old for more information about the errors associated with
;;; different variants of our method.
;;; Let's see how the error scales with the number of nodes:
(define (pde:experiment domain-maker combine-equations)
;; DOMAIN-MAKER and COMBINE-EQUATIONS should be *symbols*, not procedures.
(lambda (rectangular spherical sor-steps sor-coeff port)
;; First, reload everything (to clear residual states in data structures).
(load "load-pde")
(if port (newline port))
;; Let's start:
(let ((make-test-domain
((evaluate-symbol domain-maker) disc
make-vertices
make-no-extra-nodes
planar-triangulate
rectangular
spherical))
(f test-function))
(make-test-domain
imbedded-poly-laplacian
make-triangular-imbedded-integrator
pde:make-imbedded-poly-basis-function)
(let ((mat ((evaluate-symbol combine-equations)
disc 0-function test-function)))
(write-line '(creating normal equations...))
(set! mat (show-time (lambda () (sparse-normal-equations mat))))
(write-line '(solving normal equations...))
(let ((v (show-time (lambda () (sor mat sor-steps sor-coeff))))
(write (lambda (stuff)
(write-line stuff)
(if port
(write-line stuff port)))))
(write '(domain-maker = ,domain-maker))
(write '(combine-equations = ,combine-equations))
(write '(,(length (manifold:get-nodes disc)) nodes))
(write '(max absolute error = ,(max-error disc f v)))
(write '(min absolute error = ,(min-error disc f v)))

```
```

(write '(average absolute error = ,(avg-error disc f v)))
(urite '(max relative error = ,(max-relative-error disc f v)))
(write '(min relative error = ,(min-relative-error disc f v))))))))

```
```

;;; Use \#f for FILE-NAME if standard output is the only desired output port.
;;; Othervise, the output is sent to both standard output and the named file.
(define (run-test-case test-case file-name)
(let ((port (if file-name
(open-output-file file-name)
\#f)))
(write-line '(test case: ,test-case))
(if port (write-line '(test case: ,test-case) port))
(case test-case
((1)
(let ((try-pde (pde:experiment 'pde:make-domain-with-overlaps
'combine-equations-with-overlap)))
(try-pde '(rectangular 10 5) '(spherical 5 10) 10000 1.9 port)
(try-pde '(rectangular 20 10) '(spherical 10 20) 20000 1.9 port)
(try-pde '(rectangular 40 25) '(spherical 25 40) 30000 1.9 port)))
((2)
(let ((try-pde (pde:experiment 'pde:make-domain-with-overlaps
'combine-equations-with-overlap)))
(try-pde '(rectangular 20 10) '(spherical 10 20) 10000 1.9 port)))
((3)
(let ((try-pde (pde:experiment 'pde:make-simple-domain
'combine-equations-using-CMPGRD)))
(try-pde '(rectangular 20 10) '(spherical 10 20) 10000 1.9 port)))
((4)
(let ((try-pde (pde:experiment 'pde:make-simple-domain
'combine-equations-using-CMPGRD)))
(try-pde '(rectangular 10 5) '(spherical 5 10) 10000 1.9 port)
(try-pde '(rectangular 20 10) '(spherical 10 20) 20000 1.9 port)
(try-pde '(rectangular 40 25) '(spherical 25 40) 30000 1.9 port)))
((5)
(let ((try-pde (pde:experiment 'pde:make-domain-with-overlaps
'combine-equations-with-overlap)))
(try-pde '(rectangular 10 5)'(spherical 5 10) 0 0 \#f)))
((6)
(let ((try-pde (pde:experiment 'pde:make-domain-with-small-overlaps
'combine-equations-without-overlaps)))
(try-pde '(rectangular 20 10)'(spherical 10 20) 10000 1.9 \#f)))
(else \#f))
(if port (close-output-port port))))

```
;; A slight variant used for collecting data for thesis work:
(define (pde:experiment-too domain-maker combine-equations)
    ; ; DOMAIN-MAKER and COMBINE-EQUATIONS should be *symbols*, not procedures.
```

    (lambda (rectangular spherical sor-steps sor-coeff file)
    ;; First, reload everything (to clear residual states in data structures).
    (load "load-pde")
    ;; Let's start:
    (show-time
        (lambda ()
            (let ((make-domain
                            ((evaluate-symbol domain-maker) disc
                                    make-vertices
                                    make-no-extra-nodes
                                    planar-triangulate
                                    rectangular
                                    spherical))
                    (f test-function))
            (write-line '(constructing elements...))
            (make-domain
                        imbedded-poly-laplacian
            make-triangular-imbedded-integrator
            pde:make-imbedded-poly-basis-function)
            (let ((mat ((evaluate-symbol combine-equations) disc 0-function f)))
                    (if (not (= (+ (sparse-matrix-row-count mat) 1)
                                    (sparse-matrix-column-count mat)))
                    (begin
                        (write-line '(need normal equations.))
                        (set! mat (show-time
                                    (lambda ()
                                    (sparse-normal-equations mat))))))
                (write-line '(solving equations...))
            (let ((v (show-time (lambda () (sor mat sor-steps sor-coeff)))))
                        (write-line '(preparing to save data...))
                        (let ((states (node-states disc f v))
                        (port (open-output-file file)))
                    (write-line '(saving...))
                        (print-matrix states port)
                        (close-output-port port)))))))))
    ;;; Tests for thesis-related data:
(define test-1
(pde:experiment-too 'pde:make-domain-without-overlaps
'combine-equations-without-overlap))
(define test-2
(pde:experiment-too 'pde:make-domain-with-small-overlaps
'combine-equations-without-overlap))

```
(define test-3
```

(pde:experiment-too 'pde:make-domain-with-overlaps
'combine-equations-with-overlap))
(define test-4
(pde:experiment-too 'pde:make-domain-with-larger-overlaps
'combine-equations-with-overlap))
(define test-5
(pde:experiment-too 'pde:make-simple-domain
'combine-equations-with-overlap))
(define test-6
(pde:experiment-too 'pde:make-simple-domain
'combine-equations-using-cmpgrd))
(define test-7
(pde:experiment-too 'pde:make-simple-domain
'combine-equations-with-overlap1))
(define test-8
(pde:experiment-too 'pde:make-simple-domain
'combine-equations-with-overlap2))

```

\section*{C.1.43 pde-thesis.scm}
; ; Use this file to collect data for theis work. Based on pde-collect.scm.
(load "pde-test")
; ; R Run the experiments (sorted by size, not test):
\begin{tabular}{|c|c|c|}
\hline -1 ' & '(rectangular 10 5) & '(spherical 5 10) 100001.9 "Data/thesis/test1a") \\
\hline (test-2 ' & '(rectangular 10 5) & '(spherical 5 10) 100001.9 "Data/thesis/test2a") \\
\hline (test-3 , & '(rectangular 10 5) & '(spherical 5 10) 100001.9 "Data/thesis/test3a") \\
\hline (test-4 ' & '(rectangular 10 5) & '(spherical 5 10) 100001.9 "Data/thesis/test4a") \\
\hline (test-5 ' & '(rectangular 10 5) & '(spherical 5 10) 100001.9 "Data/thesis/test5a") \\
\hline est-1 ' & '(rectangular 14 7) & '(spherical 7 14) 110001.9 "Data/thesis/test1b") \\
\hline (test-2 ' & '(rectangular 14 7) & '(spherical 7 14) 110001.9 "Data/thesis/test2b") \\
\hline (test-3 , & '(rectangular 14 7) & '(spherical 7 14) 110001.9 "Data/thesis/test3b") \\
\hline (test-4 ' & '(rectangular 14 7) & '(spherical 7 14) 110001.9 "Data/thesis/test4b") \\
\hline (test-5 , & '(rectangular 14 7) & '(spherical 7 14) 110001.9 "Data/thesis/test5b") \\
\hline (test-1 , & '(rectangular 18 9) & '(spherical 9 18) 120001.9 "Data/thesis/test1c") \\
\hline (test-2 ' & '(rectangular 18 9) & '(spherical 9 18) 120001.9 "Data/thesis/test2c") \\
\hline (test-3 ' & '(rectangular 18 9) & '(spherical 9 18) 120001.9 "Data/thesis/test3c") \\
\hline (test-4 , & '(rectangular 18 9) & '(spherical 9 18) 120001.9 "Data/thesis/test4c") \\
\hline (test-5 ' & '(rectangular 18 9) & '(spherical 9 18) 120001.9 "Data/thesis/test5c") \\
\hline (test-1 , & '(rectangular 22 11) & '(spherical 11 22) 130001.9 "Data/thesis/test1d") \\
\hline (test-2 ' & '(rectangular 22 11) & '(spherical 11 22) 130001.9 "Data/thesis/test2d") \\
\hline (test-3 ' & '(rectangular 22 11) & '(spherical 11 22) 130001.9 "Data/thesis/test3d") \\
\hline (test-4 ' & '(rectangular 22 11) & '(spherical 11 22) 130001.9 "Data/thesis/test4d") \\
\hline (test-5 ' & '(rectangular 22 11) & '(spherical 11 22) 130001.9 "Data/thesis/test5d") \\
\hline (test-1 ' & '(rectangular 26 13) & '(spherical 13 26) 140001.9 "Data/thesis/test1e") \\
\hline
\end{tabular}


\section*{C.1.44 pde-thesis1.scm}
;;; Use this file to collect data for theis work. Based on pde-collect.scm.
(load "pde-test")
;; ; Run the experiments (sorted by size, not test):



\section*{C.1.45 pde-thesis2.scm}
;; Crashed in the middle...
(load "pde-test")
(test-7 '(rectangular 38 19) '(spherical 19 38) 170001.9 "Data/thesis/test7h") (test-8 '(rectangular 38 19) '(spherical 19 38) 170001.9 "Data/thesis/test8h")
(test-6 '(rectangular 42 21) '(spherical 21 42) 180001.9 "Data/thesis/test6i")
(test-7 '(rectangular 42 21) '(spherical 21 42) 180001.9 "Data/thesis/test7i")
(test-8 '(rectangular 42 21) '(spherical 21 42) 180001.9 "Data/thesis/test8i")
(test-6 '(rectangular 46 23) '(spherical 23 46) 190001.9 "Data/thesis/test6j")
(test-7 '(rectangular 46 23) '(spherical 23 46) 190001.9 "Data/thesis/test7j")
(test-8 '(rectangular 46 23) '(spherical 23 46) 190001.9 "Data/thesis/test8j")
(test-6 '(rectangular 50 25) '(spherical 25 50) 200001.9 "Data/thesis/test6k")
(test-7 '(rectangular 50 25) '(spherical 25 50) 200001.9 "Data/thesis/test7k")
(test-8 '(rectangular 50 25) '(spherical 25 50) 200001.9 "Data/thesis/test8k")

\section*{C.1.46 pde-thesis3.scm}
;; ; Turns out one of the bug fixes introduced an error. The new code with the ;; ; error got reloaded into the system, and...
(load "pde-test")
(test-5 '(rectangular 46 23) '(spherical 2346\() 190001.9\) "Data/thesis/test5j")
(test-1 '(rectangular 50 25) '(spherical 25 50) 200001.9 "Data/thesis/test1k")
(test-2 '(rectangular 50 25) '(spherical 25 50) 200001.9 "Data/thesis/test2k")
(test-3 '(rectangular 50 25) '(spherical 25 50) 200001.9 "Data/thesis/test3k")
(test-4 '(rectangular 50 25) '(spherical 25 50) 200001.9 "Data/thesis/test4k")
(test-5 '(rectangular 50 25) '(spherical 25 50) 200001.9 "Data/thesis/test5k")

\section*{C.1.47 pde-thesis4.scm}
;; In fixing a previous bug, I introduced another bug.
(load "pde-test")
;;; Here goes again:
(test-1 '(rectangular 10 5) '(spherical 5 10) 100001.9 "Data/thesis/test1a")
(test-2 '(rectangular 10 5) '(spherical 5 10) 100001.9 "Data/thesis/test2a")
(test-1 '(rectangular 14 7)' (spherical 7 14) 110001.9 "Data/thesis/test1b")
(test-2 '(rectangular 14 7) '(spherical 7 14) 110001.9 "Data/thesis/test2b")
(test-1 '(rectangular 18 9) '(spherical 9 18) 120001.9 "Data/thesis/test1c")
(test-2 '(rectangular 18 9) '(spherical 9 18) 120001.9 "Data/thesis/test2c")
(test-1 '(rectangular 22 11) '(spherical 11 22) 130001.9 "Data/thesis/test1d")
(test-2 '(rectangular 22 11) '(spherical 11 22) 130001.9 "Data/thesis/test2d")
(test-1 '(rectangular 26 13) '(spherical 13 26) 140001.9 "Data/thesis/test1e")
(test-2 '(rectangular 26 13) '(spherical 13 26) 140001.9 "Data/thesis/test2e")
(test-1 '(rectangular 30 15) '(spherical 15 30) 150001.9 "Data/thesis/test1f")
(test-2 '(rectangular 30 15) '(spherical 15 30) 150001.9 "Data/thesis/test2f")
(test-1 '(rectangular 34 17) '(spherical 17 34) 160001.9 "Data/thesis/test1g")
(test-2 '(rectangular 34 17) '(spherical 17 34) 160001.9 "Data/thesis/test2g")
(test-1 '(rectangular 38 19) '(spherical 19 38) 170001.9 "Data/thesis/test1h")
(test-2 '(rectangular 38 19) '(spherical 19 38) 170001.9 "Data/thesis/test 2 h ")
(test-1 '(rectangular 42 21) '(spherical 2142\() 180001.9\) "Data/thesis/testii")
(test-2 '(rectangular 42 21) '(spherical 21 42) 180001.9 "Data/thesis/test2i")
(test-1 '(rectangular 46 23) '(spherical 23 46) 190001.9 "Data/thesis/test1j")
(test-2 '(rectangular 46 23) '(spherical 23 46) 190001.9 "Data/thesis/test2j")
(test-1 '(rectangular 50 25) '(spherical 25 50) 200001.9 "Data/thesis/test1k")
(test-2 '(rectangular 50 25) '(spherical 25 50) 200001.9 "Data/thesis/test \(2 k\) ")

\section*{C.1.48 pde-tools.scm}
```

;;; This file provides the interface between the manifold PDE code and the FEM
;;; toolkit. These should be actual examples of constructors for PDE charts
;;; that use the FEM stuff we already have. This part can be as specific as is
;;; necessary.
;;; If we didn't care about good triangulations, then one might think that this
;;; recursive algorithm works for convex sets: Sort the nodes along one axis,
;;; and form the n-simplex with the highest n nodes. Take the vertices not in
;;; contact with the rest of the solid out, and apply recursively. This is an
;;; O(n log n) algorithm. Well, it actually doesn't seem to work, not without
;;; some tweaking. In any case, there is something better:
;; See http://www.geom.umn.edu/software/qhull/ for general triangulation
;;; algorithms; it's probably better than the Guibas/Stolfi algorithm we're
;;; using. (These algorithms usually work by requiring convexity, which we
;;; will also do.)
(declare (usual-integrations))
;;; Local tesselations depend on the fact that domains of charts are often
;;; simple shapes (like squares and discs). Focus on 2-D case for now.
(define make-rectangular-vertices
(let ((border-frac 1e-3))
(lambda (chart x-low x-high y-low y-high vcount hcount)
;; Stay away from the borders so that all the nodes lie in the chart.
(let ((x-border (* (- x-high x-low) border-frac))
(y-border (* (- y-high y-low) border-frac)))
(set! x-high (- x-high x-border))
(set! x-low (+ x-low x-border))
(set! y-high (- y-high y-border))
(set! y-low (+ y-low y-border))
;; Is this a boundary chart? If so, fix it so that boundary nodes are
;; really on the boundary:
(if (boundary-chart? chart)
(let ((level (chart:get-boundary-index chart)))
(if (= (chart:get-boundary-index chart) 0)
(if (almost-equal? level (- x-low x-border))
(set! x-low (- x-low x-border))
(set! x-high (+ x-high x-border)))
(if (almost-equal? level (- y-low y-border))
(set! y-low (- y-low y-border))
(set! y-high (+ y-high y-border)))))))
(let* ((hcount-1 (- hcount 1))
(vcount-1 (- vcount 1))
(dx (/ (- x-high x-low) hcount-1))
(dy (/ (- y-high y-low) vcount-1)))
(let next-row ((i 0) (nodes '()))
(if (< i hcount)

```
```

    (let ((x (+ x-low (* i dx))))
    (let next-col ((j 0) (nodes nodes))
                (if (< j vcount)
                            (let* ((y (+ y-low (* j dy)))
                                    (new-node (make-node (vector x y) chart)))
                            (if (or (= i 0) (= i hcount-1) (= j 0) (= j vcount-1))
                                    (node:set-local-boundary! new-node #t))
                            (next-col (+ j 1) (cons new-node nodes)))
                            (next-row (+ i 1) nodes))))
        nodes)))))(
    (define make-circular-vertices
(let ((border-frac 1e-3))
(lambda (chart x y radius radial-count angular-count)
;; Stay away from the borders so that all the nodes lie in the chart. By
;; our convention, this can't be a boundary chart, so no boundary nodes.
(set! radius (* (- 1 border-frac) radius))
(let* ((radial-count-1 (- radial-count 1))
(dr (/ radius radial-count-1))
(dt (/ (* 2 pi) angular-count)))
(let next-ray ((i 1) (nodes '()))
(if (< i radial-count)
(let ((r (* i dr)))
(let next-point ((j 0) (nodes nodes))
(if (< j angular-count)
(let* ((t (* j dt))
(new-node (make-node (vector (+ (* r (cos t)) x)
(+ (* r (sin t)) y))
chart)))
(if (= i radial-count-1)
(node:set-local-boundary! new-node \#t))
(next-point (+ j 1) (cons new-node nodes)))
(next-ray (+ i 1) nodes))))
(cons (make-node (vector x y) chart) nodes)))))))
;;; Let's use the range structures for meshing. Better back off from the edge
;;; a bit to a compact subset, though, to avoid trouble with charts that cover
;;; almost all of the manifold (such as spherical coordinates on the sphere).
(define (make-vertices chart . args)
(let ((dim (chart:dimension chart)))
(if (= dim 2)
(let ((nodes
(cond ((chart:cell-range? chart)
(apply make-rectangular-vertices
(append
(cons chart
(append-map
(lambda (interval)
(list (interval:inf interval)
(interval:sup interval)))
(cell-range:get-interval-list chart)))
(cdr (assq 'rectangular args)))))

```
```

                        ((chart:spherical-range? chart)
                        (let ((center (spherical-range:get-center chart)))
                        (apply make-circular-vertices
                        (append
                        (list chart
                        (vector-ref center 0)
                                    (vector-ref center 1)
                                    (spherical-range:get-radius chart))
                                    (cdr (assq 'spherical args))))))
                                    (else (error "Don't know how to mesh this chart!")))))
            nodes)
            (write-line '(can only handle planar regions for now!)))))
    ;;; A discretization routine. This is our interface to local FEM.
(define (fem-discretize chart source)
(let* ((nodes (list->vector (chart:get-nodes chart)))
(sparse (assemble-equations source nodes))
(n (sparse-matrix-row-count sparse))
(m (vector-length nodes))
(index-map (make-vector n \#f)))
(write-line '(,n equations generated for ,m nodes.))
;; Need to be able to translate row indices into node IDs.
(let loop ((i 0) (j 0))
(if (< i m)
(let ((node (vector-ref nodes i)))
(if (node:boundary? node)
(loop (+ i 1) j)
(begin
(vector-set! index-map j i)
(loop (+ i 1) (+ j 1)))))))
(let loop ((i 0) (result '()))
(if (< i n)
(let ((node (vector-ref nodes (vector-ref index-map i))))
(let next-term ((row (sparse-matrix-get-row sparse i))
(terms '())
(const 0))
(if (null? row)
(loop (+ i 1) (cons (make-equation node const terms) result))
(let* ((pair (car row))
(index (car pair))
(coeff (cadr pair)))
(if (= index n)
(next-term (cdr row) terms coeff)
(let ((node (vector-ref
nodes (vector-ref index-map index))))
(next-term (cdr row)
(cons (make-term node coeff) terms)
const)))))))
result))))
;;; Usually, we won't need more than just the vertices:

```
```

(define (make-no-extra-nodes complex)
(map (lambda (face) '()) (complex->faces complex)))
;;; However, it's always nice to have mid-edge nodes:
(define make-mid-edge-nodes
(let ((edge-index
(lambda (edge)
(apply symmetric->vector-index (map node:get-id edge)))))
(lambda (complex)
(let* ((nodes (complex->vertices complex))
(edges (complex->edges complex))
(n (choose (+ (length nodes) 2) 2))
(big-v (make-vector n \#f)))
;; First, assign each node a unique ID:
(let loop ((i 0) (nodes nodes))
(if (not (null? nodes))
(begin
(node:set-id! (car nodes) i)
(loop (+ i 1) (cdr nodes)))))
;; Next, construct mid-edge nodes for each edge and save them:
(for-each
(lambda (edge)
(let* ((org (car edge))
(dest (cadr edge))
(node (make-node
(vector:* 1/2
(apply vector:+ (map node:get-coords edge)))
(node:get-chart org))))
(if (and (node:local-boundary? org)
(node:local-boundary? dest))
(node:set-local-boundary! node \#t))
(vector-set! big-v (edge-index edge) node)))
edges)
;; Finally, assign edge nodes to each face of the complex:
(map
(lambda (face)
(append-map
(lambda (pair)
(let ((node (vector-ref big-v (edge-index pair))))
(if node
(list node)
,())))
(pairs face)))
(complex->faces complex))))))
(define (complex->edges complex)
(cadr (reverse complex)))

```
```

(define (complex->vertices complex)
(car (reverse complex)))
(define (complex->faces complex)
(car complex))
;;; Mesh-generation stuff: Here's an interface to Delaunay triangulation
;;; routines in 2-D.
(define (planar-triangulate nodes)
(reverse (cons nodes (delaunay-triangulation (list->vector nodes)))))
;;; Count the number of boundary nodes in a VECTOR of nodes:
(define (number-of-boundary-nodes nodes)
(let ((n (vector-length nodes)))
(let loop ((i 0) (count 0))
(if (< i n)
(if (node:boundary? (vector-ref nodes i))
(loop (+ i 1) (+ count 1))
(loop (+ i 1) count))
count))))

```

\section*{C.1.49 pde-works.scm}
;;; This stuff works!
(load "load-pde")
;; Try something a *little* different:
(write-line '(constructing domain...))
(define make-domain
    (show-time
        (lambda ()
            (pde:make-domain-without-overlaps
                disc make-vertices make-no-extra-nodes planar-triangulate
            '(rectangular 40 15) '(spherical 20 30)))))
;; \(;\) Construct elements, as usual (cheat on the integration):
(write-1ine '(constructing elements...))
(show-time
    (lambda ()
        (make-domain
        imbedded-poly-laplacian
        make-triangular-imbedded-integrator
        pde:make-imbedded-poly-basis-function)))
;;; Forming matrix:
```

(write-line '(forming matrix...))
(define mat
(show-time
(lambda ()
(combine-equations-without-overlap disc 0-function test-function))))
;;; Solve the equations:
(write-line '(relax!))
(define v
(show-time
(lambda ()
(sor mat 10000 1.9))))
;;; Get a rough picture of what this looks like:
(write-line '(getting a picture of the relative error...))
(define relative-error-picture
(show-time
(lambda ()
(manifold->grid 15 15 disc test-function v relative-error))))
(write-line '(getting a rough picture of the solution...))
(define solution-picture
(show-time
(lambda ()
(manifold->grid 15 15 disc test-function v (lambda (val ref) val))))
;;; We'll need to re-run these tests and save the numbers. For now, just a
;;; brief indication (so we know what to write).
;;; What works:
;;; 1. PDE:MAKE-DOMAIN-WITH-SMALL-OVERLAPS + COMBINE-EQUATIONS-WITHOUT-OVERLAP!
;;; 2. PDE:MAKE-DOMAIN-WITHOUT-OVERLAPS, with sufficiently many nodes.
;;; And what doesn't:
;;; 1. PDE:MAKE-DOMAIN-WITH-OVERLAPS requires generating constraints, and
;;; requires using the normal equations (which is very slow in converging).
;;; 2. PDE:MAKE-DOMAIN-WITH-LARGER-OVERLAPS generates much larger systems of
;;; equations. Again, using normal equations may be a bad idea.
;;; 3. PDE:MAKE-SIMPLE-DOMAIN doesn't do much better.

```

\section*{C.1.50 product.scm}
```

;;; Product manifolds. (Note that, in this scheme (no pun intended), RxR is
;;; not the same as R^2; they are, of course, diffeomorphic.)
(declare (usual-integrations))
;;; First, we need product charts:
(define (make-product-chart chart-1 chart-2)
(let* ((dim-1 (chart:dimension chart-1))
(dim-2 (chart:dimension chart-2))
(dim (+ dim-1 dim-2))
(coord-map-1 (chart:get-coord-map chart-1))
(coord-map-2 (chart:get-coord-map chart-2))
(inverse-map-1 (chart:get-inverse-map chart-1))
(inverse-map-2 (chart:get-inverse-map chart-2))
(euclidean? (make-euclidean-test dim)))
(letrec
((in-domain?
(lambda (x)
(and (list? x)
(not (null? (cdr x)))
(chart:member? (car x) chart-1)
(chart:member? (cadr x) chart-2))))
(in-range?
(lambda (x)
(and (euclidean? x)
(chart:in-range? (vector-head x dim-1) chart-1)
(chart:in-range? (vector-end x dim-2) chart-2))))
(coord-map
(lambda (x)
(vector-append (coord-map-1 (product:get-arg-1 x))
(coord-map-2 (product:get-arg-2 x)))))
(inverse-map
(lambda (x)
(product:combine (inverse-map-1 (vector-head x dim-1))
(inverse-map-2 (vector-end x dim-2)))))
(transition
(lambda (VxW)
(let ((components (chart:get-components VxW))
(V (car components))
(W (cadr components)))
(let ((f (chart:make-transition-map chart-1 V))
(g (chart:make-transition-map chart-2 W)))
(lambda (x)
(vector-append (f (vector-head x dim-1))
(g(vector-end x dim-2))))))))

```
```

        (let ((new-chart (make-chart
            (+ dim-1 dim-2)
                            in-domain? in-range? coord-map inverse-map
                    transition)))
    (chart:install-extra new-chart 'product-chart (list chart-1 chart-2))
    new-chart))))
    (define (chart:get-components chart)
(chart:get-extra chart 'product-chart))
(define (chart:first-component chart)
(car (chart:get-components chart)))
(define (chart:second-component chart)
(cadr (chart:get-components chart)))
;;; This is slow, but should be sufficient for most examples we construct (such
;;; as the 2-torus and products of Euclidean spaces with other manifolds).
(define (product-manifold M1 M2)
(let ((atlas-1 (manifold:get-finite-atlas M1))
(atlas-2 (manifold:get-finite-atlas M2)))
(let ((M
(if (and atlas-1 atlas-2)
(charts->manifold (map (lambda (l) (apply make-product-chart l))
(all-pairs atlas-1 atlas-2)))
(make-general-product-manifold M1 M2))))
(manifold:install-extra M 'product-manifold (list M1 M2))
M)))
(define (make-general-product-manifold M1 M2)
(let ((general-find-1 (manifold:get-general-chart-finder M1))
(general-find-2 (manifold:get-general-chart-finder M2))
(find-chart-1 (manifold:get-chart-finder M1))
(find-chart-2 (manifold:get-chart-finder M2))
(minimize-1 (manifold:get-general-minimizer M1))
(minimize-2 (manifold:get-general-minimizer M2)))
(letrec
((find-chart
(lambda (p . predicates)
(if (null? predicates)
(let ((chart-1 (find-chart-1 (car p)))
(chart-2 (find-chart-2 (cadr p))))
(if (and chart-1 chart-2)
(make-product-chart chart-1 chart-2)
\#f))
(call-with-current-continuation
(lambda (return)
(general-find-1
(car p)
(lambda (chart-1)
(general-find-2
(cadr p)
(lambda (chart-2)
(let ((chart (make-product-chart chart-1 chart-2)))
(let valid? ((predicates predicates))
(if (null? predicates)

```
```

(return chart)
(if ((car predicates) chart)
(valid? (cdr predicates))
\#f)))\)))\)\))(

```
```

        (minimize-chart
        (lambda (p f <)
            (cadr (minimize-1
                    (car p)
                            (lambda (chart-1)
                                (minimize-2
                                (cadr p)
                                (lambda (chart-2)
                                    (let ((chart (make-product-chart chart-1 chart-2)))
                        (list chart (f chart))))
                                (lambda (x y)
                    (< (cadr x) (cadr y)))))
                    (lambda (x y)
                        (< (cadr x) (cadr y)))))))
        (get-local-atlas
        (lambda (p)
            (map make-product-chart
                (manifold:get-local-atlas M1 (car p))
                    (manifold:get-local-atlas M2 (cadr p)))))
                            (make-manifold (apply + (map manifold:dimension (list M1 M2)))
                find-chart minimize-chart get-local-atlas))))
    (define (manifold:get-components M)
(manifold:get-extra M 'product-manifold))
(define (manifold:first-component M)
(car (manifold:get-components M)))
(define (manifold:second-component M)
(cadr (manifold:get-components M)))

```

\section*{C.1.51 ranges.scm}
```

;;; Some special structures that lets us tell simple *range* shapes. This
;;; makes mesh generation for the PDE solver much easier.
(declare (usual-integrations))
;;; Spherical ranges are very useful (e.g. stereographic projection):
(define (make-spherical-range chart center radius)
(chart:install-extra chart 'spherical-range (vector center radius)))
(define (spherical-range:get-structs chart)
(chart:get-extra chart 'spherical-range))
(define (spherical-range:get-center chart)
(let ((result (spherical-range:get-structs chart)))
(if result

```
```

    (vector-ref result 0)
    #f)))
    (define (spherical-range:get-radius chart)
(let ((result (spherical-range:get-structs chart)))
(if result
(vector-ref result 1)
\#f)))
(define (chart:spherical-range? chart)
(if (spherical-range:get-structs chart)
\#t
\#f))
;;; Ranges that are n-cells are also very useful (e.g. spherical coordinates).
(define (make-cell-range chart intervals)
(chart:install-extra
chart 'cell-range (if (list? intervals)
(list->vector intervals)
intervals)))
(define (cell-range:get-structs chart)
(chart:get-extra chart 'cell-range))
(define (cell-range:get-interval chart i)
(let ((result (cell-range:get-structs chart)))
(if result
(vector result i)
\#f)))
(define cell-range:get-intervals cell-range:get-structs)
(define (cell-range:get-interval-list chart)
(let ((result (cell-range:get-intervals chart)))
(if result
(vector->list result)
\#f)))
(define (chart:cell-range? chart)
(if (cell-range:get-structs chart)
\#t
\#f))

```
;;; The interval:
(define (make-interval a b)
    (vector a b))
(define (interval:inf interval)
    (vector-ref interval 0 ))
(define (interval:sup interval)
    (vector-ref interval 1))
(define (interval:member? \(x\) interval)
    (and (real? \(x\) )
(< (interval:inf interval) \(x\) )
(< x (interval:sup interval))))

\section*{C.1.52 richardson.scm}
;;; This file plays around with Richardson extrapolation. ORDER sould be the ;;; order of the error.
```

(declare (usual-integrations))

```
;; ; Here is the slow way:
(define (richardson \(f\) order)
    (if (> order 0)
            (let ((f (richardson f (- order 1)))
                (k (expt 2 order)))
            (lambda (h)
                (/ (- (* k (f (/ h 2))) (f h)) (-k 1))))
            f))
;; Here is a quicker way:
(define (richardson-coeffs order)
    (let ( v (make-vector order 0))
            (w (make-vector order 0 )))
        (vector-set! v 0 1)
        (let loop ((i 1) (2^i 1) (from v) (to w))
            (if (< i order)
            (let ( \(\left(2^{\wedge} i+1\left(* 22^{\wedge} i\right)\right)\)
                        (i-1 (-i 1)))
                                (vector-set! to 0 (/ (vector-ref from 0 ) (- \(12^{\text {~ } i+1))) ~}\)
                                (do ((j 1 (+ j 1)))
                            ((> j i-1))
                        (vector-set! to \(j\) (/ (- (* \(2^{\wedge} i+1\) (vector-ref from (- j 1)))
                                    (vector-ref from \(j\) ))
                                    (- 2~i+1 1))))
                                (vector-set! to i (* (/ \(\left.2^{\wedge} i+1\left(-2^{\wedge} i+11\right)\right)\)
                                    (vector-ref from i-1)))
                        (loop (+ i 1) 2~i+1 to from))
            from)))
(define (quick-r forder)
    (let ( v (richardson-coeffs order)))
        (lambda (h)
            (let loop ((i 0) (h/2^i h) (sum 0.))
                (if (< i order)
                        (loop (+ i 1) (/ h/2~i 2) (+ sum (* (vector-ref vi) (f h/2^i))))
                        sum))))
```

;;; Try some numerical differentiation:
;;; An observation that may make this better: When the derivative is computed
;;; using the central difference (instead of forward or backward difference),
;;; only the even-degree terms in the difference quotient survive.
;;; Also, try using roots of unity (complex arithmetic) to get rid of
;;; higher-order terms. Better yet, use contour integrals!
(define (make-differentiator v+ v- v* h order)
(let ((v (richardson-coeffs order)))
(lambda (f)
(let ((diff-quo
(lambda (x)
(lambda (h)
(v* (/ (* 2 h)) (v- (f (+ x h)) (f (- x h))))))))
(lambda (x)
(let ((f (diff-quo x)))
(let loop ((i 1)
(h/2^i (/ h 2))
(sum (v* (vector-ref v 0) (f h))))
(if (< i order)
(loop (+ i 1) (/ h/2^i 2)
(v+ sum (v* (vector-ref v i) (f h/2^i))))
sum)))())),
;(define derivative (make-differentiator + - * 1e-5 3))
;;; For now, let's use numerical differentiation with Richardson extrapolation,
;;; and use charts to represent tangent vectors. Later, we should provide ways
;;; of automagically switching between different representations (e.g. among
;;; different charts and/or between chart-vector representations and imbedding
;;; representations (imbeddings *are* very useful)).
(define (make-numerical-differential-operator h n scale)
(let ((d (make-differentiator vector:+ vector:- vector:* h n)))
(lambda (f)
(lambda (x)
(lambda (v)
(let* ((change-factor (/ (vector:magnitude v) scale))
(w (if (zero? change-factor)
v
(vector:* (/ change-factor) v)))
(on-path (lambda (s) (f (vector:+ x (vector:* s w))))))
(vector:* change-factor ((d on-path) 0)))))))
;;; Is this basically equivalent to Ridder's method, described in _Numerical
;;; Recipes in Fortran, Second Edition_?
;;; Minimize a convex function:
(define (minimize-convex-function f a b n)
(let loop ((n n) (a a) (b b))
(if (> n 0)
(let ((x1 (+ (* 2/3 a) (/ b 3)))
(x2 (+ (/ a 3) (* 2/3 b))))
(let ((y1 (f x1))

```
```

                (y2 (f x2)))
                (cond ((< y1 y2) (loop (- n 1) a x2))
                    ((> y1 y2) (loop (- n 1) x1 b))
                    (else (loop (- n 1) x1 x2)))))
    (let ((result (/ (+ a b) 2)))
        (list result (f result))))))
    ```
```

;;; Let's define the differential operator:

```
;;; Let's define the differential operator:
(define diff
(define diff
    (let ((order 5)
    (let ((order 5)
            (min-step-size 1e-5)
            (min-step-size 1e-5)
            (max-step-size 1e-1)
            (max-step-size 1e-1)
            (search-depth 11)
            (search-depth 11)
            (scale 1))
            (scale 1))
        (make-numerical-differential-operator
        (make-numerical-differential-operator
            (let (Cresult
            (let (Cresult
                    (minimize-convex-function
                    (minimize-convex-function
                        (lambda (h)
                        (lambda (h)
                            (let ((d (make-numerical-differential-operator h order scale)))
                            (let ((d (make-numerical-differential-operator h order scale)))
                        (abs (- (vector-ref (()d (lambda (x)
                        (abs (- (vector-ref (()d (lambda (x)
                                    (vector (sqrt (vector-ref x 0)))))
                                    (vector (sqrt (vector-ref x 0)))))
                                    (vector 1))
                                    (vector 1))
                                    (vector 1))
                                    (vector 1))
                                    0)
                                    0)
                                    1/2))))
                                    1/2))))
                    min-step-size max-step-size search-depth)))
                    min-step-size max-step-size search-depth)))
            (write-line '(richardson order = ,order))
            (write-line '(richardson order = ,order))
            (write-line '(h = ,(car result)))
            (write-line '(h = ,(car result)))
            (write-line '(error = ,(cadr result)))
            (write-line '(error = ,(cadr result)))
            (car result))
            (car result))
        order
        order
        scale)))
        scale)))
;(define diff-old (make-numerical-differential-operator 1e-5 3))
```

;(define diff-old (make-numerical-differential-operator 1e-5 3))

```

\section*{C.1.53 rigid-body.scm}
```

;; This uses ScmUtils to compute the usual axisymmetric top:

```
```

(set! *ode-integration-method* 'bulirsch-stoer)

```
(set! *ode-integration-method* 'bulirsch-stoer)
(define (my-ode-advancer v x dt tol)
    (let ((dt/2 (/ dt 2.))
            (dt/6 (/ dt 6.)))
        (let* ((F1 (v x))
            (F2 (v (vector:+ x (vector:scalar*vector dt/2 F1))))
            (F3 (v (vector:+ x (vector:scalar*vector dt/2 F2))))
            (F4 (v (vector:+ x (vector:scalar*vector dt F3)))))
            (vector:+ (vector:scalar*vector dt/6
                                    (vector:+ F1
                                    (vector:scalar*vector 2. F2)
                                    (vector:scalar*vector 2. F3)
                                    F4))
                                    x)))(
```

```
(define (rigid-body-evolver a b c x0 t-final dt tol)
    (let ((v.field (compiled-rigid-body a b c)))
        (let loop ((x x0) (results '()))
            (if (< (state->t x) t-final)
                (let ((new-x (ode-advancer v.field x dt tol)))
                        (loop new-x (cons x results)))
                    results))))
;;; The vector field:
(define (rigid-body-sysder a b c)
    (lagrangian->state-derivative
        (t-rigid-body a b c)))
(newline)
(display "*** Compiling state derivative")
(define compiled-rigid-body
        (show-time
        (lambda ()
            (compile-sysder 3 rigid-body-sysder))))
;;; Useful functions:
;((STATE->L-SPACE A B C) STATE) => angular momentum in the reference frame.
;((STATE->L-BODY A B C) STATE) => angular momentum in the body frame.
; (RELATIVE-ERROR VALUE REFERENCE-VALUE) => error of VALUE relative to
; REFERENCE-VALUE.
;;; Initial conditions from the book:
;;; In Euler coordinates:
;;; q0 = #(1 0 0), qdot0 = #(0.1 0.1 0.1).
;;; Step size = 0.01, and final time is 100.0.
;;; Maximum local truncation error is 1.0e-12.
;;; A=1, B=sqrt(2), C=2.
;;; Note that the order of components needs to be switched when using these
;;; initial conditions with the manifold stuff.
(newline)
(display "*** Evolving trajectories")
(define result
    (show-time
        (lambda ()
            (rigid-body-evolver 1 (sqrt 2) 2
                                    (->state 0. (vector 1. 0. 0.) (vector -. 1 -.01 -.01))
                                    100.
                                    .01 1.0e-12))))
```

```
(newline)
(display "*** Saving results")
(let ((port (open-output-file "rigid-reg.data")))
    (for-each
```

```
        (lambda (state)
            (display (state->t state) port)
            (display " " port)
            (display (state->q state) port)
            (display " " port)
            (display (state->qdot state) port)
            (newline port))
            (sort results (lambda (x y) (< (state->t x) (state->t y)))))
    (close-output-port port))
;;; Directly from the text:
(define (do-it A B C state0 final-t dt tol)
    (let ((dstate (compiled-rigid-body A B C))
            (LO ((state->L-space A B C) state0))
            (EO ((T-rigid-body A B C) state0)))
        (let ((LxO (vector-ref LO 0))
                (Ly0 (vector-ref LO 1))
                (LzO (vector-ref LO 2)))
            (let loop ((state state0))
            (if (< (state->t state) final-t)
                (let ((ns (ode-advancer dstate state dt tol)))
                        (let ((L ((state->L-space A B C) ns))
                            (E ((T-rigid-body A B C) ns))
                            (t (state->t ns)))
                            (let ((Lx (vector-ref L 0))
                            (Ly (vector-ref L 1))
                            (Lz (vector-ref L 2)))
                                    (let ((error-Lx (relative-error Lx Lx0))
                                    (error-Ly (relative-error Ly Ly0))
                                    (error-Lz (relative-error Lz Lz0))
                                    (error-E (relative-error E EO)))
                                    (plot-point window t error-Lx)
                                    (plot-point window t error-Ly)
                            (plot-point window t error-Lz)
                            (plot-point window t error-E)
                            (loop ns)))))))))
#|
;;; Comments by GJS:
;;; For QC Runge-Kutta 4
(set! *ode-integration-method* (quote qcrk4))
(define window (frame 0. 100. -1.e-12 1.e-12))
;;; For bulirsch-stoer
(set! *ode-integration-method* (quote bulirsch-stoer))
(define window (frame 0. 100. -1.e-13 1.e-13))
;;; Comes by the coordinate singularity several times
(do-it 1. (sqrt 2.) 2.
    (->state 0.0
            (vector 1. 0. 0.)
            (vector 0.1 0.1 0.1))
    100.0
    0.01
    1.0e-12)
```

```
;;; Whizzing rather close to a singularity
(do-it 1. (sqrt 2.) 2.
    (->state 0.0
                                    (vector 1. 0. 0.)
                    (vector -0.1 -. .01 -.01))
    100.0
    0.01
    1.0e-12)
|#
```


## C.1.54 rigid-compute.scm

```
(load "rigid")
```

;; Compute some expressions:
(for-each
(lambda (make-sysder filename)
(let ((port (open-output-file filename)))
(pp (traditional->correct-order
(vector-tail
(show-time
(lambda ()
(*sysder-simplify*
((make-sysder 'a 'b 'c) rigid-qqdot))))
1))
port)
(close-output-port port)))
(list rigid-sysder-0 rigid-sysder-1 rigid-sysder-2 rigid-sysder-3)
(list "rigid-field-0" "rigid-field-1" "rigid-field-2" "rigid-field-3"))

## C.1.55 rigid-errors.scm

```
(define (check-relative-error list)
    (let ((ref (car list)))
        (write-line '(reference value = ,ref))
        (map (lambda (val) (relative-error val ref)) list)))
(define (check-absolute-error list)
    (let ((ref (car list)))
        (write-line '(reference value = ,ref))
        (map (lambda (val) (abs (- val ref))) list)))
#|
(define reg-list
    (show-time
        (lambda ()
            (read-regular-file "rigid-reg.data"))))
(define reg-e-list
    (show-time
        (lambda ()
            (map (t-rigid-body 1. (sqrt 2.) 2.) regular))))
```

```
(define reg-l-list
    (show-time
    (lambda ()
                (map (state->L-space 1. (sqrt 2) 2.) regular))))
(define reg-l1-list (map vector-first reg-1-list))
(define reg-12-list (map vector-second reg-l-list))
(define reg-l3-list (map vector-third reg-l-list))
(define reg-e-errors (check-absolute-error reg-e-list))
(define reg-l1-errors (check-absolute-error reg-l1-list))
(define reg-12-errors (check-absolute-error reg-l2-list))
(define reg-l3-errors (check-absolute-error reg-13-list))
(define man-list
    (show-time
        (lambda ()
            (read-manifold-file "rigid-man.data"))))
(define man-e-list
    (show-time
        (lambda ()
            (map (t-rigid-body 1. (sqrt 2) 2.) manifold))))
(define man-l-list
    (show-time
        (lambda ()
            (map (state->L-space 1. (sqrt 2) 2.) man-list))))
(define man-l1-list (map vector-first man-l-list))
(define man-12-list (map vector-second man-l-list))
(define man-l3-list (map vector-third man-l-list))
(define man-e-errors (check-absolute-error man-e-list))
(define man-l1-errors (check-absolute-error man-l1-list))
(define man-l2-errors (check-absolute-error man-l2-list))
(define man-13-errors (check-absolute-error man-13-1ist))
|#
```


## C.1.56 rigid-fields.scm

```
;;; These vector fields are machine-generated:
(define (rigid-field-0 a b c x v)
    (let ((psi (vector-ref x 0))
            (theta (vector-ref x 1))
            (phi (vector-ref x 2))
            (psidot (vector-ref v 0))
            (thetadot (vector-ref v 1))
            (phidot (vector-ref v 2)))
        (vector psidot thetadot phidot
                            (/ (+ (* (cos theta) (+ (* (cos theta) (+ (* (cos psi) (+ (* (cos
                    psi) (+ (* (cos psi) (+ (* (cos psi) (+ (* a b c phidot thetadot)
                        (* (expt b 2) c phidot thetadot))) (* (sin theta) (sin psi) (+ (*
                    -1 (expt a 2) c (expt phidot 2)) (* (expt b 2) c (expt phidot
                    2)))))) (* (expt (sin psi) 2) (+ (* a (+ (* a c phidot thetadot)
```

(* 2 b c phidot thetadot))) (* (expt b 2) c phidot thetadot))) (* -1 b (expt c 2) phidot thetadot))) (* (sin theta) (sin psi) (+ (* (expt (sin psi) 2) (+ (* -1 (expt a 2) c (expt phidot 2)) (* (expt b 2) c (expt phidot 2)))) (* a (expt c 2) (expt phidot 2)) (* - 1 b (expt c 2) (expt phidot 2))))) (* (expt (sin psi) 2) (+ (* (expt (sin psi) 2) a (+ (* a c phidot thetadot) (* b c phidot thetadot))) (*-1 a (expt c 2) phidot thetadot))))) (* (cos psi) (+ (* (cos psi) (+ (* (cos psi) (+ (* (cos psi) (+ (* - 1 a b c psidot thetadot) (* (expt b 2) c psidot thetadot)) ) (* (sin theta) ( $\sin \mathrm{psi}$ ) (+ (* -1 (expt a 2) c phidot psidot) (* (expt b 2) c phidot psidot)) )) (* (expt (sin psi) 2) (+ (* a (+ (* a c psidot thetadot) (* -2 b c psidot thetadot)) ) (* (expt b 2) c psidot thetadot)) ) (* -1 b (expt c 2) psidot thetadot))) (* (sin theta) (sin psi) (+ (* (expt (sin psi) 2) (+ (* -1 (expt a 2) c phidot psidot) (* (expt b 2) c phidot psidot))) (* a (expt c 2) phidot psidot) (* -1 b (expt c 2) phidot psidot))))) (* (expt (sin psi) 2) (+ (* (expt (sin psi) 2) a (+ (* a c psidot thetadot) (* -1 b c psidot thetadot))) (*-1 a (expt c 2) psidot thetadot))))) (* (cos psi) (+ (* (cos psi) (+ (* (cos psi) (+ (* (cos psi) (+ (* (cos psi) (+ (* (cos psi) (expt (sin theta) 2) a (+ (* -1 a b phidot thetadot) (* (expt b 2) phidot thetadot))) (* (sin theta) (+ (* (expt (sin theta) 2) (sin psi) a (+ (*-1 a b (expt phidot 2)) (* (expt b 2) (expt phidot 2)))) (* (sin psi) a (+ (* a b (expt thetadot 2)) (* -1 (expt b 2) (expt thetadot 2))))))) (* (expt (sin theta) 2) (+ (* (expt (sin psi) 2) a (+ (* - 1 a b phidot thetadot) (* (expt b 2) phidot thetadot))) (* a b c phidot thetadot)) )) (* (sin theta) (+ (* (expt (sin theta) 2) (expt (sin psi) 3) a (+ (* -2 a b (expt phidot 2)) (* 2 (expt b 2) (expt phidot 2)) ) (* (expt (sin psi) 3) a (+ (* 2 a b (expt thetadot 2)) (* -2 (expt b 2) (expt thetadot 2))))))) ) (* (expt (sin theta) 2) (expt (sin psi) 2) (+ (* (expt (sin psi) 2) a (+ (* a b phidot thetadot) (* - (expt b 2) phidot thetadot))) (* 2 a b c phidot thetadot)) )) (* (sin theta) (+ (* (expt (sin theta) 2) (expt (sin psi) 5) a (+ (* - 1 a b (expt phidot 2)) (* (expt b 2) (expt phidot 2))) ) (* (expt (sin psi) 5) a (+ (* a b (expt thetadot 2)) (* -1 (expt b 2) (expt thetadot 2))))))) (* (expt (sin theta) 2) (expt (sin psi) 4) (+ (* (expt (sin psi) 2) a (+ (* a b phidot thetadot) (* -1 (expt b 2) phidot thetadot))) (* a b c phidot thetadot)))) (+ (* (expt (cos psi) 2) (+ (* (expt (cos psi) 2) (sin theta) a b c) (* 2 ( $\sin$ theta) (expt (sin psi) 2) a b c))) (* ( $\sin$ theta) (expt (sin psi) 4) a b c))) (/ (+ (* (cos theta) (+ (* (sin theta) (+ (* (expt (cos psi) 2) a (+ (* a (expt phidot 2)) (* -1 c (expt phidot 2)) ) ) (* (expt (sin psi) 2) b (+ (* b (expt phidot 2)) (* -1 c (expt phidot 2)))) ) (* (cos psi) (sin psi) (+ (* a (+ (* -1 a phidot thetadot) (* c phidot thetadot)) ) (* b (+ (* b phidot thetadot) (* -1 c phidot thetadot)))))) (* (sin theta) (+ (* (expt (cos psi) 2) (+ (*-1 (expt (cos psi) 2) a b phidot psidot) (* -2 (expt (sin psi) 2) a b phidot psidot) (* a (+ (* a phidot psidot) (* -1 c phidot psidot)))) (* (expt (sin psi) 2) (+ (*-1 (expt (sin psi) 2) a b phidot psidot) (* b (+ (* b phidot psidot) (*-1 c phidot psidot)))))) (* (cos psi) (sin psi) (+ (* a (+ (* -1 a psidot thetadot) (* c psidot thetadot)) ) (* b (+ (* b psidot thetadot) (* -1 c psidot thetadot))))) ) (+ (* (expt (cos psi) 2) (+ (* (expt (cos psi) 2) a b) (* 2 (expt (sin psi) 2) a b))) (* (expt (sin psi) 4) a b))) (/ (+ (* (cos theta) (+ (* (cos psi) (+ (* (cos psi) (+ (* (cos psi) (+ (* (cos psi) (+ (*-1 a b phidot thetadot) (* - (expt b 2) phidot thetadot))) (* (sin theta) (sin psi) (+ (* (expt a 2) (expt phidot 2)) (* -1 (expt b 2) (expt phidot 2) ) ) ) ) (* (expt (sin psi) 2) (+ (* a (+ (* -1 a phidot
thetadot) (* -2 b phidot thetadot))) (* - (expt b 2) phidot thetadot)) ) (* b c phidot thetadot))) (* (sin theta) (sin psi) (+ (* (expt (sin psi) 2) (+ (* (expt a 2) (expt phidot 2)) (* (expt b 2) (expt phidot 2)))) (* -1 a c (expt phidot 2)) (* b c (expt phidot 2))))) (* (expt (sin psi) 2) (+ (* (expt (sin psi) 2) a (+ (* -1 a phidot thetadot) (* -1 b phidot thetadot))) (* a c phidot thetadot)))) (* (cos psi) (+ (* (cos psi) (+ (* (cos psi) (+ (* (cos psi) (+ (* a b psidot thetadot) (* -1 (expt b 2) psidot thetadot)) ) (* (sin theta) (sin psi) (+ (* (expt a 2) phidot psidot) (* -1 (expt b 2) phidot psidot)))) (* (expt (sin psi) 2) (+ (* a (+ (* -1 a psidot thetadot) (* 2 b psidot thetadot))) (* -1 (expt b 2) psidot thetadot))) (* b c psidot thetadot))) (* (sin theta) (sin psi) (+ (* (expt (sin psi) 2) (+ (* (expt a 2) phidot psidot) (* -1 (expt b 2) phidot psidot))) (* -1 a c phidot psidot) (* b c phidot psidot)))) (* (expt (sin psi) 2) (+ (* (expt (sin psi) 2) a (+ (* -1 a psidot thetadot) (* b psidot thetadot))) (* a c psidot thetadot)) ) (+ (* (expt (cos psi) 2) (+ (* (expt (cos psi) 2) (sin theta) a b) (* 2 (sin theta) (expt (sin psi) 2) a b))) (* (sin theta) (expt (sin psi) 4) a b)))))
(define (rigid-field-1 a b c x v)
(let ((psi (vector-ref x 0))
(theta (vector-ref x 1))
(phi (vector-ref x 2))
(psidot (vector-ref v 0))
(thetadot (vector-ref v 1))
(phidot (vector-ref v 2)))
(vector psidot thetadot phidot
(/ (+ (* (cos theta) (+ (* (cos theta) (+ (* (cos psi) (+ (* (cos psi) (+ (* (cos psi) (+ (* (cos psi) (+ (* a b c phidot thetadot) (* (expt b 2) c phidot thetadot))) (* (sin theta) (sin psi) (+ (* -1 (expt a 2) c (expt phidot 2)) (* (expt b 2) c (expt phidot 2))))) (* (expt (sin psi) 2) (+ (* a (+ (* a c phidot thetadot) (* 2 b c phidot thetadot))) (* (expt b 2) c phidot thetadot))) (* -1 b (expt c 2) phidot thetadot)) ) (* (sin theta) (sin psi) (+ (* (expt (sin psi) 2) (+ (* -1 (expt a 2) c (expt phidot 2)) (* (expt b 2) c (expt phidot 2)))) (* a (expt c 2) (expt phidot 2)) (* -1 b (expt c 2) (expt phidot 2))))) (* (expt (sin psi) 2) (+ (* (expt (sin psi) 2) a (+ (* a c phidot thetadot) (* b c phidot thetadot))) (* -1 a (expt c 2) phidot thetadot)))) (* (cos psi) (+ (* (cos psi) (+ (* (cos psi) (+ (* (cos psi) (+ (* -1 a b c psidot thetadot) (* (expt b 2) c psidot thetadot)) ) (* (sin theta) (sin psi) (+ (*-1 (expt a 2) c phidot psidot) (* (expt b 2) c phidot psidot)))) (* (expt (sin psi) 2) (+ (* a (+ (* a c psidot thetadot) (* -2 b c psidot thetadot)) ) (* (expt b 2) c psidot thetadot)) ) (*-1 b (expt c 2) psidot thetadot))) (* (sin theta) (sin psi) (+ (* (expt (sin psi) 2) (+ (* -1 (expt a 2) c phidot psidot) (* (expt b 2) c phidot psidot))) (* a (expt c 2) phidot psidot) (* -1 b (expt c 2) phidot psidot)))) ) (* (expt (sin psi) 2) (+ (* (expt (sin psi) 2) a (+ (* a c psidot thetadot) (* -1 b c psidot thetadot))) (*-1 a (expt c 2) psidot thetadot))))) (* (cos psi) (+ (* (cos psi) (+ (* (cos psi) (+ (* (cos psi) (+ (* (cos psi) (+ (* (cos psi) (expt (sin theta) 2) a (+ (* -1 a b phidot thetadot) (* (expt b 2) phidot thetadot))) (* (sin theta) (+ (* (expt (sin theta) 2) (sin psi) a (+ (* -1 a b (expt phidot 2)) (* (expt b 2) (expt phidot 2))) ) (* (sin psi) a (+ (* a b (expt thetadot 2)) (*-1 (expt b 2) (expt thetadot 2)))))))) (* (expt (sin theta) 2) (+ (* (expt (sin psi) 2) a (+ (* -1 a b phidot thetadot) (* (expt b 2) phidot thetadot))) (* a b c phidot
thetadot)) )) (* (sin theta) (+ (* (expt (sin theta) 2) (expt (sin psi) 3) a (+ (* -2 a b (expt phidot 2)) (* 2 (expt b 2) (expt phidot 2)) ) (* (expt (sin psi) 3) a (+ (* 2 a b (expt thetadot 2)) (* $-2(\operatorname{expt} \mathrm{~b} 2)(\operatorname{expt}$ thetadot 2)))))))) (* (expt (sin theta) 2) (expt (sin psi) 2) (+ (* (expt (sin psi) 2) a (+ (* a b phidot thetadot) (*-1 (expt b 2) phidot thetadot))) (* 2 a b c phidot thetadot))) ) (* (sin theta) (+ (* (expt (sin theta) 2) (expt (sin psi) 5) a (+ (* -1 a b (expt phidot 2)) (* (expt b 2) (expt phidot 2))) ) (* (expt (sin psi) 5) a (+ (* a b (expt thetadot 2)) (* -1 (expt b 2) (expt thetadot 2))))))) (* (expt (sin theta) 2) (expt (sin psi) 4) (+ (* (expt (sin psi) 2) a (+ (* a b phidot thetadot) (* -1 (expt b 2) phidot thetadot))) (* a b c phidot thetadot)))) (+ (* (expt (cos psi) 2) (+ (* (expt (cos psi) 2) (sin theta) a b c) (* $2(\sin$ theta) ( $\operatorname{expt}(\sin p s i) 2) a b c)))$ (* (sin theta) (expt (sin psi) 4) a b c))) (/ (+ (* (cos theta) (+ (* (sin theta) (+ (* (expt (cos psi) 2) a (+ (* a (expt phidot 2)) (* -1 c (expt phidot 2)) ) (* (expt (sin psi) 2) b (+ (* b (expt phidot 2)) (* -1 c (expt phidot 2)) )) ) (* (cos psi) (sin psi) (+ (* a (+ (* a phidot thetadot) (* c phidot thetadot)) ) (* b (+ (* b phidot thetadot) (* -1 c phidot thetadot)))))) ) (* (sin theta) (+ (* (expt (cos psi) 2) (+ (* - 1 (expt (cos psi) 2) a b phidot psidot) (* - 2 (expt (sin psi) 2) a b phidot psidot) (* a (+ (* a phidot psidot) (* -1 c phidot psidot))))) (* (expt (sin psi) 2) (+ (*-1 (expt (sin psi) 2) a b phidot psidot) (* b (+ (* b phidot psidot) (* - 1 c phidot psidot)))))) (* (cos psi) (sin psi) (+ (* a (+ (* -1 a psidot thetadot) (* c psidot thetadot)) ) (* b (+ (* b psidot thetadot) (*-1 c psidot thetadot))))) (+ (* (expt (cos psi) 2) (+ (* (expt (cos psi) 2) a b) (* $2(\operatorname{expt}(\sin p s i) 2)$ a b))) (* (expt (sin psi) 4) a b))) (/ (+ (* (cos theta) (+ (* (cos psi) (+ (* (cos psi) (+ (* (cos psi) (+ (* (cos psi) (+ (* -1 a b phidot thetadot) (* -1 (expt b 2) phidot thetadot))) (* (sin theta) (sin psi) (+ (* (expt a 2) (expt phidot 2)) (* -1 (expt b 2) (expt phidot 2)) )) ) (* (expt (sin psi) 2) (+ (* a (+ (* -1 a phidot thetadot) (* -2 b phidot thetadot))) (* -1 (expt b 2) phidot thetadot)) ) (* b c phidot thetadot))) (* (sin theta) (sin psi) (+ (* (expt (sin psi) 2) (+ (* (expt a 2) (expt phidot 2)) (* (expt b 2) (expt phidot 2)))) (* -1 a c (expt phidot 2)) (* b c (expt phidot 2))))) (* (expt (sin psi) 2) (+ (* (expt (sin psi) 2) a (+ (*-1 a phidot thetadot) (*-1 b phidot thetadot))) (* a c phidot thetadot))) ) (* (cos psi) (+ (* (cos psi) (+ (* (cos psi) (+ (* (cos psi) (+ (* a b psidot thetadot) (* -1 (expt b 2) psidot thetadot)) ) (* (sin theta) (sin psi) (+ (* (expt a 2) phidot psidot) (* -1 (expt b 2) phidot psidot)))) (* (expt (sin psi) 2) (+ (* a (+ (* -1 a psidot thetadot) (* 2 b psidot thetadot))) (* -1 (expt b 2) psidot thetadot))) (* b c psidot thetadot))) (* (sin theta) (sin psi) (+ (* (expt (sin psi) 2) (+ (* (expt a 2) phidot psidot) (* -1 (expt b 2) phidot psidot))) (* -1 a c phidot psidot) (* b c phidot psidot)))) (* (expt (sin psi) 2) (+ (* (expt (sin psi) 2) a (+ (* -1 a psidot thetadot) (* b psidot thetadot))) (* a c psidot thetadot)) ) (+ (* (expt (cos psi) 2) (+ (* (expt (cos psi) 2) (sin theta) a b) (* 2 (sin theta) (expt (sin psi) 2) a b))) (* (sin theta) (expt (sin psi) 4) a b)))))

```
(define (rigid-field-2 a b c x v)
    (let ((psi (vector-ref x 0))
            (theta (vector-ref x 1))
            (phi (vector-ref x 2))
            (psidot (vector-ref v 0))
            (thetadot (vector-ref v 1))
```


## (vector psidot thetadot phidot

(/ (+ (* (cos theta) (+ (* (cos theta) (+ (* (cos psi) (+ (* (cos psi) (+ (* (cos psi) (+ (* (cos psi) (+ (* -1 a b c phidot thetadot) (* - (expt b 2) c phidot thetadot))) (* (sin theta) (sin psi) (+ (* - 1 (expt a 2) c (expt phidot 2)) (* (expt b 2) c (expt phidot 2))))) (* (expt (sin psi) 2) (+ (* a (+ (* -1 a c phidot thetadot) (* -2 b c phidot thetadot))) (* -1 (expt b 2) c phidot thetadot))) (* b (expt c 2) phidot thetadot))) (* (sin theta) (sin psi) (+ (* (expt (sin psi) 2) (+ (* -1 (expt a 2) c (expt phidot 2)) (* (expt b 2) c (expt phidot 2)))) (* a (expt c 2) (expt phidot 2)) (* - 1 b (expt c 2) (expt phidot 2)))))) (* (expt (sin psi) 2) (+ (* (expt (sin psi) 2) a (+ (*-1 a c phidot thetadot) (*-1 b c phidot thetadot))) (* a (expt c 2) phidot thetadot))) ) (* (cos psi) (+ (* (cos psi) (+ (* (cos psi) (+ (* (cos psi) (+ (* -1 a b c psidot thetadot) (* (expt b 2) c psidot thetadot)) ) (* (sin theta) (sin psi) (+ (* (expt a 2) c phidot psidot) (* -1 (expt b 2) c phidot psidot)))) (* (expt (sin psi) 2) (+ (* a (+ (* a c psidot thetadot) (* -2 b c psidot thetadot))) (* (expt b 2) c psidot thetadot))) (* -1 b (expt c 2) psidot thetadot)) ) (* (sin theta) (sin psi) (+ (* (expt (sin psi) 2) (+ (* (expt a 2) c phidot psidot) (* -1 (expt b 2) c phidot psidot))) (* -1 a (expt c 2) phidot psidot) (* b (expt c 2) phidot psidot)) )) (* (expt (sin psi) 2) (+ (* (expt (sin psi) 2) a (+ (* a c psidot thetadot) (* -1 b c psidot thetadot)) ) (* -1 a (expt c 2) psidot thetadot)) )) (* (cos psi) (+ (* (cos psi) (+ (* (cos psi) (+ (* (cos psi) (+ (* (cos psi) (+ (* (cos psi) (expt (sin theta) 2) a (+ (* a b phidot thetadot) (* -1 (expt b 2) phidot thetadot)) ) (* (sin theta) (+ (* (expt (sin theta) 2) (sin psi) a (+ (* -1 a b (expt phidot 2)) (* (expt b 2) (expt phidot 2)))) (* (sin psi) a (+ (* a b (expt thetadot 2)) (* -1 (expt b 2) (expt thetadot 2) )) )) )) (* (expt (sin theta) 2) (+ (* (expt (sin psi) 2) a (+ (* a b phidot thetadot) (* -1 (expt b 2) phidot thetadot)) ) (* -1 a b c phidot thetadot))))) (* (sin theta) (+ (* (expt (sin theta) 2) (expt (sin psi) 3) a (+ (* -2 a b (expt phidot 2)) (* 2 (expt b 2) (expt phidot 2)))) (* (expt (sin psi) 3) a (+ (* 2 a b (expt thetadot 2)) (* -2 (expt b 2) (expt thetadot 2)) )) )) ) (* (expt (sin theta) 2) (expt (sin psi) 2) (+ (* (expt (sin psi) 2) a (+ (* -1 a b phidot thetadot) (* (expt b 2) phidot thetadot))) (* -2 a b c phidot thetadot))))) (* (sin theta) (+ (* (expt (sin theta) 2) (expt (sin psi) 5) a (+ (* -1 a b (expt phidot 2)) (* (expt b 2) (expt phidot 2)))) (* (expt (sin psi) 5) a (+ (* a b (expt thetadot 2)) (* -1 (expt b 2) (expt thetadot 2))))) )) (* (expt (sin theta) 2) (expt (sin psi) 4) (+ (* (expt (sin psi) 2) a (+ (* -1 a b phidot thetadot) (* (expt b 2) phidot thetadot)) ) (* -1 a b c phidot thetadot))) (+ (* (expt (cos psi) 2) (+ (* (expt (cos psi) 2) (sin theta) a b c) (* 2 (sin theta) (expt (sin psi) 2) a b c))) (* (sin theta) (expt (sin psi) 4) a b c)) ) (/ (+ (* (cos theta) (+ (* (sin theta) (+ (* (expt (cos psi) 2) a (+ (* a (expt phidot 2)) (* -1 c (expt phidot 2)))) (* (expt (sin psi) 2) b (+ (* b (expt phidot 2)) (* -1 c (expt phidot 2))))) (* (cos psi) (sin psi) (+ (* a (+ (* a phidot thetadot) (* -1 c phidot thetadot)) ) (* b (+ (* -1 b phidot thetadot) (* c phidot thetadot)))))) (* (sin theta) (+ (* (expt (cos psi) 2) (+ (* (expt (cos psi) 2) a b phidot psidot) (* 2 (expt (sin psi) 2) a b phidot psidot) (* a (+ (* -1 a phidot psidot) (* c phidot psidot)))) (* (expt (sin psi) 2) (+ (* (expt (sin psi) 2) a b phidot psidot) (* b (+ (* -1 b phidot psidot) (*
c phidot psidot))))) ) (* (cos psi) (sin psi) (+ (* a (+ (* -1 a psidot thetadot) (* c psidot thetadot))) (* b (+ (* b psidot thetadot) (* -1 c psidot thetadot))))) ) (+ (* (expt (cos psi) 2) (+ (* (expt (cos psi) 2) a b) (* $2(\operatorname{expt}(\sin p s i) 2)$ a b))) (* (expt (sin psi) 4) a b))) (/ (+ (* (cos theta) (+ (* (cos psi) (+ (* (cos psi) (+ (* (cos psi) (+ (* (cos psi) (+ (* -1 a b phidot thetadot) (*-1 (expt b 2) phidot thetadot))) (* (sin theta) (sin psi) (+ (* -1 (expt a 2) (expt phidot 2)) (* (expt b 2) (expt phidot 2)))) ) (* (expt (sin psi) 2) (+ (* a (+ (* -1 a phidot thetadot) (* -2 b phidot thetadot))) (* -1 (expt b 2) phidot thetadot)) ) (* b c phidot thetadot))) (* (sin theta) (sin psi) (+ (* (expt (sin psi) 2) (+ (* -1 (expt a 2) (expt phidot 2)) (* (expt b 2) (expt phidot 2)))) (* a c (expt phidot 2)) (* - 1 b c (expt phidot 2))))) (* (expt (sin psi) 2) (+ (* (expt (sin psi) 2) a (+ (* -1 a phidot thetadot) (* -1 b phidot thetadot))) (* a c phidot thetadot)))) (* (cos psi) (+ (* (cos psi) (+ (* (cos psi) (+ (* (cos psi) (+ (* -1 a b psidot thetadot) (* (expt b 2) psidot thetadot)) ) (* (sin theta) (sin psi) (+ (* (expt a 2) phidot psidot) (* - 1 (expt b 2) phidot psidot)))) (* (expt (sin psi) 2) (+ (* a (+ (* a psidot thetadot) (* -2 b psidot thetadot))) (* (expt b 2) psidot thetadot))) (* -1 b c psidot thetadot))) (* (sin theta) (sin psi) (+ (* (expt (sin psi) 2) (+ (* (expt a 2) phidot psidot) (* -1 (expt b 2) phidot psidot))) (* -1 a c phidot psidot) (* b c phidot psidot)))) (* (expt (sin psi) 2) (+ (* (expt (sin psi) 2) a (+ (* a psidot thetadot) (* -1 b psidot thetadot))) (* -1 a c psidot thetadot))) (+ (* (expt (cos psi) 2) (+ (* (expt ( $\cos \mathrm{psi}$ ) 2) ( $\sin$ theta) a b) (* 2 (sin theta) (expt (sin psi) 2) a b))) (* (sin theta) (expt (sin psi) 4) a b)))))

```
(define (rigid-field-3 a b c x v)
    (let ((psi (vector-ref x 0))
            (theta (vector-ref x 1))
            (phi (vector-ref x 2))
            (psidot (vector-ref v 0))
            (thetadot (vector-ref v 1))
            (phidot (vector-ref v 2)))
        (vector psidot thetadot phidot
            (/ (+ (* (cos theta) (+ (* (cos theta) (+ (* (cos psi) (+ (* (cos
                        psi) (+ (* (cos psi) (+ (* (cos psi) (+ (* -1 a b c phidot
                thetadot) (* -1 (expt b 2) c phidot thetadot))) (* (sin theta)
                (sin psi) (+ (* -1 (expt a 2) c (expt phidot 2)) (* (expt b 2) c
                (expt phidot 2))))) (* (expt (sin psi) 2) (+ (* a (+ (* -1 a c
                        phidot thetadot) (* -2 b c phidot thetadot))) (* -1 (expt b 2) c
                        phidot thetadot))) (* b (expt c 2) phidot thetadot))) (* (sin
                        theta) (sin psi) (+ (* (expt (sin psi) 2) (+ (* -1 (expt a 2) c
                        (expt phidot 2)) (* (expt b 2) c (expt phidot 2)))) (* a (expt c
                        2) (expt phidot 2)) (* -1 b (expt c 2) (expt phidot 2)))))) (*
                (expt (sin psi) 2) (+ (* (expt (sin psi) 2) a (+ (* - - a c phidot
                thetadot) (* -1 b c phidot thetadot))) (* a (expt c 2) phidot
                thetadot))))) (* (cos psi) (+ (* (cos psi) (+ (* (cos psi) (+ (*
                (cos psi) (+ (* -1 a b c psidot thetadot) (* (expt b 2) c psidot
                thetadot))) (* (sin theta) (sin psi) (+ (* (expt a 2) c phidot
                psidot) (* - ( (expt b 2) c phidot psidot)))) (* (expt (sin psi)
                2) (+ (* a (+ (* a c psidot thetadot) (* -2 b c psidot thetadot)))
                (* (expt b 2) c psidot thetadot))) (* -1 b (expt c 2) psidot
                thetadot))) (* (sin theta) (sin psi) (+ (* (expt (sin psi) 2) (+
                (* (expt a 2) c phidot psidot) (* -1 (expt b 2) c phidot psidot)))
            (* -1 a (expt c 2) phidot psidot) (* b (expt c 2) phidot
            psidot)))) (* (expt (sin psi) 2) (+ (* (expt (sin psi) 2) a (+ (*
```

a c psidot thetadot) (* -1 b c psidot thetadot)) ) (* - 1 a (expt c 2) psidot thetadot)))) (* (cos psi) (+ (* (cos psi) (+ (* (cos psi) (+ (* (cos psi) (+ (* (cos psi) (+ (* (cos psi) (expt (sin theta) 2) a (+ (* a b phidot thetadot) (* -1 (expt b 2) phidot thetadot)) ) (* (sin theta) (+ (* (expt (sin theta) 2) (sin psi) a (+ (* -1 a b (expt phidot 2)) (* (expt b 2) (expt phidot 2)))) (* (sin psi) a (+ (* a b (expt thetadot 2)) (* -1 (expt b 2) (expt thetadot 2) )) )) )) (* (expt (sin theta) 2) (+ (* (expt (sin psi) 2) a (+ (* a b phidot thetadot) (* -1 (expt b 2) phidot thetadot))) (*-1 a b c phidot thetadot))))) (* (sin theta) (+ (* (expt (sin theta) 2) (expt (sin psi) 3) a (+ (* -2 a b (expt phidot 2)) (* 2 (expt b 2) (expt phidot 2)))) (* (expt (sin psi) 3) a (+ (* 2 a b (expt thetadot 2) ) (* -2 (expt b 2) (expt thetadot 2) )) )) )) (* (expt (sin theta) 2) (expt (sin psi) 2) (+ (* (expt (sin psi) 2) a (+ (* -1 a b phidot thetadot) (* (expt b 2) phidot thetadot))) (* -2 a b c phidot thetadot)))) (* (sin theta) (+ (* (expt (sin theta) 2) (expt (sin psi) 5) a (+ (* -1 a b (expt phidot 2)) (* (expt b 2) (expt phidot 2)))) (* (expt (sin psi) 5) a (+ (* a b (expt thetadot 2)) (* -1 (expt b 2) (expt thetadot 2)) )) )) ) (* (expt (sin theta) 2) (expt (sin psi) 4) (+ (* (expt (sin psi) 2) a (+ (* -1 a b phidot thetadot) (* (expt b 2) phidot thetadot)) ) (* -1 a b c phidot thetadot))) (+ (* (expt (cos psi) 2) (+ (* (expt (cos psi) 2) (sin theta) a b c) (* 2 (sin theta) (expt (sin psi) 2) a b c))) (* (sin theta) (expt (sin psi) 4) a b c)) ) (/ (+ (* (cos theta) (+ (* (sin theta) (+ (* (expt (cos psi) 2) a (+ (* a (expt phidot 2)) (* -1 c (expt phidot 2)))) (* (expt (sin psi) 2) b (+ (* b (expt phidot 2)) (* -1 c (expt phidot 2))))) (* (cos psi) (sin psi) (+ (* a (+ (* a phidot thetadot) (* -1 c phidot thetadot)) ) (* b (+ (*-1 b phidot thetadot) (* c phidot thetadot)))))) (* (sin theta) (+ (* (expt (cos psi) 2) (+ (* (expt (cos psi) 2) a b phidot psidot) (* 2 (expt (sin psi) 2) a b phidot psidot) (* a (+ (*-1 a phidot psidot) (* c phidot psidot)))) (* (expt (sin psi) 2) (+ (* (expt ( $\sin \mathrm{psi}$ ) 2) a b phidot psidot) (* b (+ (* -1 b phidot psidot) (* c phidot psidot))))) ) (* (cos psi) (sin psi) (+ (* a (+ (* -1 a psidot thetadot) (* c psidot thetadot))) (* b (+ (* b psidot thetadot) (*-1 c psidot thetadot))))) ) (+ (* (expt (cos psi) 2) (+ (* (expt (cos psi) 2) a b) (* 2 (expt (sin psi) 2) a b))) (* (expt (sin psi) 4) a b))) (/ (+ (* (cos theta) (+ (* (cos psi) (+ (* (cos psi) (+ (* (cos psi) (+ (* (cos psi) (+ (* -1 a b phidot thetadot) (* -1 (expt b 2) phidot thetadot))) (* (sin theta) (sin psi) (+ (* -1 (expt a 2) (expt phidot 2)) (* (expt b 2) (expt phidot 2))))) (* (expt (sin psi) 2) (+ (* a (+ (* -1 a phidot thetadot) (* -2 b phidot thetadot))) (* -1 (expt b 2) phidot thetadot)) ) (* b c phidot thetadot))) (* (sin theta) (sin psi) (+ (* (expt (sin psi) 2) (+ (* -1 (expt a 2) (expt phidot 2)) (* (expt b 2) (expt phidot 2)))) (* a c (expt phidot 2)) (* -1 b c (expt phidot 2))))) (* (expt (sin psi) 2) (+ (* (expt (sin psi) 2) a (+ (* -1 a phidot thetadot) (* -1 b phidot thetadot)) ) (* a c phidot thetadot)))) (* (cos psi) (+ (* (cos psi) (+ (* (cos psi) (+ (* (cos psi) (+ (* -1 a b psidot thetadot) (* (expt b 2) psidot thetadot)) ) (* (sin theta) (sin psi) (+ (* (expt a 2) phidot psidot) (* -1 (expt b 2) phidot psidot)))) (* (expt (sin psi) 2) (+ (* a (+ (* a psidot thetadot) (* -2 b psidot thetadot))) (* (expt b 2) psidot thetadot))) (* -1 b c psidot thetadot))) (* (sin theta) (sin psi) (+ (* (expt (sin psi) 2) (+ (* (expt a 2) phidot psidot) (* -1 (expt b 2) phidot psidot))) (* -1 a c phidot psidot) (* b c phidot psidot)))) (* (expt (sin psi) 2) (+ (* (expt (sin psi) 2) a (+ (* a psidot thetadot) (* -1 b psidot thetadot))) (*

```
-1 a c psidot thetadot)))) (+ (* (expt (cos psi) 2) (+ (* (expt
(cos psi) 2) (sin theta) a b) (* 2 (sin theta) (expt (sin psi) 2)
a b))) (* (sin theta) (expt (sin psi) 4) a b)))))
```


## C.1.57 rigid-read.scm

```
;;; Generate some plots from data! Of course, the two runs are using different
;;; formats, which makes it kind of hard...
(declare (usual-integrations))
(define (eof? port)
    (eof-object? (peek-char port)))
(define (read-number port)
    (let loop ((c (read-char port)) (string '()))
        (if (or (eof-object? c) (memq c '(#\space #\tab #\newline)))
            (string->number (list->string (reverse string)))
            (if (memq c (string->list "-.0123456789e"))
                            (loop (read-char port) (cons c string))
                            (loop (read-char port) string)))))
(define (read-vector n port)
        (let ((v (make-vector n)))
        (do ((i 0 (+ i 1)))
            ((>= i n) v)
            (vector-set! v i (read-number port)))))
(define (read-regular-state port)
    (let ((t (read-number port))
            (x (read-vector 3 port))
            (v (read-vector 3 port)))
        (->state t x v)))
(define read-manifold-state
    (let ((charts (manifold:get-finite-atlas TSO3))
            (Tchart (make-tangent-chart euler-angles)))
        (lambda (port)
            (let* ((t (read-number port))
                    (chart (read-number port))
                    (coords (read-vector 6 port))
                    (v (chart:point->coords
                            (chart:coords->point coords (list-ref charts chart))
                            Tchart))
                    (psi (vector-ref v 0))
                    (theta (vector-ref v 1))
                    (phi (vector-ref v 2))
                    (psidot (vector-ref v 3))
                    (thetadot (vector-ref v 4))
                    (phidot (vector-ref v 5)))
                    (->state t (vector theta phi psi)
                        (vector thetadot phidot psidot))))))
(define (read-regular-file filename)
```

```
    (let ((port (open-input-file filename)))
        (let loop ((states '()) (count 0) (total 0))
            (if (eof? port)
                        (begin
                            (close-input-port port)
                            (sort states (lambda (x y) (< (state->t x) (state->t y)))))
                    (if (> count 100)
                            (begin
                            (write-line '(read ,total states))
                            (loop (cons (read-regular-state port) states) O (+ total 1)))
                                    (loop (cons (read-regular-state port) states)
                                    (+ count 1) (+ total 1)))))))
(define (read-manifold-file filename)
    (let ((port (open-input-file filename)))
        (let loop ((states '()) (count 0) (total 0))
            (if (eof? port)
                        (begin
                                    (close-input-port port)
                                    (sort states (lambda (x y) (< (state->t x) (state->t y)))))
                            (if (> count 100)
                            (begin
                            (write-line '(read ,total states))
                            (loop (cons (read-manifold-state port) states) 0 (+ total 1)))
                            (loop (cons (read-manifold-state port) states)
                            (+ count 1) (+ total 1))))))
(define (read-pendulum-file filename)
    (let ((port (open-input-file filename)))
        (let loop ((result '()) (count 0) (total 0))
            (if (eof? port)
                        (begin
                        (close-input-port port)
                    (reverse result))
                        (if (> count 100)
                        (begin
                            (write-line '(read ,total states))
                            (loop (cons (read-pendulum-state port) result) 0 (+ total 1)))
                    (loop (cons (read-pendulum-state port) result)
                        (+ count 1) (+ total 1)))))))
(define (read-pendulum-state port)
    (read-number port)
    (let* ((x (read-number port))
            (y (read-number port))
            (z (read-number port))
            (px (read-number port))
            (py (read-number port))
            (pz (read-number port)))
        (imbedding->cotangent S^2 (vector x y z) (vector px py pz))))
```


## C.1.58 rigid-test.scm

```
(load "load-ode")
(define p bad-init)
(define m (tangent:get-anchor p))
```

```
(define p0 (make-tangent c0 m (chart:push-forward p c0)))
(define p1 (make-tangent c1 m (chart:push-forward p c1)))
(define p2 (make-tangent c2 m (chart:push-forward p c2)))
(define p3 (make-tangent c3 m (chart:push-forward p c3)))
(define atlas (manifold:get-finite-atlas so3))
(define cO (list-ref atlas 0))
(define c1 (list-ref atlas 1))
(define c2 (list-ref atlas 2))
(define c3 (list-ref atlas 3))
(define tcO (make-tangent-chart c0))
(define tc1 (make-tangent-chart c1))
(define tc2 (make-tangent-chart c2))
(define tc3 (make-tangent-chart c3))
(define ttcO (make-tangent-chart tc0))
(define ttc1 (make-tangent-chart tc1))
(define ttc2 (make-tangent-chart tc2))
(define ttc3 (make-tangent-chart tc3))
(define (evaluate-rigid-field chart p)
    (let ((x (chart:point->coords p chart)))
        (vector-append x ((make-rigid-body-field chart) x))))
(define vO (evaluate-rigid-field tcO pO))
(define v1 (evaluate-rigid-field tc1 p1))
(define v2 (evaluate-rigid-field tc2 p2))
(define v3 (evaluate-rigid-field tc3 p3))
(for-each
    (lambda (v chart)
        (write-line
            (vector:distance (chart:point->coords (chart:coords->point v chart) ttc0)
                        v0)))
    (list v0 v1 v2 v3)
    (list ttcO ttc1 ttc2 ttc3))
```


## C.1.59 rigid-too.scm

```
(load "load-ode")
```

(load "load-ode")
(define (make-v.field a b c)
(define (make-v.field a b c)
(lambda (state)
(lambda (state)
(vector-append (vector 1)
(vector-append (vector 1)
(rigid-field-1 a b c
(rigid-field-1 a b c
(state->q state)
(state->q state)
(state->qdot state)))))
(state->qdot state)))))
(define (reorder v)
(define (reorder v)
(let ((t (state->t v))
(let ((t (state->t v))
(q (state->q v))
(q (state->q v))
(qdot (state->qdot v)))
(qdot (state->qdot v)))
(let ((psi (vector-ref q 0))
(let ((psi (vector-ref q 0))
(theta (vector-ref q 1))
(theta (vector-ref q 1))
(phi (vector-ref q 2))

```
                (phi (vector-ref q 2))
```

```
    (psidot (vector-ref qdot 0))
    (thetadot (vector-ref qdot 1))
        (phidot (vector-ref qdot 2)))
        (->state t (vector theta phi psi) (vector thetadot phidot psidot)))))
(define v.field (make-v.field 1. (sqrt 2) 2.))
(define step (compose integrator:get-new-x (make-rk4-integrator .01)))
(define results
    (show-time
        (lambda ()
            (let loop ((i 0)
                        (x (->state 0 (vector 0 1 0) (vector . 1 . 1 .1)))
                    (results '()))
            (if (< i 10000)
                (loop (+ i 1) (step x v.field (lambda () 'foo)) (cons x results))
                (map reorder
                    (sort results
                        (lambda (x y) (< (state->t x) (state->t y))))))))))
(define E-errors (map (t-rigid-body 1. (sqrt 2) 2.) results))
(define L-errors (map (state->L-space 1. (sqrt 2) 2.) results))
(define L1-errors (map vector-first L-errors))
(define L2-errors (map vector-second L-errors))
(define L3-errors (map vector-third L-errors))
;;; Graphics devices:
(define dev 'undefined)
(define (open)
    (if (eq? dev 'undefined)
            (begin
                    (set! dev (make-graphics-device 'x))
                    (graphics-operation dev 'set-background-color "white")
                    (graphics-operation dev 'set-foreground-color "blue")
                    (graphics-operation dev 'set-mouse-color "black")
                (graphics-set-coordinate-limits dev -2 -2 2 2)
                (graphics-clear dev))))
(define (close)
    (if (not (eq? dev 'undefined))
            (begin
                    (graphics-close dev)
                    (set! dev 'undefined))))
(define (clear)
    (open)
    (graphics-clear dev))
(define (plot-conservation-error l)
    (let* ((ref (car l))
                    (1 (map (lambda (x) (relative-error x ref)) l))
                    (max (apply max 1))
                    (min (apply min l))
                    (len (- (length 1) 1))
                    (ref (car 1)))
        (write-line '(range: ,min to ,max))
        (open)
```

```
    (graphics-enable-buffering dev)
    (graphics-set-coordinate-limits dev 0. min len max)
    (graphics-move-cursor dev 0. 0.)
    (let loop ((i 1) (l (cdr l)))
        (if (null? 1)
            (graphics-disable-buffering dev)
            (begin
                (graphics-drag-cursor dev i (car l))
                (loop (+ i 1) (cdr 1)))))))
(define (set-fg color)
    (graphics-operation dev 'set-foreground-color color))
```


## C.1.60 rigid-tops.scm

```
;;; This file uses ScmUtils to derive the Lagrangian for tops. This is much
;;; like the rigid body stuff.
(load "rigid")
;;; The Lagrangian for the top (note that the moments of inertia are with
;;; respect to the pivot):
(define (make-top-lagrangian-1 a b c MgR)
    (let ((T (T-rigid-body a b c)))
        (lambda (state)
            (let ((theta (vector-ref (state->q state) 0)))
                        (- (T state) (* MgR (cos theta)))))))
(define (make-top-lagrangian-2 a b c MgR)
    (let ((T (kT-rigid-body a b c)))
        (lambda (state)
            (let* ((q (state->q state))
                (theta (vector-ref q 0))
                    (phi (vector-ref q 1)))
            (+ (T state) (* MgR (cos phi) (sin theta)))))))
```

;; The corresponding Euler-Lagrange equations:
(define (top-sysder-1 a b c MgR)
(lagrangian->state-derivative
(make-top-lagrangian-1 a b c MgR)))
(define (top-sysder-2 a b c MgR)
(lagrangian->state-derivative
(make-top-lagrangian-2 a b c MgR)))
;;; Compute some expressions:
(let ((port (open-output-file "foo")))
(pp (traditional->correct-order
(vector-tail
(show-time
(lambda ()

```
                (*sysder-simplify*
                    ((top-sysder-1 'a 'b 'c 'MgR) rigid-qqdot))))
            1))
        port)
    (close-output-port port))
(let ((port (open-output-file "bar")))
    (pp (traditional->correct-order
                (vector-tail
                    (show-time
                        (lambda ()
                        (*sysder-simplify*
                            ((top-sysder-2 'a 'b 'c 'MgR) rigid-qqdot))))
            1))
        port)
    (close-output-port port))
```


## C.1.61 rigid.scm

```
;;; This file uses ScmUtils to derive the Lagrangian for rigid bodies using an
;; Euler-like coordinate system that covers a region of SO(3) different from
;;; standard Euler angles.
```

;; These are the Euler-like charts we are actually working with:
;;; A canonical rotation matrix that isn't defined in ScmUtils:
(define (rotate-y angle)
(vector
(vector (cos angle) $0 \quad$ (- (sin angle)))
(vector $0 \quad 1 \quad 0$ )
(vector (sin angle) (cos angle))))
;; ; Some procedures for testing charts built on Euler angles:
(define (compare-euler-angles chart euler->m)
(let ( (heta (random pi))
(phi (- (random (* 2 pi)) pi))
(psi (- (random (* 2 pi)) pi)))
(let ((A (chart:coords->point (vector psi theta phi) chart))
(B (euler->m (vector theta phi psi))))
(print-matrix A)
(print-matrix B)
(write-line '(diff = ,(matrix:max (matrix:- A B))))))
(define (test-euler-chart chart)
(let ((psi (- (random (* 2 pi)) pi))
(theta (random pi))
(phi (- (random (* 2 pi)) pi)))
(let ((v (vector psi theta phi)))
(vector:distance (chart:point->coords (chart:coords->point v chart)
chart)
v)) )
(define (test-euler-tangent-chart chart range)

```
    (let ((psi (- (random (* 2 pi)) pi))
        (theta (random pi))
        (phi (- (random (* 2 pi)) pi))
        (psidot (- (random (* 2 range)) range))
        (thetadot (- (random (* 2 range)) range))
        (phidot (- (random (* 2 range)) range))
        (chart (make-tangent-chart chart)))
        (let ((v (vector psi theta phi psidot thetadot phidot)))
            (vector:distance (chart:point->coords (chart:coords->point v chart)
                                    chart)
    v))))
(define (test-euler-tt-chart chart range)
    (let ((psi (- (random (* 2 pi)) pi))
                (theta (random pi))
        (phi (- (random (* 2 pi)) pi))
        (psidot (- (random (* 2 range)) range))
        (thetadot (- (random (* 2 range)) range))
        (phidot (- (random (* 2 range)) range))
        (a (- (random (* 2 range)) range))
        (b (- (random (* 2 range)) range))
        (c (- (random (* 2 range)) range))
        (d (- (random (* 2 range)) range))
        (e (- (random (* 2 range)) range))
        (f (- (random (* 2 range)) range))
            (chart (make-tangent-chart (make-tangent-chart chart))))
        (let ((v (vector psi theta phi psidot thetadot phidot
                        a b c def)))
            (vector:distance (chart:point->coords (chart:coords->point v chart)
                                    chart)
                    v))))
```

;; Generate rotation matrix from angles: (Why is the order of arguments used
;;; in the book so *odd*?)
(define (euler0->m angles)
(let ((theta (vector-ref angles 0))
(phi (vector-ref angles 1))
(psi (vector-ref angles 2)))
(matrix:* (rotate-z phi)
(rotate-y (- theta))
(rotate-z psi))))
(define euler1->m
(let ( (rot (vector (vector -1 00 )
(vector $0-10$ )
(vector 00 1))))
(lambda (angles)
(let ((theta (vector-ref angles 0))
(phi (vector-ref angles 1))
(psi (vector-ref angles 2)))
(matrix:* (rotate-z phi)
(rotate-y (- theta))
(rotate-z psi)
rot))))
(define euler2->m

```
    (let ((rot-x (vector (vector 1 0 0)
    (vector 0 0 1)
    (vector 0-1 0)))
        (rot-y (vector (vector -1 0 0)
                            (vector 0}100
                            (vector 0 0 -1))))
    (lambda (angles)
        (let ((theta (vector-ref angles 0))
            (phi (vector-ref angles 1))
            (psi (vector-ref angles 2)))
            (matrix:* (rotate-y phi)
                                    rot-y
                            (rotate-z (- theta))
                            rot-x
                            (rotate-z psi))))))
(define euler3->m
    (let ((rot-x (vector (vector 1 0 0)
                        (vector 0 0 1)
                            (vector 0 -1 0)))
        (rot-y (vector (vector -1 0 0)
                            (vector 0 1 0)
                            (vector 0}00-1))
            (rot-z (vector (vector -1 0 0)
                            (vector 0-1 0)
                            (vector 0 0 1))))
        (lambda (angles)
            (let ((theta (vector-ref angles 0))
                        (phi (vector-ref angles 1))
                (psi (vector-ref angles 2)))
            (matrix:* (rotate-y phi)
                rot-y
                        (rotate-z (- theta))
                        rot-x
                        (rotate-z psi)
                rot-z)))))
```

;;; Generate the angular velocity vector:
(define (((make-euler->omega euler->m) angles-path) t)
(define ( $m$-on-path t )
(euler->m (angles-path t)))
(define ( $w$-cross $t$ )
(matrix:* ((derivative m-on-path) t)
(matrix:transpose (m-on-path t))))
(antisymmetric->3vector-components (w-cross t)))
;; Generate the angular velocity vector in the body frame:
(define (((make-euler->omega-body euler->m) angles-path) t)
(matrix:matrix*vector
(matrix:transpose (euler->m (angles-path t)))
(((make-euler->omega euler->m) angles-path) $t)$ ))
;; Compute the expression for the angular velocity in terms of the angles:

```
(define (angular-velocity-expression euler->m)
    (let ((euler->omega-body (make-euler->omega-body euler->m)))
        (show-time
            (lambda ()
                ((compose ham:simplify easy-simplify)
                ((euler->omega-body
                    (vector (literal-function 'theta)
                                (literal-function 'phi)
                                (literal-function 'psi)))
                    't()))|)
;;; Compute the kinetic energy!
(define ((t-rigid-body-0 a b c) state)
    (let ((q (state->q state))
                (qdot (state->qdot state))
                (t (state->t state)))
        (let ((theta (vector-ref q 0))
                    (phi (vector-ref q 1))
                    (psi (vector-ref q 2))
                    (thetadot (vector-ref qdot 0))
                    (phidot (vector-ref qdot 1))
                    (psidot (vector-ref qdot 2)))
                (let ((w_a (- (* thetadot (sin psi)) (* (sin theta) (cos psi) phidot)))
                    (w_b (+ (* (cos psi) thetadot) (* (sin theta) (sin psi) phidot)))
                            (w_c (+ psidot (* (cos theta) phidot))))
                (* 1/2 (+ (* a (square w_a))
                            (* b (square w_b))
                            (* c (square w_c))))))))
(define ((t-rigid-body-1 a b c) state)
    (let ((q (state->q state))
                (qdot (state->qdot state))
                (t (state->t state)))
        (let ((theta (vector-ref q 0))
                    (phi (vector-ref q 1))
                    (psi (vector-ref q 2))
                    (thetadot (vector-ref qdot 0))
                    (phidot (vector-ref qdot 1))
                    (psidot (vector-ref qdot 2)))
            (let ((w_a (- (* (sin theta) (cos psi) phidot) (* thetadot (sin psi))))
                    (w_b (- (+ (* (sin theta) (sin psi) phidot)
                                    (* (cos psi) thetadot))))
                                    (w_c (+ psidot (* (cos theta) phidot))))
                (* 1/2 (+ (* a (square w_a))
                            (* b (square w_b))
                            (* c (square w_c))))))))
(define ((t-rigid-body-2 a b c) state)
    (let ((q (state->q state))
                (qdot (state->qdot state))
                (t (state->t state)))
            (let ((theta (vector-ref q 0))
                    (phi (vector-ref q 1))
                    (psi (vector-ref q 2))
                    (thetadot (vector-ref qdot 0))
                    (phidot (vector-ref qdot 1))
                    (psidot (vector-ref qdot 2)))
```

```
        (let ((w_a (+ (* (sin theta) (cos psi) phidot) (* (sin psi) thetadot)))
                    (w_b (- (* (cos psi) thetadot) (* (sin theta) (sin psi) phidot)))
                (w_c (- psidot (* (cos theta) phidot))))
    (* 1/2 (+ (* a (square w_a))
                    (* b (square w_b))
                            (* c (square w_c))))))))
(define ((t-rigid-body-3 a b c) state)
    (let ((q (state->q state))
            (qdot (state->qdot state))
            (t (state->t state)))
        (let ((theta (vector-ref q 0))
                    (phi (vector-ref q 1))
                    (psi (vector-ref q 2))
                    (thetadot (vector-ref qdot 0))
                    (phidot (vector-ref qdot 1))
                    (psidot (vector-ref qdot 2)))
            (let ((w_a (- (+ (* (sin theta) (cos psi) phidot)
                    (* (sin psi) thetadot))))
                        (w_b (- (* (sin theta) (sin psi) phidot) (* (cos psi) thetadot)))
                (w_c (- psidot (* (cos theta) phidot))))
            (* 1/2 (+ (* a (square w_a))
                            (* b (square w_b))
                            (* c (square w_c))))))))
;;; Compute the angular momentum!
(define ((state->L-body-0 a b c) state)
    (let ((q (state->q state))
            (qdot (state->qdot state))
            (t (state->t state)))
        (let ((theta (vector-ref q 0))
                    (phi (vector-ref q 1))
                    (psi (vector-ref q 2))
                (thetadot (vector-ref qdot 0))
                    (phidot (vector-ref qdot 1))
                    (psidot (vector-ref qdot 2)))
            (let ((w_a (- (* thetadot (sin psi)) (* (sin theta) (cos psi) phidot)))
                    (w_b (+ (* (cos psi) thetadot) (* (sin theta) (sin psi) phidot)))
                    (w_c (+ psidot (* (cos theta) phidot))))
            (vector (* a w_a) (* b w_b) (* c w_c))))))
(define ((state->L-body-1 a b c) state)
    (let ((q (state->q state))
            (qdot (state->qdot state))
            (t (state->t state)))
        (let ((theta (vector-ref q 0))
                (phi (vector-ref q 1))
                    (psi (vector-ref q 2))
                    (thetadot (vector-ref qdot 0))
                    (phidot (vector-ref qdot 1))
                    (psidot (vector-ref qdot 2)))
            (let ((w_a (- (* (sin theta) (cos psi) phidot) (* thetadot (sin psi))))
                    (w_b (- (+ (* (sin theta) (sin psi) phidot)
                            (* (cos psi) thetadot))))
                            (w_c (+ psidot (* (cos theta) phidot))))
            (vector (* a w_a) (* b w_b) (* c w_c))))))
```

```
(define ((state->L-body-2 a b c) state)
    (let ((q (state->q state))
                (qdot (state->qdot state))
                    (t (state->t state)))
        (let ((theta (vector-ref q 0))
                (phi (vector-ref q 1))
                (psi (vector-ref q 2))
                (thetadot (vector-ref qdot 0))
                (phidot (vector-ref qdot 1))
                (psidot (vector-ref qdot 2)))
            (let ((w_a (+ (* (sin theta) (cos psi) phidot) (* (sin psi) thetadot)))
                (w_b (- (* (cos psi) thetadot) (* (sin theta) (sin psi) phidot)))
                (w_c (- psidot (* (cos theta) phidot))))
                (vector (* a w_a) (* b w_b) (* c w_c))))))
(define ((state->L-body-3 a b c) state)
    (let ((q (state->q state))
                (qdot (state->qdot state))
                    (t (state->t state)))
        (let ((theta (vector-ref q 0))
                (phi (vector-ref q 1))
                (psi (vector-ref q 2))
                (thetadot (vector-ref qdot 0))
                (phidot (vector-ref qdot 1))
                (psidot (vector-ref qdot 2)))
            (let ((w_a (- (+ (* (sin theta) (cos psi) phidot)
                            (* (sin psi) thetadot))))
                (w_b (- (* (sin theta) (sin psi) phidot) (* (cos psi) thetadot)))
                (w_c (- psidot (* (cos theta) phidot))))
            (vector (* a w_a) (* b w_b) (* c w_c))))))
(define ((state->L-space-0 a b c) state)
    (let ((angles (state->q state)))
        (* (euler0->m angles) ((state->L-body-0 a b c) state))))
(define ((state->L-space-1 a b c) state)
    (let ((angles (state->q state)))
        (* (euler1->m angles) ((state->L-body-1 a b c) state))))
(define ((state->L-space-2 a b c) state)
    (let ((angles (state->q state)))
        (* (euler2->m angles) ((state->L-body-2 a b c) state))))
(define ((state->L-space-3 a b c) state)
    (let ((angles (state->q state)))
            (* (euler3->m angles) ((state->L-body-3 a b c) state))))
;;; The state derivatives:
(define (rigid-sysder-0 a b c)
    (lagrangian->state-derivative
        (t-rigid-body-0 a b c)))
(define (rigid-sysder-1 a b c)
    (lagrangian->state-derivative
        (t-rigid-body-1 a b c)))
(define (rigid-sysder-2 a b c)
```

```
    (lagrangian->state-derivative
        (t-rigid-body-2 a b c)))
(define (rigid-sysder-3 a b c)
    (lagrangian->state-derivative
        (t-rigid-body-3 a b c)))
;;; This is rather useful:
(define rigid-qqdot
    (->state 't
                            (vector 'theta 'phi 'psi)
                            (vector 'thetadot 'phidot 'psidot)))
(define rigid-qP
    (->state 't
            (vector 'theta 'phi 'psi)
            (vector 'p_theta 'p_phi 'p_psi)))
;;; Something useful to remember (thanks to CPH): This should fix the
;;; simplifier so that the really big expressions can be simplified.
;(ge '(user))
;(in-package scmutils-base-environment
; ((pcf-package 'set-gcd-method!) (pcf-package 'gcd-euclid)))
;(ge generic-environment)
;;; Stupid exchange of order of arguments:
(define (traditional->correct-order v)
    (let ((theta (vector-ref v 0))
            (phi (vector-ref v 1))
            (psi (vector-ref v 2))
            (thetadot (vector-ref v 3))
            (phidot (vector-ref v 4))
            (psidot (vector-ref v 5)))
        (vector psi theta phi psidot thetadot phidot)))
```


## C.1.62 smooth.scm

; ; Here is an example to worry about: Let $X$ be a manifold, and let $H$ be a
; ; ; real-valued smooth function on its cotangent bundle $T * X$. Let $Y$ be a
; ; ; submanifold of $X$, and let $i: Y->X$ be the inclusion map. Then we can define
; ; ; $\mathrm{Ti}(\mathrm{x}, \mathrm{v})=\mathrm{di}(\mathrm{x})(\mathrm{v})$, as usual, and consider H ' $=\mathrm{Ti} \mathrm{H}_{\mathrm{H}}$, the restriction of H
;; to the submanifold Y. (Holonomic constraints!) Now consider this: dH' =
; ; $\mathrm{d}(\mathrm{Ti} * \mathrm{H})=\mathrm{Ti} * \mathrm{dH}$, which basically means ( $\mathrm{T}(\mathrm{Ti})) *(\mathrm{dH})$, right? And anyway TH'
$; ; ;=T(T i * H)=T(T i) *(T H)$, by covariant functoriality. Now, the latter
; ; ; expression is computable under the current system, but does not provide
; ; ; closure, while the first expression is not even computable, but would
; ; ; provide closure if it were. Obviously, this system needs to be
; ; ; restructured: We should at least be able to express holonomic constraints!
;; The answer to this is that SMOOTH-MAP:DIFF needs to push the tangent

```
;;; functor *into* compositions, not pull them out. So maps are differentiated
;;; in stages rather than as a whole. This sounds like it stands a chance,
;;; actually.
;;; This file defines the structures for smooth maps between manifolds. A
;;; smooth map, in addition to a Scheme procedure that computes the point
;;; transformation, should also contain pointers to its domain, range, and
;;; methods for making transition maps. But then that makes constructing
;;; charts a bit more recursive than necessary, so domains and ranges are not
;;; included (it's not useful information).
(declare (usual-integrations))
;;; How to make one:
(define (make-smooth-map domain range point-function make-transition)
    ;; POINT-FUNCTION should be a scheme procedure that computes the function
    ;; given a point in its domain. Note that this function is *not* meant to be
    ;; used in manipulations of the function, but only in computing the values.
    ;; In order that the functorial properties are satisfied (whatever that
    ;; means), compositions and exterior differentiation need to be handled
    ;; separately so that the resulting functions are always differentiable. (In
    ;; particular, this makes ScmUtils work.)
    ;; MAKE-TRANSITION should take as arguments two charts and create a
    ;; transition function between the respective Euclidean spaces.
    (vector domain range point-function make-transition '()))
(define (make-simple-map domain range f)
    (make-smooth-map domain range f (make-simple-transition-maker f)))
(define (make-simple-transition-maker f)
    (lambda (U V)
        (compose (chart:get-coord-map V) f (chart:get-inverse-map U))))
;;; Will probably be useful in our PDE work:
(define R^1 (make-euclidean-space 1))
(define real-line R^1)
(define (make-real-map domain f)
    (make-simple-map domain real-line (compose vector f)))
;;; Accessors:
(define (smooth-map:get-domain f)
    (vector-ref f 0))
(define (smooth-map:get-range f)
    (vector-ref f 1))
(define (smooth-map:get-point-function f)
    (vector-ref f 2))
```

```
(define (smooth-map:get-transition-maker f)
    (vector-ref f 3))
(define (smooth-map:make-transition f U V)
    ((smooth-map:get-transition-maker f) U V))
(define (smooth-map:get-extra f tag)
    (let ((result (assq tag (vector-ref f 4))))
        (if result
                        (cdr result)
                        #f)))
(define (smooth-map:install-extra f tag datum)
    (let ((result (assq tag (vector-ref f 4))))
        (if result
                        (set-cdr! result datum)
                        (vector-set! f 4 (cons (cons tag datum) (vector-ref f 4))))))
```

;;; Useful constructs:
(define (apply-smooth-map f p)
((smooth-map:get-point-function f) p))
(define (smooth-map:compose f . rest)
(if (null? rest)
$f$
(let* ((last-guy (car (reverse rest)))
(flist (cons frest))
(point-map (apply compose (map smooth-map:get-point-function
flist)))
(h (make-smooth-map (smooth-map:get-domain last-guy)
(smooth-map:get-range f)
point-map
(make-simple-transition-maker point-map))))
(smooth-map:install-extra h 'composition flist)
h)))
(define (smooth-map:decompose f)
(let ((result (smooth-map:get-extra $f$ 'composition)))
(if result
result
\#f))
;;; It would be nice to make everything else (such as composition and
;; differentiation) do the right thing with regards to inverses:
(define (make-diffeomorphism fintinverse)
(smooth-map:install-extra $f$ 'inverse f-inverse)
(smooth-map:install-extra f-inverse 'inverse f)
f)
(define (smooth-map:invert f)
(let ((result (smooth-map:get-extra f 'inverse)))
(if result
result
\#f))

```
(define (make-simple-diffeomorphism domain range f g)
    (let ((sf (make-simple-map domain range f))
            (sg (make-simple-map range domain g)))
        (make-diffeomorphism sf sg)))
;;; Some useful (covariant) functors:
;;; This one maps from the category of smooth manifolds into point sets, and
;;; maps smooth maps.
(define forgetful-functor smooth-map:get-point-function)
;;; This uses the differential to map from smooth manifolds into the category
;;; of tangent bundles.
(define (smooth-map:diff f)
    (let ((Tf (smooth-map:get-extra f 'tangent-extension)))
        (if Tf
            Tf
            (let ((flist (smooth-map:decompose f)))
                (if flist
                            (apply smooth-map:compose (map smooth-map:diff flist))
                            (let ((components (product-map:get-components f)))
                        (if components
                            (make-product-map (smooth-map:diff (car component))
                                    (smooth-map:diff (cadr component)))
                                    (smooth-map:new-diff f))))))))
(define (smooth-map:new-diff smap)
    (let* ((TM (make-tangent-bundle (smooth-map:get-domain smap)))
                        (TN (make-tangent-bundle (smooth-map:get-range smap)))
                (transit (diff-transition-map smap))
                (df (diff-point-function smap)))
        (let ((new-map (make-smooth-map TM TN df transit)))
            (smooth-map:install-extra smap 'tangent-extension new-map)
            new-map)))
(define (diff-point-function f)
    (let ((N (smooth-map:get-range f)))
        (lambda (v)
            (let* ((p (tangent:get-anchor v))
                    (q (apply-smooth-map f p))
                            (M-chart (tangent:get-chart v))
                            (N-chart (manifold:find-best-chart N q))
                            (transit (smooth-map:make-transition f M-chart N-chart)))
                (make-tangent N-chart
                        q
                        (push-forward-in-coords transit
                                    (chart:point->coords p M-chart)
                                    (tangent:get-coords v)))))))
(define (diff-transition-map smap)
    ;; Make a transition map between the tangent charts of two given charts.
```

```
    ;; Note that this depends on the fact that SMOOTH-MAP:DIFF decomposes
    ;; compositions of functions into chunks whose transition maps are directly
    ;; differentiable.
    (let ((make-transition-map (smooth-map:get-transition-maker smap)))
        (lambda (TU TV)
        (let* ((U (chart:get-base-chart TU))
            (V (chart:get-base-chart TV))
                    (f (make-transition-map U V))
                    (dim (chart:dimension U)))
                (lambda (p)
            (let ((x (vector-head p dim)))
                        (vector-append (f x)
                            (push-forward-in-coords
                        f x (vector-end p dim))))))))
;;; Another very useul construction:
(define product:combine cons)
(define product:get-arg-1 car)
(define product:get-arg-2 car)
(define (make-product-map fg)
    (let ((fp (smooth-map:get-point-function f))
            (gp (smooth-map:get-point-function g))
            (M-1 (smooth-map:get-domain f))
            (M-2 (smooth-map:get-domain g))
            (N-1 (smooth-map:get-range f))
            (N-2 (smooth-map:get-range g)))
        (let ((point-map
                    (lambda (x)
                        (product:combine (fp (product:get-arg-1 x))
                            (gp (product:get-arg-2 x)))))
                (make-transition-map
                    (lambda (U V)
                        (let ((dim-1 (chart:dimension U))
                            (dim-2 (chart:dimension V)))
                        (lambda (x)
                            (vector-append (fp (vector-head x dim-1))
                                    (gp (vector-end x dim-2))))))))
            (let ((f&g (make-smooth-map (product-manifold M1 M2)
                                    (product-manifold N1 N2)
                                    point-map
                                    make-transition-map)))
                (smooth-map:install-extra f&g 'product-map-structs (list f g))
                f&g))))
(define (product-map:get-structs f)
    (smooth-map:get-extra f 'product-map-structs))
(define (product-map:get-components f)
    (let ((result (product-map:get-structs f)))
```

```
    (if result
        result
        #f)))
```

```
;;; Some useful examples:
(define (make-simple-projection-map n i)
    ;; Make a map from R^n to R^(n-1) by dropping the ith coordinate.
    (make-simple-map (make-euclidean-space n)
                            (make-euclidean-space (- n 1))
                            (lambda (v) (vector:drop-coord v i))))
(define (make-simple-imbedding-map n i)
    ;; Do the opposite:
    (make-simple-map (make-euclidean-space (- n 1))
                                    (make-euclidean-space n)
                            (lambda (v) (vector:add-coord v i))))
```


## C.1.63 spaces.scm

```
;;; Some manifolds:
```

; (declare (usual-integrations cos sin acos atan + * / ))
;; ; The n-sphere? What sort of chart should we use? Stereogrphic projection?
;;; Mercator projection? Both? The representation of points isn't so trivial
;; ; in this case. I guess we'll just use the imbedding, since in this context
;; it's perfectly natural.
(define (make-imbedded-sphere-test dim)
(let ((euclidean? (make-euclidean-test (+ dim 1))))
(lambda (v)
(and (euclidean? v)
(almost-equal? (vector:magnitude ${ }^{\wedge} 2$ v) 1)))))
;;; Do the obvious thing: Stereographic projection.
(define (make-stereographic-chart dim pole-dim pole-dir)
(let* ( ${ }^{\text {ubound 5) }}$
(dim+1 (+ dim 1))
(pole (vector:basis dim+1 pole-dim pole-dir)))
(letrec
( (in-domain?
(let ((sphere? (make-imbedded-sphere-test dim)))
(lambda (v)
(and (sphere? v)
(not (almost-equal? (vector:distance^2 $v$ pole) 0 ))
(< (- (/ 4 (vector:magnitude^2 (vector:- v pole))) 1)
ubound))))

```
        (in-range?
        (let ((euclidean? (make-euclidean-test dim)))
            (lambda (v)
                (and (euclidean? v)
                    (< (vector:magnitude^2 v) ubound)))))
    (map
        (lambda (x)
        (let* ((d (vector:- x pole))
                    (y (vector:* (/ 2 (vector:magnitude-2 d)) d)))
                (vector:drop-coord (vector:+ y pole) pole-dim))))
        (inverse
        (lambda (x)
        (let* ((d (vector:- (vector:add-coord x pole-dim) pole))
                    (y (vector:* (/ 2 (vector:magnitude^2 d)) d)))
                (vector:+ y pole)))))
(let ((chart (make-simple-chart dim in-domain? in-range? map inverse)))
    (make-spherical-range chart (make-vector dim 0) (sqrt ubound))
    chart))))
;:; Of course, in most applications, it's better to have (generalized)
;;; spherical coordinates...
;;; ILIST should be a permutation of 0, 1, ..., dim. It determines the order
;;; in which the angles are generated. The singularity is a half-sphere of
;;; dimension (dim - 1), and is orthogonal to the last coordinate in ILIST,
;;; occupying the negative half space with respect to the next-to-last
;;; coordinate. ROT should be an angle in radians; the final two coordinates
;;; are rotated by this angle before being generated.
(define (make-planar-rotation angle)
    (let* ((cosine (cos angle))
                    (sine (sin angle))
            (A (list->matrix 2 2'(,cosine ,sine ,(- sine) ,cosine))))
        (lambda (v)
            (apply-linear-transformation A v))))
(define (make-spherical-chart dim ilist angle)
    (let ((e1 0)
        (e2 (/ pi 9))
        (dim-1 (- dim 1))
        (dim+1 (+ dim 1))
        (rot (make-planar-rotation angle))
        (-rot (make-planar-rotation (- angle))))
        ;; The membership test is rather circular... (No pun intended! :)
        (letrec
            ((in-domain?
                (let ((sphere? (make-imbedded-sphere-test dim)))
                        (lambda (v)
                        (and (sphere? v)
                        (in-range? (map v))))))
                (in-range?
```

```
            (let ((euclidean? (make-euclidean-test dim)))
                        (lambda (v)
                (and (euclidean? v)
                        (let valid? ((i 0))
                                (let ((angle (vector-ref v i)))
                                    (if (< i dim-1)
                                    (and (< e1 angle)
                                    (< angle (- pi e1))
                                    (valid? (+ i 1)))
                                    (and (< (+ -pi e2) angle)
                                    (< angle (- pi e2)))))))))
```

(map
(lambda (x)
(let ((result (make-vector dim 0$)$ ))
(let loop ( i 0 ) (ilist ilist) ( $r$ 1))
(if (= i dim-1)
(let ( $z$ (rot (vector (vector-ref $x$ (car ilist))
(vector-ref $x(c a d r$ ilist))))))
(vector-set! result i
(atan (vector-ref z 1) (vector-ref z 0)))
result)
(let ((val (/ (vector-ref $x$ (car ilist)) r)))
(vector-set! result $i$ (acos val))
(loop (+ i 1)
(cdr ilist)
(* r (sqrt (- 1 (square val))))))))))
(inverse
(lambda (x)
(let ( p (make-vector dim+1)))
(let loop ( (i 0) (ilist ilist) ( r 1))
(let ((angle (vector-ref $x$ i)))
(if (< i dim-1)
(begin
(vector-set! p (car ilist) (* r (cos angle)))
(loop (+ i 1) (cdr ilist) (* r (sin angle))))
(let ((z (-rot (vector (cos angle) (sin angle)))))
(vector-set! p (car ilist) (* r (vector-ref z 0)))
(vector-set! $p$ (cadr ilist) (* $r$ (vector-ref $z 1$ )))
p) ) ) ) ) ) )
(let ((chart (make-simple-chart dim in-domain? in-range? map inverse))
(intervals (make-vector dim (make-interval e1 (- pi e1))))
(vector-set! intervals dim-1 (make-interval (+ -pi e2) (- pi e2)))
(make-cell-range chart intervals)
chart))))
; ; ; Here's one way to make a sphere; it turns out to be very hard to define
;;; vector fields on its tangent bundle. (Try the pendulum!)
(define (make-stereographic-sphere dim)
(charts->manifold (list (make-stereographic-chart dim 0 1.)
(make-stereographic-chart $\operatorname{dim} 0-1$.$) )))$
;;; Another way:
(define (make-spherical-sphere dim)

```
(let* ((l1 (list-integers dim))
        (12 (reverse 11)))
    (charts->manifold (list (make-spherical-chart dim 11 0)
                                    (make-spherical-chart dim 12 pi)))))
```

```
;;; Choose a way to make spheres:
```

;;; Choose a way to make spheres:
(define make-sphere make-spherical-sphere)
;;; The next thing to make is SO(3). Note that because the inverse function
;;; theorem can't be used to compute coordinate systems directly, it
;;; complicates the creation of charts for Lie subgroups of GL(n).
;;; For now, we won't bother with explicitly representing Lie group structures
;;; computationally.
(define (make-special-orthogonal-group n)
;; Don't bother making SO(n) in general:
(case n
;; n = 2 is just the circle group:
((2) (make-sphere 1))
;; n = 3 is the rotational group in 3-space:
((3) (make-rotational-group))
;; Otherwise panic:
(else
(error "Sorry! I only know how to make SO(2) and SO(3)! -- MAKE-SO(n)"))))
;;; Make a planar rotation matrix in n-space, in a plane specified by two
;;; canonical coordinate axes:
(define (make-rotation-matrix dim x-axis y-axis theta)
(let ((rot (make-matrix dim dim)))
(do ((i 0 (+ i 1)))
((>= i dim) rot)
(do ((j 0 (+ j 1)))
((>= j dim))
(cond ((and (= i x-axis) (= j x-axis))
(matrix-set! rot i j (cos theta)))
((and (= i x-axis) (= j y-axis))
(matrix-set! rot i j (- (sin theta))))
((and (= i y-axis) (= j y-axis))
(matrix-set! rot i j (cos theta)))
((and (= i y-axis) (= j x-axis))
(matrix-set! rot i j (sin theta)))
((= i j)
(matrix-set! rot i j 1)))))))

```
```

;;; Make the rotational group using Euler angles. This is a nice test because
;;; they contain singularities:
;;; Need support for Lagrangian and Hamiltonian dynamics, too, if you believe
;;; in such things.
(define make-rotational-group
(let ((result \#f))
(lambda ()
(if result
result
(let ((SO3 (charts->manifold
(list (make-euler-angles 0 1 0 0)
(make-euler-angles 0 1 0 pi)
(make-euler-angles 0 2 pi 0)
(make-euler-angles 0 2 pi pi)))))
;; This ensures that we get an atlas. Might be a bit of an
;; overkill, but...
(set! result SO3)
result)))))
;;; SO(3) = S^2 x S^1? But it's probably not easier to do, and this (mostly)
;;; works...
(define (make-euler-angles i-axis j-axis r1 r2)
;; General strategy: Given a specification of two axes (I and J), we can
;; deduce K. Using I, J, and K, we construct a spherical chart. We can then
;; decompose the rotation R into an S^2 and an S^1 component by its action on
;; some fixed vector v: Rv gives the S^2 component, and its action about the
;; axis specified by v gives the S^1 component. We choose v to be the
;; standard z-axis.
(let* ((id (make-identity-matrix 3))
(3-vector? (make-euclidean-test 3))
(k-axis (- 3 (+ i-axis j-axis)))
(S-chart (make-spherical-chart 2 (list k-axis i-axis j-axis) r1))
(C-chart (make-spherical-chart 1 (list 0 1) r2))
(xv (vector:basis 3 O 1))
(zv (vector:basis 3 2 1))
(rot (generate-axis-rotation k-axis 2))
(inv-rot (transpose rot)))
(letrec
((in-domain?
(lambda (R)
;; First, check that it's a matrix:
(and (matrix? R)
;; Next, check that it's orthogonal:
(let ((diff (matrix:- (matrix:* R (transpose R)) id)))
(almost-equal? (matrix:max diff) 0))
;; And then check that it works with the S-chart:

```
```

                    ;; Finally, check that it really checks out completely:
    ```
                    (in-range? (coord-map R))))
                (in-range?
(lambda
                (lambda ( \(x\) )
            (and (3-vector? \(x\) )
                    (chart:in-range? (vector-head \(\times 1\) ) C-chart)
                    (chart:in-range? (vector-tail \(\times 1\) ) S-chart))))
                (coord-map
                        (let ((i (vector:basis 3 i-axis 1))
                    (j (vector:basis 3 j-axis 1))
                    (k (vector:basis 3 k -axis 1)))
        (lambda (R)
            (let* ( \((\mathrm{v}\) (apply-linear-transformation R zv))
                        (coords (chart:point->coords v S-chart))
                        (S (generate-rotation coords r1 i-axis j-axis k-axis))
                            (T (matrix:* rot (transpose S) R))
                            (w (apply-linear-transformation \(T \times V\) ))
                            (psi (chart:point->coords (vector-head w 2) C-chart)))
                    (vector-append psi coords)))))
        (inverse-map
        (lambda (x)
        (let ( psi (circle->rotation
                            (chart:coords->point (vector-head \(x\) 1) C-chart)))
                    (S (generate-rotation (vector-tail \(\times 1\) ) \(r 1\)
                                    i-axis j-axis \(k\)-axis)))
            (matrix:* S inv-rot psi))))
        (make-simple-chart 3 in-domain? in-range? coord-map inverse-map))))
(define (circle->rotation \(v\) )
    (let (mat (make-matrix 3 3)))
        (matrix-set! mat 22 1)
        (let ((cos (vector-ref \(\vee 0)\) )
            (sin (vector-ref vi)))
            (matrix-set! mat 00 cos)
            (matrix-set! mat 11 cos)
            (matrix-set! mat 01 (- sin))
            (matrix-set! mat \(10 \sin )\) )
        mat))
(define (generate-rotation coords angle \(x\)-axis \(y\)-axis \(z\)-axis)
    ; ; Generate a rotation matrix that takes the z-axis to the coordinates
    ; ; specified, minus an extra rotation of ANGLE about the z-axis.
    (matrix:*
        (make-rotation-matrix \(3 x\)-axis \(y\)-axis (+ (vector-ref coords 1) angle))
        (make-rotation-matrix 3 z-axis \(x\)-axis (vector-ref coords 0 ))))
(define (generate-axis-rotation axis-1 axis-2)
    ;; Generate an arbitrary (but deterministic) rotation that takes AXIS-1 to
    ; ; AXIS-2:
```

    (if (= axis-1 axis-2)
    (make-identity-matrix 3)
    (let ((i (vector:basis 3 axis-1 1))
            (j (vector:basis 3 axis-2 1))
            (R (make-rotation-matrix 3 axis-1 axis-2 (/ pi 2))))
        (if (almost-zero? (vector:distance
                            (apply-linear-transformation R i) j))
            R
            (transpose R)))))
    ;;; We need the unit closed n-ball for testing the PDE solver.
(define (make-ball n . argl)
;; It might be easier to do this in terms of the n-sphere. The unit interval
;; (n = 1) case would have to be treated separately, then, since in that case
;; the boundary is a 0-dimensional manifold.
(let ((make-sphere make-sphere))
;; Really, no matter how the sphere is made, so long as it has a finite
;; atlas, this will work. But the user might have a preference, e.g. when
;; generating the mesh for solving PDEs.
(if (and (not (null? argl))
(procedure? (car argl)))
(set! make-sphere (car argl)))
(cond ((= n 1)
(error "Sorry! The unit interval hasn't been implemented yet!"))
((>= n 2)
(let ((boundary (make-spherical-sphere (- n 1))))
(let* ((n-vector? (make-euclidean-test n))
(in? (lambda (p)
(and (n-vector? p)
(< (vector:magnitude p) 2/3))))
(center-chart (make-simple-chart
n in? in? identity identity)))
;; Define a chart that covers a neighborhood of the center, then
;; define the rest as a deformation retract onto the boundary.
(make-spherical-range center-chart (make-vector n 0) 2/3)
(let loop ((B-charts (list center-chart))
;; The sphere should have a finite atlas.
(S-charts (manifold:get-finite-atlas boundary)))
(if (null? S-charts)
(charts->manifold B-charts)
(let ((S-chart (car S-charts)))
(let ((S-coord-map (chart:get-coord-map S-chart))
(S-inverse (chart:get-inverse-map S-chart))

```
```

    (in-S-domain? (chart:get-membership-test S-chart))
    (in-S-range? (chart:get-range-test S-chart)))
    ;; Construct a chart on the ball out of a chart on the
;; sphere:
(let ((coord-map
(lambda (p)
(let* ((len (vector:magnitude p))
(x (vector:add-coord
(S-coord-map (vector:* (/ len) p))
0)))
(vector-set! x 0 len)
x)))
(inverse-map
(lambda (x)
(vector:* (vector-ref x 0)
(S-inverse (vector-tail x 1)))))
(in-domain?
(lambda (p)
(and (n-vector? p)
(let ((len (vector:magnitude p)))
(and (< 1/3 len)
(<= len 1)
(in-S-domain?
(vector:* (/ len) p)))))))
(in-range?
(lambda (x)
(and (n-vector? x)
(let ((len (vector-ref x 0)))
(and (< 1/3 len) (<= len 1)))
(in-S-range? (vector-tail x 1))))))
(let ((B-chart (make-simple-chart
n in-domain? in-range?
coord-map inverse-map)))
;; This thing has a boundary:
(add-boundary-to-chart B-chart 0 1)
;; Check the range type; this is useful for
; ; meshing. Note that we only handle n-cells
;; for now.
(if (chart:cell-range? S-chart)
(let ((int (cell-range:get-interval-list
S-chart)))
(make-cell-range
B-chart (cons (make-interval 1/3 1) int))))
;; Keep going:
(loop (cons B-chart B-charts)
((cdr S-charts)))))))))))

```
```

(else (error "Error: Invalid argument. -- MAKE-BALL")))))

```

\section*{C.1.64 stubs.scm}
```

;;; Using Scmutils' differentiation facilities to replace the numerical
;;; differentiation stuff.
(declare (usual-integrations))
;;; Ugly kludge, but sort of works. For everything we care about, at any rate.
(define (diff f)
(let ((df (derivative f)))
(lambda (p)
(let ((J (df p)))
(if (number? J)
(set! J (vector (vector J)))
(if (not (vector? (vector-ref J 0)))
(set! J (vector->row-matrix J))))
(lambda (v)
(if (number? v)
(set! v (vector v)))
(apply-linear-transformation J v))))))
;;; Scmutils can't handle this, though:
;(define (f x)
;(if (> x 0)
;(exp (- (/ (square x))))
;0))
;(((derivative f) x) 1)
;;; Need to replace some linear algebra stuff, too:
(define (list->matrix m n l)
(let ((v (list->vector l)))
(generate-matrix
m n
(lambda (i j)
(vector-ref v (+ (* i n) j))))))
(define vector:* vector:scalar*vector)
(define vector:dot vector:dot-product)
(define inner-product vector:dot)
(define apply-linear-transformation matrix:matrix*vector)
;;; Printing matrices:
(define matrix-row-count matrix:num-rows)
(define matrix-column-count matrix:num-cols)
(define (matrix-size M)

```
```

    (list (matrix-row-count M) (matrix-column-count M)))
    (define (print-matrix matrix)
(newline)
(let ((m (matrix-row-count matrix))
(n (matrix-column-count matrix)))
(do ((i O (+ i 1)))
((>= i m))
(display (matrix-ref matrix i 0))
(do ((j 1 (+ j 1)))
((>= j n))
(display \#\tab)
(display (matrix-ref matrix i j)))
(newline))))
(define matrix-get-column matrix:nth-col)
;;; This can come in handy sometimes:
(define (make-matrix m n)
(generate-matrix m n (lambda (i j) 0)))
;;; As can this:
(define det matrix:determinant)
;;; And to solve linear equations:
(define (lu-solve eqs . whatever)
(let ((m (matrix-row-count eqs))
(n (matrix-column-count eqs)))
(if (= n (+ m 1))
(let ((v (make-vector m))
(A (make-matrix m m)))
(do ((i O (+ i 1)))
((>= i m))
(do ((j 0 (+ j 1)))
((>= j m))
(matrix-set! A i j (matrix-ref eqs i j)))
(vector-set! v i (matrix-ref eqs i m)))
(matrix:solve-linear-system A v))
(error "Input has incorrect dimensions. -- LU-SOLVE"))))
;;; And something else as well... (This is getting out of hand!)
(define (apply-affine-transformation A b v)
(vector:+ (matrix:matrix*vector A v) b))

```

\section*{C.1.65 tangent.scm}
;;; This file defines tangent bundles as vector bundles (see vbundle.scm).
```

(declare (usual-integrations))
;;; Make some tangent vectors
(define (make-tangent chart p v)
;; p is the (abstract) point to which v is tangent, and v is the *coordinate
;; representation* of the tangent vector in the coordinates provided by the
;; given chart.
(vector 'tangent chart p v))
(define (tangent? x)
(and (vector? x)
(> (vector-length x) 0)
(eq? 'tangent (vector-ref x 0))))
(define (tangent:get-chart v)
(vector-ref v 1))
(define (tangent:get-anchor v)
(vector-ref v 2))
(define (tangent:get-coords v)
(vector-ref v 3))
(define (tangent:dimension v)
(vector-length (tangent:get-coords v)))
(define (make-binary-tangent-operation op)
(lambda (v w)
(let ((p (tangent:get-anchor v))
(q (tangent:get-anchor w)))
(if (equal? p q)
(let ((chart (tangent:get-chart v)))
(make-tangent chart
P
(op (tangent:get-coords v)
(chart:push-forward w chart))))
(error "Cannot add vectors tangent to different points."))))
(define tangent+ (make-binary-tangent-operation vector:+))
(define tangent- (make-binary-tangent-operation vector:-))
(define (tangent* a v)
(make-tangent (tangent:get-chart v)
(tangent:get-anchor v)
(vector:* a (tangent:get-coords v))))
;;; We can measure the distortion by how close the composition of a coordinate
;;; map with its "inverse" comes to the identity...
(define (local-distortion chart tangent)
(let ((f (chart:make-transition-map chart chart))
(x (chart:point->coords (tangent:get-anchor tangent) chart))
(v (chart:push-forward tangent chart)))
(vector:distance v (((diff f) x) v))))

```
```

(define distorted?
(let ((close-enuf? (make-comparator 1e-5)))
(lambda (chart v)
(not (close-enuf? (local-distortion chart v) 0))))
;;; Push a tangent vector along a chart:
(define (chart:push-forward tv chart)
(let ((other (tangent:get-chart tv))
(v (tangent:get-coords tv)))
(if (eq? chart other)
v
(push-forward-in-coords
(chart:make-transition-map other chart)
(chart:point->coords (tangent:get-anchor tv) other)
v))))
(define (push-forward-in-coords f x v)
(((diff f) x) v))
;;; Tangent charts:
(define (make-tangent-chart chart)
(let ((new-chart (chart:get-extra chart 'tangent-chart)))
(if new-chart
(force new-chart)
(make-new-tangent-chart chart))))
(define (make-new-tangent-chart chart)
(let* ((dim (chart:dimension chart))
(2*dim (* 2 dim))
(in-M-domain? (chart:get-membership-test chart))
(in-M-range? (chart:get-range-test chart))
(M-map (chart:get-coord-map chart))
(M-inverse (chart:get-inverse-map chart))
(dim-vector? (make-euclidean-test dim))
(2*dim-vector? (make-euclidean-test 2*dim)))
(letrec
((in-domain?
(lambda (v)
(and (in-M-domain? (tangent:get-anchor v))
(dim-vector? (tangent:get-coords v)))))
(in-range?
(lambda (v)
(and (2*dim-vector? v)
(in-M-range? (vector-head v dim)))))
(coord-map
(lambda (v)
(vector-append (M-map (tangent:get-anchor v))
(chart:push-forward v chart))))

```
```

            (inverse-map
            (lambda (x)
                (make-tangent chart
                                    (M-inverse (vector-head x dim))
                                    (vector-end x dim))))
            (transition
            (lambda (Tother)
                (let* ((other (chart:get-base-chart Tother))
                    (f (chart:make-transition-map chart other)))
                    (lambda (x)
                            (let ((anchor (vector-head x dim))
                            (tangent (vector-end x dim)))
                    (vector-append (f anchor)
                                    (push-forward-in-coords
                                    f anchor tangent)))))))
            (let ((new-chart (make-chart 2*dim in-domain? in-range?
                    coord-map inverse-map transition)))
    (chart:install-extra new-chart 'base-chart (delay chart))
    (chart:install-extra chart 'tangent-chart (delay new-chart))
    new-chart))))
    (define (chart:get-base-chart chart)
(let ((result (chart:get-extra chart 'base-chart)))
(if result
(force result)
\#f)))
;;; This is sometimes useful for procedures (such as vector fields) that need
;;; to explicitly manipulate charts on tangent bundles:
(define (make-tangent-chart-finder find-chart-in-M)
(let ((chart-finder
(lambda (x . aux)
(let ((chart (apply find-chart-in-M
(cons (tangent:get-anchor x) aux))))
(if chart
(make-tangent-chart chart)
\#f)))))
chart-finder))
;;; Here's how we make a tangent bundle:
(define (make-tangent-bundle M)
(let ((TM (manifold:get-extra M 'tangent-bundle)))
(if TM
(force TM)
(make-new-tangent-bundle M))))
(define (make-new-tangent-bundle M)
(let ((dim-M (manifold:dimension M)))
(let ((E
(let ((charts (manifold:get-finite-atlas M)))
(if charts
(charts->manifold (map (lambda (chart)

```
```

                        (make-tangent-chart chart))
                            charts))
    (let ((find-chart-in-M (manifold:get-general-chart-finder M))
        (minimize-in-M (manifold:get-general-minimizer M)))
    (letrec
        ((general-find-chart
                (lambda (p . predicates)
                (call-with-current-continuation
                (lambda (return)
                        (find-chart-in-M
                        (tangent:get-anchor p)
                            (lambda (chart)
                            (let ((new-chart (make-tangent-chart chart)))
                            (let valid? ((predicates predicates))
                                    (if (null? predicates)
                                    (return new-chart)
                        (if ((car predicates) new-chart)
                                    (valid? (cdr predicates))
                                    #f)))())))))
        (find-minimizing-chart
            (lambda (p f <)
                (cadr (minimize-in-M
                        (tangent:get-anchor p)
                        (lambda (chart)
                        (let ((new-chart
                            (make-tangent-chart chart)))
                            (list new-chart (f new-chart))))
                                (lambda (x y)
                            (< (cadr x) (cadr y)))))))
        (local-atlas-finder
            (lambda (p)
                (map (lambda (chart) (make-tangent-chart chart))
                    (manifold:get-local-atlas
                        M (tangent:get-anchor p))))))
        (make-manifold (* 2 dim-M)
                    general-find-chart
                    find-minimizing-chart
                    local-atlas-finder))))))
    (proj tangent:get-anchor)
    (fiber
    (lambda (p)
        (make-fiber tangent+ tangent- tangent*
            (lambda (v)
                (equal? p (tangent:get-anchor v)))))))
    (let ((TM (make-vector-bundle M E proj fiber)))
(manifold:install-extra M 'tangent-bundle (delay TM))
TM))))

```

\section*{C.1.66 vbundle.scm}
```

;;; Definitely need cotangent bundles!
;;; OLD COMMENTS THAT ARE STILL RELEVANT:
;;; We Shouldn't be too particular about vector representations, and should
;;; leave several different options.
;;;
;;; The differential operator representation is a good one, though, since via
;;; compositions we can generate most other differential operators of interest.

```
;; \(\operatorname{OLD}\) COMMENTS THAT APPEAR TO BE IRRELEVANT:
;;; Some of the issues involved in making tangent bundles, etc.:
;;;
;;; Something that looks like vector bundles will allow us to implement vector
;;; fields and differential forms. Maybe differential operators, too.
;;
;; Should we make this more like a manifold?
;; ;
;;; Maybe the right thing is to make local trivializations a special kind of
;;; chart (just as product charts are). We need standard ways to attach and
;; ; access optional structures on charts (and manifolds), such as product
;;; structures, metric structures, symplectic structures, etc.
;;;
;; \(;\) Is the vector space structure necessary on the coordinate-free level?
;;;
;;; And when do we ever need to treat the bundle as a manifold?
;;;
;;; Make tangent and cotangent bundles? Metric tensors, symplectic forms,
;;; differential operators are all related to these bundles...
;;;
;;; GJS used nscmutils to directly differentiate the functions. Maybe we
;;; should consider using that at some point, too. Tangent vectors appear to
;; ; be represented by first-order differential operators on functions.
;;
;; ; BUT, we don't know that charts always map to an imbedding of the manifold
;;; in a euclidean space! What if it uses some other representation on the
;; ; other end? We can numerically differentiate transition maps, but in
;;; general not charts. So use some other representation of tangent vectors
;;; and vector fields?
;;
;; Chart and vector may be the best represenatation; that appears to be what
;;; GJS uses, too.
(declare (usual-integrations))
(load "tangent")
(load "imbedding")
(load "cotangent")
;;; Abstract vector bundles:
(define (make-vector-bundle ME proj fiber)
    ; ; PROJ takes a pair ( \(x, v\) ) and returns \(x\), FIBER yields the operations for the
    ; ; vector space structure on the fiber above \(x\).
```

    (let ((dim (- (manifold:dimension E) (manifold:dimension M))))
        (manifold:install-extra E 'vector-bundle (vector (delay M)
                                    proj
                                    fiber
                                    dim)))
    E)
    (define (vbundle:get-manifold E)
(let ((structs (manifold:get-extra E 'vector-bundle)))
(if structs
(force (vector-ref structs 0))
\#f)))
(define (vbundle:get-fiber-map E)
(let ((structs (manifold:get-extra E 'vector-bundle)))
(if structs
(vector-ref structs 2)
\#f)))
(define (vbundle:get-projection E)
(let ((structs (manifold:get-extra E 'vector-bundle)))
(if structs
(vector-ref structs 1)
\#f)))
(define (vbundle:dimension E)
(let ((structs (manifold:get-extra E 'vector-bundle)))
(if structs
(vector-ref structs 3)
\#f)))
;;; Abstract vector spaces (fibers of the bundle) above each point of the
;;; manifold:
(define (make-fiber + - * member?)
(vector + - * member?))
(define (fiber:get+ fiber)
(vector-ref fiber 0))
(define (fiber:get- fiber)
(vector-ref fiber 1))
(define (fiber:get* fiber)
(vector-ref fiber 2))
(define (fiber:get-membership-test fiber)
(vector-ref fiber 3))
(define (fiber:+ fiber v w)
((fiber:get+ fiber) v w))
(define (fiber:- fiber v w)
((fiber:get- fiber) v w))
(define (fiber:* fiber a v)
((fiber:get* fiber) a v))

```
```

(define (fiber:member? v fiber)

```
    ((fiber:get-membership-test fiber) v))

\section*{C. 2 Finite elements}

The following programs implement a finite element solver that can be used both as a standalone program and as part of the coordinate-independent solver listed in the prevoius section. It implements the Rayleigh-Ritz method, described in Appendix A. The interested reader should begin with fem.scm to understand the top-level structure of the program, and refer to load.scm for dependencies.

\section*{C.2.1 2d-domains.scm}
```

;;; This file uses the Delaunay triangulation code to construct two-dimensional
;;; domains of solution. It defines constructors for different types of
;;; domains of solution, and also defines ways of setting different kinds of
;;; boundary conditions.
(declare (usual-integrations))

```
;; A generic template for making domains. Note that this is limited to planar
;;; regions. For higher dimensions, we need to handle higher-dimensional
;;; simplices, not just faces and edges...
(define (domain-maker make-vertex-nodes
    make-edge-nodes
    make-interior-nodes
    tesselate
    make-element
    make-boundary)
; ; MAKE-VERTEX-NODES is given the arguments to MAKE-DOMAIN, and should return
; ; a vector of (vertex) nodes.
; MAKE-EDGE-NODES should return a list of edge nodes on the edge defined by
; ; the two given nodes.
MAKE-INTERIOR-NODES should make interior nodes in the element defined by a
; ; list of given nodes.
    TESSELATE takes a vector of nodes and returns a pair containing a list of
    edges and a list of faces, in that order. (Edges are defined by pairs of
    nodes, while faces are defined by triplets of nodes. Just like abstract
; ; simplicial complexes...)
    ; MAKE-BOUNDARY takes a vector of nodes and sets the appropriate ones to be
;; boundary nodes.
; See fem.scm for the definition of MAKE-ELEMENT.
(define (make-domain . args)
;; Make vertex nodes:
```

(write-line '(making nodes...))
(let* ((vertex-nodes (apply make-vertex-nodes args))
(vertex-count (vector-length vertex-nodes)))
;; Tesselate the vertex nodes:
(write-line '(tesselating nodes...))
(let* ((complex (tesselate vertex-nodes))
(edges (car complex))
(faces (cadr complex)))
;; Record some debugging information:
(set! *debugging-info* edges)
;; Create edge nodes:
(write-line '(creating edge nodes...))
(let ((edge-nodes (make-vector (choose (+ vertex-count 1) 2) '()))
(edge-index symmetric->vector-index))
(do ((i 0 (+ i 1)))
((>= i vertex-count))
(node:set-id! (vector-ref vertex-nodes i) i))
(for-each
(lambda (pair)
(vector-set! edge-nodes
(apply edge-index (map node:get-id pair))
(apply make-edge-nodes pair)))
edges)
;; Create interior nodes:
(write-line '(creating interior nodes and making elements...))
(let loop ((faces faces) (interior-nodes '()))
(if (not (null? faces))
(let* ((face (car faces))
(elist (append-map
(lambda (pair)
(vector-ref edge-nodes
(apply edge-index pair)))
(pairs (map node:get-id face))))
(ilist (apply make-interior-nodes (append face elist))))
(make-element face (append elist ilist))
(loop (cdr faces) (append ilist interior-nodes)))
(begin

```
                    ; We now need to combine the three lists of nodes into one
                    ; ; big list, and to figure out the boundary:
```

(write-line '(cleaning up...))
;; First, count the number of edge nodes and create a vector
;; to store the node:
(let ((edge-count (vector-length edge-nodes)))
(let loop ((count 0) (i 0))
(if (< i edge-count)
(loop (+ count (length (vector-ref edge-nodes i)))
(+ i 1))
(let* ((icount (length interior-nodes))
(nodes (make-vector
(+ vertex-count count icount))))
;; Report data:
(write-line '(,count edge nodes))
(write-line '(,icount interior nodes))
(write-line '(,vertex-count vertices))
;; Copy the vertex nodes:
(write-line '(copying vertices...))
(do ((i 0 (+ i 1)))
((>= i vertex-count))
(vector-set! nodes i
(vector-ref vertex-nodes i)))
;; Copy the edge nodes:
(write-line '(copying edge nodes...))
(let loop ((i 0) (j vertex-count))
(if (< i edge-count)
(loop (+ i 1)
(let loop ((l (vector-ref edge-nodes
i))
(j j))
(if (null? l)
j
(begin
(vector-set! nodes j (car 1))
(loop (cdr l) (+ j 1))))))))
;; Copy the interior nodes:
(write-line '(copying interior nodes...))
(let loop ((i (+ vertex-count count))
(1 interior-nodes))
(if (null? 1)
(begin

```
                                    ; ; Make boundary nodes:
                                    (write-line '(setting boundary...))
                                    (make-boundary nodes)
```

    ;; Sort and return nodes:
    (write-line '(sorting nodes...))
    (sort! nodes lexicographic<))
    (begin
(vector-set! nodes i (car l))
(loop (+ i 1) ((dr l))))))))))))))))

```
```

    make-domain)
    ```
```

;;; Some useful routines:
(define (make-no-edge-nodes v0 v1) '())
(define (make-no-interior-nodes . args) '())
(define (predicate->make-boundary boundary?)
(lambda (nodes)
(let ((n (vector-length nodes)))
(do ((i O (+ i 1)))
((>= i n))
(let ((node (vector-ref nodes i)))
(node:set-boundary! node (boundary? node)))))))
(define (make-midpoint-node v0 v1)
(list (make-node (/ (+ (node:get-x v0) (node:get-x v1)) 2.)
(/ (+ (node:get-y v0) (node:get-y v1)) 2.))))
(define (do-nothing-to-nodes nodes)
'done)

```
;;; Various procedures to help build domains:
;; Circular domains:
(define (make-circular-domain-vertices angular-count radial-count)
    (let* ((rcount+1 (+ radial-count 1))
                (count (+ (* rcount+1 angular-count) 1))
                (nodes (make-vector count))
                (dt (/ (* 2 3.141592653589793) angular-count))
                (dr (/ . 5 rcount +1 )))
        (vector-set! nodes 0 (make-node . 5 .5))
        (let next-ray ( (i 0) (count 1))
            (if (< i angular-count)
                (let ((t (* i dt)))
                (let next-node ((j 1) (count count))
                        (if ( \(<=\mathrm{j}\) rcount +1 )
                        (let ( \(r\) (* j dr)))
                            (vector-set! nodes count
                        (make-node (+ .5 (* r (cos t)))
                                    \((+.5(* r(\sin t)))))\)
                            (next-node (+ j 1) (+ count 1)))
                    (next-ray (+ i 1) count))))
```

            nodes))))
    (define (circular-boundary? node)
(let ((x (node:get-x node))
(y (node:get-y node)))
(almost-zero? (- . 5 (sqrt (+ (square (- x .5)) (square (- y .5))))))))
;;; Square domains:
;;; We need to play a trick to keep track of the number of nodes in the square
;;; between calls to MAKE-VERTEX-NODES and TESSELATE in MAKE-ELEMENT; we should
;;; find some way to restructure MAKE-ELEMENT so that this is not necessary.
(define square-domain-constructor
(let ((width 0)
(height 0)
(h 0.)
(k 0.))
(list
(lambda (m n)
(set! width (+ m 2))
(set! height (+ n 2))
(let ((nodes (make-vector (* width height))))
(set! h (exact->inexact (/ (+ m 1))))
(set! k (exact->inexact (/ (+ n 1))))
(write-line '(h/k = ,(/ h k)))
(do ((i 0 (+ i 1)))
((>= i width) nodes)
(do ((j 0 (+ j 1)))
((>= j height))
(vector-set! nodes (+ (* i height) j)
(make-node (* i h) (* j k)))))))
(lambda (nodes)
;; Triangulate:
(let ((height-1 (- height 1))
(width-1 (- width 1))
(get-node (lambda (i) (vector-ref nodes i))))
(let column ((i 0) (results '(()())))
(if (< i width)
(column
(+ i 1)
(let row ((j 0) (edges (car results)) (faces (cadr results)))
(if (< j height)
(let ((me (+ (* i height) j))
(north (+ (* i height) j 1))
(east (+ (* i height) j height))
(ne (+ (* i height) j height 1)))
(if (= j height-1)
(list edges faces)
(if (= i width-1)
(row (+ j 1)
(cons (map get-node (list me north))

```
```

        edges)
    faces)
    (let ((f1 (list (vector-ref nodes me)
                        (vector-ref nodes north)
                        (vector-ref nodes ne)))
        (f2 (list (vector-ref nodes me)
        (vector-ref nodes ne)
        (vector-ref nodes east))))
    (row (+ j 1)
        (append (map
            (lambda (1)
                    (map get-node 1))
                            (list (list me north)
                                    (list north ne)
                                    (list me ne)
                                    (list ne east)
                                    (list me east)))
            edges)
        (cons f1 (cons f2 faces))))))))))
    results)))(
    (lambda () (list h k)))))
    (define make-square-domain-vertices
(car square-domain-constructor))
(define square-domain-triangulation
(cadr square-domain-constructor))
(define square-domain-element-size
(caddr square-domain-constructor))
(define (dirichlet-boundary? node)
(let ((x (node:get-x node))
(y (node:get-y node)))
(memq \#t (map almost-zero? (list x y (- 1 x) (- 1 y))))))
(define (cauchy-boundary? node)
(let ((x (node:get-x node))
(t (node:get-y node))
(threshold (* . 75 (cadr (square-domain-element-size)))))
(or (memq \#t (map almost-zero? (list x (- 1 x))))
(< t threshold))))
;;; Square domain with right triangle attached:
(define (hat-boundary? node)
(let* ((x (node:get-x node))
(t (node:get-y node))
(sizes (square-domain-element-size))
(dx (car sizes))
(dt (cadr sizes))
(m (/ dt dx))
(b (-1 m)))
(or (< t (* . 75 dt))
(> x (- 1 (* . 25 dx)))
(> t (- (+ (* x m) b) (* . 25 dt)))

```
```

(< x (* . 25 dx)))))

```
```

;;; Random domains:
(define (make-random-square-domain-vertices n)
(let* ((border (inexact->exact (floor (sqrt n))))
(size (+ n (* 4 border) 4))
(nodes (make-vector size)))
(write-line '(border: ,(+ (* 4 border) 4) total: ,size))
;; Create the border (top and bottom):
(let ((h (/ 1. (+ border 1))))
(let ((border+2 (+ border 2)))
(do ((i 0 (+ i 1)))
((>= i border+2))
(let ((new-node (make-node (* i h) 0.)))
(vector-set! nodes (* 2 i) new-node))
(let ((new-node (make-node (* i h) 1.)))
(vector-set! nodes (+ (* 2 i) 1) new-node))))
;; Left and right:
(let ((2*border+2 (+ (* 2 border) 2)))
(do ((j 1 (+ j 1)))
((> j border))
(let ((new-node (make-node 0. (* j h))))
(vector-set! nodes (+ (* 2 j) 2*border+2) new-node))
(let ((new-node (make-node 1. (* j h))))
(vector-set! nodes (+ (* 2 j) 2*border+2 1) new-node))))
;; Make internal nodes:
(do ((i (+ (* 4 border) 4) (+ i 1)))
((>= i size) nodes)
(let* ((x (random 1.))
(y (random 1.))
(new-node (make-node x y)))
(vector-set! nodes i new-node)))))
(define (make-random-domain-vertices n)
(let ((nodes (make-vector n)))
(do ((i O (+ i 1)))
((>= i n) nodes)
(let* ((x (random 1.))
(y (random 1.))
(new-node (make-node x y \#f)))
(vector-set! nodes i new-node)))))
(define (random-domain-triangulation nodes)
;; Triangulate:
(let ((n (vector-length nodes)))
(do ((i O (+ i 1)))
((>= i n))
(node:set-boundary! (vector-ref nodes i) \#f)))

```
```

    (let ((chull (convex-hull nodes)))
    (write-line '(the convex hull has ,(length chull) nodes...))
    (for-each
        (lambda (e) (node:set-boundary! (org e) #t))
        chull))
    (list (map (lambda (e)
                (list (org e) (dest e)))
                (list-edges))
            (map (lambda (f)
                (map org f))
                (list-faces))))
    (define (make-not-so-random-domain-vertices n max-r)
(let ((nodes (make-vector n)))
(do ((i 0 (+ i 1)))
((>= i n) nodes)
(let try ((x (random 1.)) (y (random 1.)))
(let ((new-node (make-node x y \#f)))
(let loop ((j 0))
(if (< j i)
(if (< (nodal-distance new-node (vector-ref nodes j)) max-r)
(try (random 1.) (random 1.))
(loop (+ j 1)))
(vector-set! nodes i new-node))))))))
;;; Making triangular domains:
(define triangular-domain-constructor
(let ((dx 0.)
(dy 0.))
(list
(lambda (n)
;; N is the number of nodes along the base.
(let ((nodes (make-vector (/ (* n (+ n 1)) 2))))
(set! dx (/ 1. (- n 1)))
(set! dy dx)
(write-line '(making ,(* n (- (* 2 n) 1)) nodes...))
(let ((count 0))
(do ((i O (+ i 1))
(y 0. (+ y dy))
(row (- n 1) (- row 1)))
((>= i n) nodes)
(do ((j O (+ j 1))
(x (/ y 2) (+ x dx)))
((> j row))
(vector-set! nodes count (make-node x y))
(set! count (+ count 1))))))
(lambda () (list dx dy))))
(define make-triangular-domain-vertices (car triangular-domain-constructor))
(define triangular-domain-element-size (cadr triangular-domain-constructor))

```
```

(define (triangular-boundary? node)
(let ((dt (cadr (triangular-domain-element-size)))
(x (node:get-x node))
(t (node:get-y node)))
(or (memq \#t (map almost-zero? (list (- t (* 2 x)) (- t (- 2 (* 2 x))))))
(< t (* . 75 dt)))))
;;; Construct a true hat domain:
(define *hat-vertices-data* '())
(define (make-hat-vertices t-count x-count ratio)
;; Build a "hat domain," where the rectangular part is the unit square and
;; contains X-COUNT by T-COUNT nodes. RATIO is the slope of the leg of the
;; hat.
(let* ((node-count (+ (* t-count x-count) (/ (* x-count (- x-count 1)) 2)))
(nodes (make-vector node-count))
(dx (/ 1. (- x-count 1)))
(dt (/ 1. (- t-count 1))))
(write-line '(,node-count nodes))
(set! *hat-vertices-data* (list dx dt ratio))
;; The square part:
(do ((i 0 (+ i 1)))
((>= i x-count))
(do ((j 0 (+ j 1)))
((>= j t-count))
(let ((x (* i dx))
(t (* j dt)))
(vector-set! nodes (+ (* i t-count) j) (make-node x t)))))
;; The triangle:
(let ((count (* t-count x-count)))
(set! dt (* ratio dx))
(do ((j 1 (+ j 1)))
((>= j x-count))
(let* ((t (+ 1 (* j dt)))
(x0 (/ (- t 1) ratio 2)))
(do ((i 0 (+ i 1)))
((>= i (- x-count j)))
(let ((x (+ x0 (* i dx))))
(vector-set! nodes count (make-node x t))
(set! count (+ count 1))))))
nodes))
(define (true-hat-boundary? node)
(let ((dx (car *hat-vertices-data*))
(dt (cadr *hat-vertices-data*))
(ratio (caddr *hat-vertices-data*)))
(let ((x (node:get-x node))

```
```

    (t (node:get-y node))
    (dx/4 (/ dx 4))
    (3/4*dt (* 3/4 dt))
    (dx/8 (/ dx 8)))
    (or (and (<= t (+ 1 dx/4))
(or (<= t 3/4*dt)
(<< x dx/4)
(>= x (- 1 dx/4))))
(and (>= t 1)
(or (and (<= x (+ .5 dx/8))
(<= x (+ (/ (- t 1) ratio 2) dx/8)))
(and (>= x (-.5 dx/8))
(>= x (- (- 1 (/ (- t 1) ratio 2)) dx/8))))))))

```

\section*{C.2.2 2d-examples.scm}
```

;;; This file defines some examples of PDEs over planar regions, particularly
;;; the unit square. This file goes with 2d-domains.scm and 2d-basis.scm.
;;; Some methods for constructing elements:
;;; For Laplace's equation:
(define make-laplacian-element
(element-maker laplacian
make-triangular-integrator
make-polynomial-basis-function))
;;; For the linear wave equation:
(define (make-wave-element-with-coeff c)
(element-maker (make-wave-operator c)
make-triangular-integrator
make-polynomial-basis-function))
(define *wave-constant* 1.00001)
(define make-wave-element (make-wave-element-with-coeff *wave-constant*))
;;; For characteristic bending:
(define make-bent-element
(element-maker (make-bent-operator *wave-constant* . }5\mathrm{ 1.)
make-triangular-integrator
make-polynomial-basis-function))
;;; For testing real functions:
(define make-real-laplacian-element
(element-maker real-laplacian
(trapezoidal-integrator-maker 16)
make-real-basis-function))

```
```

;;; Constructors for various test cases:
;;; Estimate the solution in the center of a disk, using N nodes on the
;;; boundary.
(define make-circular-domain
(domain-maker make-circular-domain-vertices
make-no-edge-nodes
make-no-interior-nodes
delaunay-triangulation
make-laplacian-element
(predicate->make-boundary circular-boundary?)))
;;; Make a square consisting of MxN interior nodes. M is the number of nodes
;;; along the x-axis, N is the number of nodes along the y-axis.
(define make-square-domain
(domain-maker make-square-domain-vertices
make-no-edge-nodes
make-no-interior-nodes
square-domain-triangulation
make-laplacian-element
(predicate->make-boundary dirichlet-boundary?)))
(define make-quadratic-domain
(domain-maker make-square-domain-vertices
make-midpoint-node
make-no-interior-nodes
square-domain-triangulation
make-laplacian-element
(predicate->make-boundary dirichlet-boundary?)))
(define make-wave-domain
(domain-maker make-square-domain-vertices
make-midpoint-node
make-no-interior-nodes
square-domain-triangulation
make-wave-element
(predicate->make-boundary cauchy-boundary?)))
(define make-dirichlet-wave-domain
(domain-maker make-square-domain-vertices
make-no-edge-nodes
make-no-interior-nodes
square-domain-triangulation
make-wave-element
(predicate->make-boundary dirichlet-boundary?)))
(define make-bent-domain
(domain-maker make-square-domain-vertices
make-midpoint-node
make-no-interior-nodes
square-domain-triangulation
make-bent-element
(predicate->make-boundary cauchy-boundary?)))
(define make-hat-domain
(domain-maker make-square-domain-vertices

```
```

make-midpoint-node
make-no-interior-nodes
square-domain-triangulation
(make-wave-element-with-coeff 4.)
(predicate->make-boundary hat-boundary?)))
(define make-true-hat-domain
(domain-maker make-hat-vertices
make-midpoint-node
make-no-interior-nodes
delaunay-triangulation
make-wave-element
(predicate->make-boundary true-hat-boundary?)))
;;; For testing real functions:
(define make-real-square-domain
(domain-maker make-square-domain-vertices
make-no-edge-nodes
make-no-interior-nodes
square-domain-triangulation
make-real-laplacian-element
(predicate->make-boundary dirichlet-boundary?)))
;;; A triangular domain:
(define make-triangular-domain
(domain-maker make-triangular-domain-vertices
make-midpoint-node
make-no-interior-nodes
delaunay-triangulation
(make-wave-element-with-coeff 1.)
(predicate->make-boundary triangular-boundary?)))
;; Now let's try a randomized distribution, using the Delaunay triangulation:
(define make-random-square-domain
(domain-maker make-random-square-domain-vertices
make-no-edge-nodes
make-no-interior-nodes
delaunay-triangulation
make-laplacian-element
(predicate->make-boundary dirichlet-boundary?)))
;;; The boundary, instead of the unit square, is the convex hull:
(define make-random-domain
(domain-maker make-random-domain-vertices
make-no-edge-nodes
make-no-interior-nodes
random-domain-triangulation
make-laplacian-element
do-nothing-to-nodes))

```
```

;;; Same thing, but the nodes shouldn't get too close:
(define make-not-so-random-domain
(domain-maker make-not-so-random-domain-vertices
make-no-edge-nodes
make-no-interior-nodes
random-domain-triangulation
make-laplacian-element
do-nothing-to-nodes))
;;; Some useful boundary/initial conditions:
(define potential
(let* ((pi 3.141592653589793)
(sinh (lambda (x) (/ (- (exp x) (exp (- x))) 2)))
(A (/ (sinh pi))))
(lambda (node)
(* A (sinh (* pi (node:get-y node))) (sin (* pi (node:get-x node)))))))
(define (wave node)
(let ((x (node:get-x node))
(t (node:get-y node)))
(cos (* 2 pi (- x (* t *wave-constant*))))))
(define (standing-wave node)
(* (sin (* 2 pi (node:get-x node)))
(sin (* 2 pi *wave-constant* (node:get-y node)))))
;;; A harmonic function for testing the programs:
(define (test-function node)
(let ((x (node:get-x node))
(y (node:get-y node)))
(*4 (- (square (- x . 5)) (square (- y .5))))))

```

\section*{C.2.3 2d-operators.scm}
```

;;; Some examples:
(declare (usual-integrations))

```
;;; The two-dimensional Laplcian:
(define (laplacian nodes)
    (make-operator
        poly-gradient
        poly-gradient
        (lambda (v w) (basis:scalar* -1 (basis:dot \(v\) w)))))
;;; The 1+1-dimensional d'Alembertian:
(define (make-wave-operator c)
    (lambda (nodes)
```

    (make-operator
    (lambda (f)
        (vector (d/dt f) (basis:* (- c) (d/dx f))))
    (lambda (f)
        (vector (d/dt f) (basis:* c (d/dx f))))
    (lambda (v w)
        (basis:* -1 (basis:dot v w))))))
    ```
```

;;; Characteristic bending: For t < t1, the operator agrees with the wave

```
;;; Characteristic bending: For t < t1, the operator agrees with the wave
;;; operator. For t > t2, the equation becomes elliptic. The characteristics
;;; operator. For t > t2, the equation becomes elliptic. The characteristics
;;; are "bent" between t1 and t2.
;;; are "bent" between t1 and t2.
(define (make-bent-operator c t1 t2)
(define (make-bent-operator c t1 t2)
    (let ((phi (make-bending-coeff t1 t2)))
    (let ((phi (make-bending-coeff t1 t2)))
        (lambda (nodes)
        (lambda (nodes)
            (let ((phi (function->poly phi nodes)))
            (let ((phi (function->poly phi nodes)))
                (make-operator
                (make-operator
                        (lambda (f)
                        (lambda (f)
                        (vector (basis:* phi (d/dt f)) (basis:* (- (square c)) (d/dx f))))
                        (vector (basis:* phi (d/dt f)) (basis:* (- (square c)) (d/dx f))))
                        (lambda (f)
                        (lambda (f)
                        (vector (d/dt f) (d/dx f)))
                        (vector (d/dt f) (d/dx f)))
                    (lambda (v w)
                    (lambda (v w)
                        (basis:* -1 (basis:dot v w))))))))
                        (basis:* -1 (basis:dot v w))))))))
;;; Let this be a polynomial for now:
(define (cut-off t)
    ;; This one lets the wave operator transition nicely into a "parabolic"
    ;; operator. (But without a time derivative!)
    (cond ((<= t 0) 1.)
        ((>= t 1) 0.)
        (else (+ (* 2 (cube t)) (* -3 (square t)) 1))))
(define (cut-off-1 t)
    ;; This one lets the wave operator transition into an elliptic operator.
    (if (> t 0)
        (- 1 (cube t))
        1.))
(define (make-bending-coeff t1 t2)
    (let ((delta (- t2 t1)))
        (lambda (node)
            (let ((t (node:get-y node)))
                (cut-off (/ (- t t1) delta))))))
;;; The Laplcian operator for "real" functions:
(define (real-laplacian nodes)
    (make-operator
    real-gradient
    real-gradient
    (lambda (v w) (basis:scalar* -1 (basis:dot v w)))))
```


## C.2.4 2d-poly-basis.scm

```
;; This file defines basis function constructors and integrator codes. What
;;; it provides is a way to handle functions over elements. Note that the code
;; ; in fem.scm operates independent of the representation we use here.
;; ; There is an abuse of terms here. By "basis function" we mean basis
;;; functions and their linear combinations. Thus, functions over elements
;;; represented by sums of basis functions are also considered basis functions.
;;; This code is specific to polynomial basis functions of two variables. The
;; ; polynomial code we have still needs lots of work, so we won't use it here.
;;; Most of the procedures operate directly on vector representations of basis
;; functions.
(declare (usual-integrations))
;;; Basic constructor:
(define (vector->poly v)
    (package-basis-function-methods
        '2d-poly-basis-function
        v
        (poly->function v)
        (make-2d-poly-adder v)
        (make-2d-poly-subtractor v)
        (make-2d-poly-multiplier v)
        (make-2d-poly-scalar-multiplier v)))
(define (make-polynomial-basis-function nodes center)
    (let* ( \(n\) (length nodes))
                (vals (make-vector n))
                (points (make-vector n)))
        (let loop ((nodes nodes) (i 0))
            (if (null? nodes)
                    (vector->poly (poly:point-value->coeff vals points))
                        (let ((node (car nodes)))
                        (if (= i center)
                        (vector-set! vals i 1)
                            (vector-set! vals i 0 ))
                            (vector-set! points i (vector (node:get-x node) (node:get-y node)))
                            (loop (cdr nodes) (+ i 1)))))))
(define (make-2d-poly-adder v)
        (lambda (w)
        (vector->poly (poly:+ v (basis:get-rep w)))))
(define (make-2d-poly-subtractor v)
    (lambda (w)
        (vector->poly (poly:- v (basis:get-rep w))))
(define (make-2d-poly-multiplier v)
    (lambda (w)
        (vector->poly (poly:* v (basis:get-rep w)))))
(define (make-2d-poly-scalar-multiplier v)
```

```
    (lambda (a)
        (vector->poly (poly:scalar* a v))))
;;; A slightly different kind of constructor:
(define (function->poly f nodes)
    (let* ((n (length nodes))
                (vals (make-vector n))
                    (points (make-vector n)))
        (let loop ((i 0) (nodes nodes))
            (if (null? nodes)
                (vector->poly (poly:point-value->coeff vals points))
                (let ((node (car nodes)))
                (vector-set! points i (node:get-coords node))
                (vector-set! vals i (f node))
                (loop (+ i 1) (cdr nodes)))))))
;;; And its inverse:
(define (poly->function f)
    (lambda (x)
        (poly:evaluate f x)))
(define (poly:evaluate f x)
    (vector-first (poly:coeff->point-value f (vector x))))
;;; Operations on basis functions:
(define (poly:+ v w)
    (let ((m (vector-length v))
                (n (vector-length w)))
        (let ((m (max m n))
                (n (min m n))
                (v (if (>= m n) v w))
                (w (if (< m n) v w)))
            (let ((result (make-vector m)))
                (do ((i 0 (+ i 1)))
                    ((>= i n))
                (vector-set! result i (+ (vector-ref v i) (vector-ref w i))))
            (do ((i n (+ i 1)))
                            ((>= i m) result)
                (vector-set! result i (vector-ref v i)))))))
(define (poly:scalar* a v)
    (let* ((n (vector-length v))
                (w (make-vector n)))
            (do ((i 0 (+ i 1)))
                        ((>= i n) w)
            (vector-set! w i (* a (vector-ref v i))))))
(define (poly:* p1 p2)
    (let* ((n1 (vector-length p1))
                (n2 (vector-length p2))
                (degree (+ (poly:degree p1) (poly:degree p2)))
```

```
            (n (choose (+ degree 2) 2))
            (p (make-vector n 0)))
        (do ((i 0 (+ i 1)))
            ((>= i n1) p)
        (let ((powers (zig-zag i))
            (coeff (vector-ref p1 i)))
            (do ((j 0 (+ j 1)))
                ((>= j n2))
            (let ((k (apply inverse-zig-zag (map + powers (zig-zag j)))))
                (vector-set! p k (+ (vector-ref p k)
                                    (* coeff (vector-ref p2 j)))))))))
(define (poly:- v w)
    (poly:+ v (poly:scalar* -1 w)))
(define (poly-basis:partial k)
    (if (not (or (= k 0) (= k 1)))
        (error "Only (partial 0) and (partial 1) exist (for now)! -- PARTIAL")
        (let ((select (if (= k 0) car cadr)))
            (lambda (v)
                (let* ((v (basis:get-rep v))
                    (n (vector-length v))
                            (w (make-vector n 0)))
                        (do ((i 0 (+ i 1)))
                        ((>= i n) (vector->poly w))
                    (let ((powers (zig-zag i)))
                    (if (> (select powers) 0)
                                    (vector-set!
                                    w (- i (apply + powers) k)
                                    (* (select powers) (vector-ref v i)))))))))))
;;; Some useful definitions in two dimensions:
(define d/dx (poly-basis:partial 0))
(define d/dy (poly-basis:partial 1))
(define d/dt d/dy)
;;; Converting between point-value and coefficient representations; is there a
;;; higher-dimensional analog of the FFT trick? Point-value representation is
;;; great for everything *except* differentiation...
(define (poly:coeff->point-value v sample-points)
    (let* ((n (vector-length sample-points))
            (w (make-vector n))
            (m (vector-length v)))
        (do ((i 0 (+ i 1)))
            ((>= i n) w)
            (let* ((coords (vector-ref sample-points i))
                    (x (vector-ref coords 0))
                        (y (vector-ref coords 1)))
            (let loop ((j 0) (sum 0.))
                        (if (< j m)
                        (let ((powers (zig-zag j)))
```

```
        (loop (+ j 1) (+ sum (* (vector-ref v j)
                                    (expt x (car powers))
                                    (expt y (cadr powers))))))
    (vector-set! w i sum))))))
(define (poly:point-value->coeff w sample-points)
    (let* ((n (vector-length sample-points))
            (A (make-matrix n (+ n 1))))
        (let next-row ((i 0))
            (if (< i n)
                (let* ((coords (vector-ref sample-points i))
                                    (x (vector-ref coords 0))
                                    (y (vector-ref coords 1)))
                            (let next-column ((j 0))
                        (if (< j n)
                            (let* ((powers (zig-zag j))
                                    (p (car powers))
                                    (q (cadr powers)))
                                    (matrix-set! A i j (* (expt x p) (expt y q)))
                                    (next-column (+ j 1)))
                                    (begin
                                    (matrix-set! A i n (vector-ref w i))
                                    (next-row (+ i 1))))))
                            (lu-solve A 'no-copy)))))
(define (poly:slow-make-sample-points n)
    (if (> n 0)
            (let* ((delta (exact->inexact (/ n)))
                                    (n (choose (+ n 2) 2))
                            (v (make-vector n)))
                    (do ((i 0 (+ i 1)))
                            ((>= i n) v)
                        (let ((powers (zig-zag i)))
                            (vector-set! v i (vector (* (car powers) delta)
                                    (* (cadr powers) delta))))))
            (vector (vector 0 0))))
(define poly:make-sample-points
    (simple-memoize poly:slow-make-sample-points 10))
;;; Some useful operations on basis functions:
(define (poly:degree p)
    (apply + (zig-zag (- (vector-length p) 1))))
(define (poly:coeff->expr v)
    (let* ((v (basis:get-rep v))
            (n (vector-length v)))
        (let loop ((expr '()) (i (- n 1)))
            (if (>= i 0)
                (let ((powers (zig-zag i)))
                            (loop (cons '((x ,(car powers) y ,(cadr powers)),(vector-ref v i))
                                    expr)
                            (- i 1)))
                    expr))))
```

```
;;; The truly messy stuff: Integrals! This needs to run a lot faster. What
;;; about doing away with the coordinate transformations?
(define (make-triangular-integrator vertex-nodes)
    ;; We assume that there are three vertex nodes, and that the triangle they
    ;; form is the boundary of the element:
    (if (not (= (length vertex-nodes) 3))
        (error (string-append "Error: Elements must have three vertex nodes."
                        " -- MAKE-TRIANGULAR-INTEGRATOR")))
    (let ((p1 (car vertex-nodes))
        (p2 (cadr vertex-nodes))
        (p3 (caddr vertex-nodes)))
    ;; Find the absolute value of the Jacobian of the affine transformation
    ; mapping the reference triangle {(0,0),(1,0),(0,1)} to this triangle.
    (let* ((A (list->matrix
                2
                        (list
                    (- (node:get-x p2) (node:get-x p1))
                    (- (node:get-x p3) (node:get-x p1))
                    (- (node:get-y p2) (node:get-y p1))
                    (- (node:get-y p3) (node:get-y p1)))))
                (b (node:get-coords p1))
                (jacobian (abs (det A))))
        (define (integrate f . rest)
            (let* ((f (apply basis:* (cons f rest)))
                    (degree (poly:degree f))
                    (reference (poly:make-sample-points degree))
                    (n (choose (+ degree 2) 2))
                    (real (make-vector n)))
                (do ((i 0 (+ i 1)))
                            ((>= i n))
                        (vector-set! real i
                            (apply-affine-transformation
                                    A b (vector-ref reference i))))
                (* jacobian
                        (inner-product
                        (poly:point-value->coeff
                        (poly:coeff->point-value (basis:get-rep f) real) reference)
                        (make-reference-integrals degree)))))
        integrate)))
(define (slow-make-reference-integrals degree)
    (let* ((n (choose (+ degree 2) 2))
            (integrals (make-vector n)))
        (do ((i 0 (+ i 1)))
            ((>= i n) integrals)
            (vector-set! integrals i (apply reference-integral (zig-zag i))))))
```

(define make-reference-integrals

```
    (simple-memoize slow-make-reference-integrals 10))
(define (reference-integral m n)
    (let ((n+1 (+ n 1)))
        (let loop ((i 0) (sum 0.) (-1^i 1))
            (if (<= i n+1)
                    (loop (+ i 1)
                                    (+ sum (* (choose n+1 i) (/ (exact->inexact (+ i m 1))) -1^i))
                                    (* -1^i -1))
                    (/ sum n+1)))))
```

;; Zig-zag across the two-dimensional square lattice:
(define (slow-zig-zag n)
(let loop ( $(\mathrm{n} \mathrm{n}$ ) ( p 0 ) ( q 0 ))
(if (> n 0)
(if (zero? p)
(loop (- n 1) (+ q 1) 0)
(loop (- n 1) (- p 1) (+ q 1)))
(list p q))))
(define zig-zag (simple-memoize slow-zig-zag 20))
(define (inverse-zig-zag m $n$ )
(if (and (zero? m) (zero? n))
0
(+ (choose (+ m n 1) 2) n)) )
;; We need to define the gradient to help define the laplacian:
(define (poly-gradient f)
(vector (d/dx f) (d/dy f)))

## C.2.5 2d-real-basis.scm

;;; This file defines basis functions that are still polynomial, but are
;;; represented by real Scheme procedures and can thus undergo general
;;; coordinate transformations in a nice way. This is not so important here
;; ; (in fact, it is a slower and less accurate implementation), but is useful
;; for extending FEM to manifolds.
(declare (usual-integrations))
;;; Constructor:
(define (proc->real f)
(package-basis-function-methods '2d-real-basis-function
f
f
(make-real-adder f)
(make-real-subtractor f)
(make-real-multiplier f)
(make-real-scalar-multiplier f)))

```
(define make-real-basis-function
    (compose proc->real basis-function->function make-polynomial-basis-function))
;;; Operations on basis functions:
(define (make-real-adder f)
    (lambda (g)
        (let ((g (basis:get-rep g)))
            (proc->real
                (lambda (x)
                            (+ (f x) (g x)))))))
(define (make-real-subtractor f)
    (lambda (g)
        (let ((g (basis:get-rep g)))
            (proc->real
                    (lambda (x)
                        (- (f x) (g x)))))))
(define (make-real-multiplier f)
        (lambda (g)
            (let ((g (basis:get-rep g)))
                (proc->real
                (lambda (x)
                    (* (f x) (g x)))))))
(define (make-real-scalar-multiplier f)
    (lambda (a)
            (proc->real
                (lambda (x)
                    (* a (f x))))))
```

;; The gradient is needed for defining the laplacian:
(define (real-gradient f)
(let* ((f (compose vector (basis:get-rep f)))
(fx (proc->real (compose vector-first ((pdiff 0) f))))
(fy (proc->real (compose vector-first ((pdiff 1) f)))))
(vector fx fy))

## C.2.6 2d-real-diff.scm

;; ; This file loads the appropriate definitions for the numerical
;; ; differentiation of real functions.
(declare (usual-integrations))
(load "manifolds/linear")
(load "manifolds/lshared")
(load "manifolds/richardson")
;; ; Stolen from manifolds/misc.scm:
(define (pdiff i)

```
(lambda (f)
    (let ((df (diff f)))
        (lambda (x)
            (let ((v (make-vector (vector-length x) 0)))
                (vector-set! v i 1)
                ((df x) v))))))
```


## C.2.7 2d-trapezoid.scm

```
;;; This file defines a simple numerical integrator over triangular subregions
;;; of the plane. It uses the trapezoidal rule because that's the easiest
;;; thing to implement, and I'd just like to see if it improves FEM on
;;; manifolds.
(declare (usual-integrations))
;;; Heh heh...
;;; We can follow the same idea as in 2d-basis.scm: Map the triangular region
;;; to a standard isoceles triangle by an affine transformation and apply the
;;; trapezoidal rule. Note that its assumptions about basis functions are
;;; different.
(define (trapezoidal-integrator-maker count)
    ;; This parameter determines how many thingamajigs to use for integration.
    ;; It should really scale depending on the element, but for simplicity let's
    ;; keep it a constant (for now).
    (let* ((count-1 (- count 1))
        (h (/ 1. count))
        (area (/ (square h) 2)))
    (lambda (vertex-nodes)
        ;; We assume that there are three vertex nodes, and that the triangle
        ;; they form is the boundary of the element:
        (if (not (= (length vertex-nodes) 3))
        (error (string-append "Error: Elements must have three vertex nodes."
                                    " -- MAKE-TRIANGULAR-INTEGRATOR")))
        (let ((p1 (car vertex-nodes))
            (p2 (cadr vertex-nodes))
            (p3 (caddr vertex-nodes)))
            ;; Find the absolute value of the Jacobian of the affine transformation
            ;; mapping the reference triangle {(0,0),(1,0),(0,1)} to this triangle.
            (let* ((A (list->matrix
                2
                (list
                (- (node:get-x p2) (node:get-x p1))
                (- (node:get-x p3) (node:get-x p1))
                        (- (node:get-y p2) (node:get-y p1))
                        (- (node:get-y p3) (node:get-y p1)))))
                    (b (node:get-coords p1))
                    (jacobian (abs (det A)))
```

```
            (aff (lambda (x) (apply-affine-transformation A b x))))
            (lambda (f . rest)
            (let ((flist (map basis-function->function (cons f rest)))
                    (sum 0))
            (do ((i 0 (+ i 1)))
                    ((>= i count))
                    (let ((x1 (* i h))
                    (x2 (* (+ i 1) h)))
                    (do ((j (- count i 1) (- j 1)))
                    ((< j 0))
                            (let* ((y1 (* j h))
                                    (y2 (* (+ j 1) h))
                                    (ll (aff (vector x1 y1)))
                                    (lr (aff (vector x2 y1)))
                                    (ul (aff (vector x1 y2))))
                                    (set! sum (+ sum (trapezoidal-average
                                    flist (list ll lr ul))))
                                    (if (< (+ i j) count-1)
                                    (let ((ur (aff (vector x2 y2))))
                                    (set! sum (+ sum (trapezoidal-average
                                    flist (list ul ur lr))))))))))
            (* sum area jacobian))))))))
(define (trapezoidal-average flist plist)
    (let next-point ((plist plist) (sum 0) (count 0))
        (if (null? plist)
            (/ sum count)
            (let ((p (car plist)))
                    (let next-function ((flist flist) (prod 1))
                        (if (null? flist)
                        (next-point (cdr plist) (+ sum prod) (+ count 1))
                        (next-function (cdr flist) (* prod ((car flist) p)))))))))
```


## C.2.8 basis.scm

; ; We need to provide a common structure for handling basis functions. This ;; way, all that the user needs to change in order to change basis functions ;; ; is the constructor passed into the domain constructor.

```
(declare (usual-integrations))
```

; ; Basis functions need to carry around their own methods:
(define (package-basis-function-methods
type rep eval + - * scalar*)
(vector type rep eval + - * scalar* '()))
(define (basis-function? f)
(and (vector? f)
(= (vector-length f) 7)))
(define (basis:type f)

```
    (if (basis-function? f)
        (vector-ref f 0)
        #f))
(define (basis:get-rep f)
    (vector-ref f 1))
(define (basis-function->function f)
    (vector-ref f 2))
(define (basis:binary+ f g)
    (if (basis:same-type? f g)
            ((vector-ref f 3) g)
            (error "Cannot add basis functions of different types.")))
(define (basis:binary- f g)
    (if (basis:same-type? f g)
            ((vector-ref f 4) g)
            (error "Cannot subtract basis functions of different types.")))
(define (basis:binary* f g)
    (if (number? f)
        (if (number? g)
                        (* f g)
                        (basis:scalar* f g))
        (if (number? g)
            (basis:scalar* g f)
            (if (basis:same-type? f g)
                            ((vector-ref f 5) g)
                            (error "Cannot multiply basis functions of different types.")))))
(define (basis:scalar* a f)
    ((vector-ref f 6) a))
(define (basis:install-extra f tag datum)
    (let ((result (assq tag (vector-ref f 7))))
            (if result
                        (set-cdr! result datum)
                        (vector-set! f 7 (cons (cons tag datum) (vector-ref f 7))))))
(define (basis:get-extra f tag)
    (let ((result (assq tag (vector-ref f 7))))
        (if result
                        (cdr result)
                #f)))
```

;; $;$ Derived from the basic methods:
(define (basis:same-type? f g)
(eq? (basis:type f) (basis:type g)))
(define (evaluate-basis-function $f p$ )
((basis-function->function f) p))
(define (basis:+ f.rest)
(let loop ( $(f f)$ (l rest))
(if (null? 1)
$f$

```
            (loop (basis:binary+ f (car l)) (cdr l)))))
(define (basis:- f . rest)
    (if (null? rest)
        f
            (basis:binary- f (apply basis:+ rest))))
(define (basis:* f . rest)
    (let loop ((f f) (l rest))
        (if (null? l)
            f
                    (loop (basis:binary* f (car 1)) (cdr l)))))
(define (basis:dot v w)
    (let ((n (vector-length v)))
        (let loop ((i 1) (result (basis:* (vector-ref v 0) (vector-ref w 0))))
            (if (< i n)
                    (loop (+ i 1)
                            (basis:+ result (basis:* (vector-ref v i) (vector-ref w i))))
                    result))))
```


## C.2.9 bent.scm

```
;;; Drawing the bent characteristics:
;;; First, open graphics device and set scale:
(define dev (make-graphics-device 'x))
(graphics-set-coordinate-limits dev -. 1 -. 1 1.1 1.1)
(if #f
    (begin
                (graphics-operation dev 'set-background-color "black")
            (graphics-operation dev 'set-foreground-color "red")
            (graphics-operation dev 'set-mouse-color "white")))
;;; Next, define a procedure to integrate and draw the characteristics (using a
;;; silly foward-Euler integrator).
(define (draw-characteristic slope x0 t0 dt)
    (graphics-move-cursor dev x0 t0)
    (let loop ((i 1) (x x0))
        (let ((t (+ (* i dt) t0)))
            (if (and (<= 0 t) (<= t 1)
                        (<= 0 x) (<= x 1))
                            (let* ((dt/dx (slope t))
                                (new-x (+ x (/ dt dt/dx))))
                            (cond ((> new-x 1) (graphics-drag-cursor dev 1 t))
                                ((< new-x 0) (graphics-drag-cursor dev 0 t))
                                (else (graphics-drag-cursor dev new-x t)))
                            (loop (+ i 1) new-x)))))
        'done)
```

```
;;; The functions we want to use:
(define (cut-off t)
    (cond ((<= t 0) 1.)
                ((>= t 1) 0.)
                (else (+ (* 2 (expt t 3)) (* -3 (expt t 2)) 1))))
(define (f t)
    (cut-off (- (* 2 t) 1)))
(define (g t)
    (- (f t)))
;;; Something to generate sample points:
(define (samples min max count)
    (let ((dx (/ (- max min) (- count 1))))
                (let loop ((result '()) (i (- count 1)))
                (if (< i 0)
                        result
                        (loop (cons (+ (* i dx) min) result) (- i 1))))))
(define (constant-list val count)
    (vector->list (make-vector count val)))
;;; Do it!
(graphics-clear dev)
(graphics-draw-line dev 0. 0.0.1.)
(graphics-draw-line dev 0. 0. 1. 0.)
(graphics-draw-line dev 0. 0. 1. 0.)
(graphics-draw-line dev 1. 0. 1. 1.)
(graphics-draw-line dev 0. 1. 1. 1.)
(for-each
    (lambda (x t)
        (draw-characteristic f x t .01))
    (samples 0. 1. 11)
    (constant-list 0. 11))
(for-each
    (lambda (x t)
        (draw-characteristic g x t .01))
    (samples 0. 1. 11)
    (constant-list 0. 11))
(for-each
    (lambda (x t)
        (draw-characteristic f x t .01))
    (constant-list 0. 11)
    (samples 0. .9 11))
(for-each
    (lambda (x t)
        (draw-characteristic g x t .01))
    (constant-list 1. 11)
```

```
(samples 0. .9 11))
;(graphics-close dev)
```


## C.2.10 collect.scm

```
;; Collect some data for thesis work. First, load the FEM programs and set
;;; the speed of light to 1.
;; ; Here's how we run experiments:
(define (make-experiment domain-maker)
    (lambda (argl filename)
        ; Reload to clear hidden states:
        (load "load")
        (set! *wave-constant* 1.)
        (let ((make-domain (evaluate-symbol domain-maker)))
            (write-line '(constructing domain...))
            (let ((nodes (show-time (lambda () (apply make-domain argl)))))
                (write-line '(, (vector-length nodes) nodes constructed))
                (write-line '(constructing matrix...))
                        (let ((mat (show-time
                        (lambda ()
                            (sparse->matrix (fem 0-function nodes wave)))))
                        (write-line '(matrix size \(=\), (matrix-size mat)))
                (write-line '(solving equations...))
                (let ((v (show-time (lambda () (lu-solve mat)))))
                        (write-line '(computing results...))
                        (let ((results (compute-results nodes \(v\) wave)))
                        (write-line '(saving...))
                        (let ((port (open-output-file filename)))
                        (print-matrix results port)
                            ((close-output-port port)))))))))
(define (make-relaxing-experiment domain-maker)
        (lambda (argl sor-count sor-factor filename)
            ; Reload to clear hidden states:
            (load "load")
            (set! *wave-constant* 1.)
            (let ((make-domain (evaluate-symbol domain-maker)))
                (write-line '(constructing domain...))
                (let ((nodes (show-time (lambda () (apply make-domain argl)))))
                (write-line '(,(vector-length nodes) nodes constructed))
                (write-line '(constructing matrix...))
                (let ( (mat (show-time
                                    (lambda ()
                                    (sparse-normal-equations
                                    (fem 0-function nodes wave))))))
                                    (write-line '(matrix size \(=\),(sparse-matrix-size mat)))
                (write-line '(relaxing...))
```

```
(let ((v (show-time (lambda () (sor mat sor-count sor-factor))))
    (write-line '(computing results...))
    (let ((results (compute-results nodes v wave)))
        (write-line '(saving...))
        (let ((port (open-output-file filename)))
            (print-matrix results port)
            (close-output-port port)))))))))
```

```
;;; A couple of tests:
(define test-1 (make-experiment 'make-true-hat-domain))
(define test-2 (make-experiment 'make-bent-domain))
(define test-3 (make-relaxing-experiment 'make-true-hat-domain))
```


## C.2.11 debug.scm

;;; For debugging purposes, this program draws the output of the Delaunay ;;; triangulation program, TRIANGULATE, in delaunay.scm.
;;
; ; ; We assume that the sites are subsets of the unit square $[0,1] \times[0,1]$.
(declare (usual-integrations))
;; Draw the mesh:
(define *delaunay-device* 'undefined)
(define *debugging-info* '())
(define draw
(let ((background "black")
(cursor "white")
(line "blue")
(boundary "red")
(node "white")
(boundary-node "purple"))
(lambda (nodes)
(if (eq? *delaunay-device* 'undefined)
(set! *delaunay-device* (make-graphics-device 'x))
(graphics-clear *delaunay-device*))
(let* ((dev *delaunay-device*)
(b .02)
(elist (set-coordinate-limits dev nodes b .005))
(ex (car elist))
(ey (cadr elist)))
(graphics-operation dev 'set-foreground-color line)
(graphics-operation dev 'set-background-color background)
(graphics-operation dev 'set-mouse-color cursor)
(graphics-clear dev)
(for-each
(lambda (e)
(let* ( org-e (car e))
(dest-e (cadr e))
(org (node:get-coords org-e))

```
            (dest (node:get-coords dest-e))
            (org-boundary? (node:boundary? org-e))
            (dest-boundary? (node:boundary? dest-e)))
                    (if (and org-boundary? dest-boundary?)
                    (graphics-operation dev 'set-foreground-color boundary))
                    (graphics-move-cursor dev (vector-ref org 0) (vector-ref org 1))
                (graphics-drag-cursor dev (vector-ref dest 0) (vector-ref dest 1))
                    (if (and org-boundary? dest-boundary?)
            (graphics-operation dev 'set-foreground-color line))))
            *debugging-info*)
            (graphics-operation dev 'set-foreground-color node)
            (for-each
            (lambda (n)
            (let ((x (node:get-x n))
                    (y (node:get-y n)))
            (if (node:boundary? n)
                    (graphics-operation dev 'set-foreground-color boundary-node))
            (graphics-draw-line dev (- x ex) (- y ey) (+ x ex) (+ y ey))
            (graphics-draw-line dev (- x ex) (+ y ey) (+ x ex) (- y ey))
            (if (node:boundary? n)
                (graphics-operation dev 'set-foreground-color node))))
    (vector->list nodes))))))
(define (set-coordinate-limits dev nodes border edge)
    (apply
        (lambda (x-left y-bottom x-right y-top)
            (if (= x-left x-right)
                    (begin
                        (set! x-left (- x-left .5))
                        (set! x-right (+ x-right .5))))
            (if (= y-top y-bottom)
                    (begin
                        (set! y-bottom (- y-bottom .5))
                        (set! y-top (+ y-top .5))))
            (let ((dx (- x-right x-left))
                    (dy (- y-top y-bottom)))
                (graphics-set-coordinate-limits
                dev (- x-left (* dx border)) (- y-bottom (* dy border))
                (+ x-right (* dx border)) (+ y-top (* dy border)))
                (list (* edge dx) (* edge dy))))
            (bounding-box (vector->list nodes) node:get-coords)))
(define (close)
    (if (not (eq? *delaunay-device* 'undefined))
            (begin
```

```
        (graphics-close *delaunay-device*)
        (set! *delaunay-device* 'undefined))))
;;; Quick and easy way to dump a vector, matrix, whatever into a file:
(define (dump obj file-name)
    ;; Compound objects should always come before primitive ones, because they
    ;; may be implemented from primitives like LISTs or VECTORs.
    (cond ((matrix? obj)
            (write-line '(dumping matrix to file ,file-name))
            (let ((port (open-output-file file-name)))
                (print-matrix obj port)
                (close-output-port port)))
            ((sparse-matrix? obj)
            (write-line '(dumping sparse-matrix to file ,file-name))
            (let ((port (open-output-file file-name)))
                (print-sparse-matrix obj port)
                (close-output-port port)))
            ((vector? obj)
            (write-line '(dumping vector to file ,file-name))
            (let ((n (vector-length obj))
                (port (open-output-file file-name)))
                (do ((i O (+ i 1)))
                ((>= i n))
                    (display (vector-ref obj i) port)
                (newline port))
                    (close-output-port port)))
            ((list? obj)
            (write-line '(dumping list to file ,file-name))
            (let ((port (open-output-file file-name)))
                (for-each
                (lambda (x)
                        (display x port)
                (newline port))
            obj)
                (close-output-port port)))
            (else
            (error "Object must be a VECTOR, MATRIX, or LIST -- DUMP"))))
;;; Dump a matrix into a Maple-readable file:
(define matrix->maple
    (let ((variable-name "foo"))
        (lambda (A file-name)
            (let* ((port (open-output-file file-name))
                (print (lambda (obj) (display obj port))))
            (print (string-append variable-name " := ["))
```

```
(print #\newline)
(let ((m (matrix-row-count A))
            (n (matrix-column-count A)))
        (print (string-append "[" (number->string (matrix-ref A 0 0))))
        (do ((j 1 (+ j 1)))
            ((>= j n))
            (print (string-append "," (number->string (matrix-ref A 0 j)))))
        (print #\])
        (do ((i 1 (+ i 1)))
                ((>= i m))
            (print (string-append (list->string '(#\, #\newline #\[))
                                    (number->string (matrix-ref A i 0))))
            (do ((j 1 (+ j 1)))
                ((>= j n))
                (print (string-append "," (number->string (matrix-ref A i j)))))
            (print #\]))
        (print "]:")
        (print #\newline)
        (print (string-append variable-name
                        " := array("
                        variable-name
                        "):"))
        (print #\newline)
        (print (string-append variable-name
            " := [seq([seq([i, j, "
                        variable-name
            "[i, j]], i=1.."
            (number->string m)
            ") ], j=1.."
                        (number->string n)
                        ")]:"))
        (print #\newline)
        (print (string-append "plots[surfdata]("
                                    variable-name
                                    ", axes=frame, style=wireframe);"))
        (print #\newline))
(close-output-port port)))))
```

```
;;; Compute the error vector:
(define (compute-error nodes v f)
    ;; V should be the output of SOR.
    ;; F computes the initial condition/solution, given a node.
    (let ((e (make-matrix (vector-length v) 3))
        (size (vector-length nodes))
        (max 0)
        (max-index 0))
        (let loop ((i 0) (j 0))
            (if (< j size)
            (if (node:boundary? (vector-ref nodes j))
```

```
(loop i (+ j 1))
(begin
    (matrix-set! e i O (f (vector-ref nodes j)))
    (matrix-set! e i 1 (vector-ref v i))
    (matrix-set! e i 2 (abs (- (f (vector-ref nodes j))
                            (vector-ref v i))))
    (if (>= (matrix-ref e i 2) max)
                (begin
                    (set! max (matrix-ref e i 2))
            (set! max-index i)))
        (loop (+ i 1) (+ j 1))))
(begin
        (write-line '(maximum error: ,max at node ,max-index))
        e)))))
;;; Generate results that we can plot with MATLAB:
(define (compute-results nodes v f)
    ;; V should be the output of SOR.
    (let ((e (make-matrix (interior-node-count nodes) 4))
            (size (vector-length nodes)))
        (let loop ((i 0) (j 0))
            (if (< j size)
                (if (node:boundary? (vector-ref nodes j))
                        (loop i (+ j 1))
                        (let ((node (vector-ref nodes j)))
                        (matrix-set! e i 0 (node:get-x node))
                                (matrix-set! e i 1 (node:get-y node))
                                (matrix-set! e i 2 (vector-ref v i))
                                (matrix-set! e i 3 (f node))
                                (loop (+ i 1) (+ j 1))))
                e))))
(define (interior-node-count nodes)
    (let ((n (vector-length nodes)))
        (let loop ((count 0) (i 0))
            (if (< i n)
                        (if (node:boundary? (vector-ref nodes i))
                            (loop count (+ i 1))
                            (loop (+ count 1) (+ i 1)))
                        count))))
;;; Save the data back to the nodes:
(define (store-results! nodes v)
    ;; V should be the output of SOR.
    (let ((size (vector-length nodes)))
        (let loop ((i 0) (j 0))
            (if (< j size)
                        (if (node:boundary? (vector-ref nodes j))
                            (loop i (+ j 1))
                        (begin
                                    (node:set-value! (vector-ref nodes j) (vector-ref v i))
                                    (loop (+ i 1) (+ j 1))))
                            (write-line '(,i interior nodes))))))
```


## C.2.12 delaunay.scm

```
;;; This implements the Delaunay triangulation algorithm described in:
;;;
;;; "Primitives for the manipulation of general subdivisions and the
;;; computation of Voronoi diagrams," ACM transactions on graphics, Vol. 4,
;;; No. 2, April 1985, P. 74-123.
;;; Leonidas Guibas and Jorge Stolfi, Xerox PARC and Stanford University.
;;; There is something better now; take a look at the Geometry Center's home
;;; page at "http://www.geom.umn.edu/", and check out the quickhull algorithm.
(declare (usual-integrations))
;;; This is kind of useful for the FEM stuff:
(define (delaunay-triangulation nodes)
    (triangulate nodes)
    (list (map (lambda (e)
                            (list (org e) (dest e)))
            (list-edges))
            (map (lambda (f)
                            (map org f))
                    (list-faces))))
;;; This is the divide-and-conquer algorithm. The nodal data structure should
;;; provide the methods NODE:GET-X and NODE:GET-Y.
(define (triangulate nodes)
    ;; NODES should be a vector containing nodal data structures.
    (set! *delaunay-edges* (make-dynamic-table))
    (delaunay (sort (vector->list nodes) lexicographic<)))
(define (lexicographic< n1 n2)
    (let ((delta-x (- (node:get-x n2) (node:get-x n1))))
        (if (almost-zero? delta-x)
            (< (node:get-y n1) (node:get-y n2))
            (> delta-x 0))))
(define (delaunay S)
    ;; This returns the counterclockwise convex hull edge out of
    ;; the leftmost vertex and the clockwise convex hull edge out
    ;; of the rightmost vertex.
    ;;
    ;; S is assumed to be a list of nodes, sorted along the abscissa.
    (cond
        ((< (length S) 2)
                (error "Need at least two nodes to triangulate -- DELAUNAY"))
        ((= (length S) 2)
            ;; Let s1, s2 be the two sites, in sorted order, and create an edge from s1
            ;; to s2:
            (let* ((s1 (car S))
                    (s2 (cadr S))
                            (a (make-edge)))
            (set-org! a s1)
```

```
    (set-dest! a s2)
    (list a (sym a))))
((= (length S) 3)
    ;; Let s1, s2, and s3 be the three sites, in sorted order.
    ;; Create edges a connecting s1 to s2 and b connecting s2 to s3:
    (let* ((s1 (car S))
            (s2 (cadr S))
            (s3 (caddr S))
            (a (make-edge))
            (b (make-edge)))
    (splice (sym a) b)
    (set-org! a s1)
    (set-dest! a s2)
    (set-org! b s2)
    (set-dest! b s3)
    ;; Now close the triangle:
    (cond
        ((ccw s1 s2 s3)
            (connect b a)
            (list a (sym b)))
            ((ccu s1 s3 s2)
            (let ((c (connect b a)))
                (list (sym c) c)))
            (else ; the three points are collinear
            (list a (sym b))))))
(else
    ;; |S| > 3. Let L and R be the left and right halves of S.
    (let* ((L&R (halve S))
                (L (car L&R))
                (R (cadr L&R))
                (ldo&ldi (delaunay L))
                (ldo (car ldo&ldi))
                (ldi (cadr ldo&ldi))
                (rdi&rdo (delaunay R))
                (rdi (car rdi&rdo))
                (rdo (cadr rdi&rdo)))
    ;; Compute the lower common tangent of L and R:
    (call-with-current-continuation
        (lambda (exit)
                (do () (#f)
                        (cond ((left-of (org rdi) ldi)
                            (set! ldi (lnext ldi)))
                            ((right-of (org ldi) rdi)
                            (set! rdi (rprev rdi)))
                            (else (exit 'done))))))
    ;; Create a first cross edge basel from rdi.Org to ldi.Org:
    (let* ((basel (connect (sym rdi) ldi))
```

```
        (valid (lambda (e) (right-of (dest e) basel))))
(if (node= (org ldi) (org ldo)) (set! ldo (sym basel)))
(if (node= (org rdi) (org rdo)) (set! rdo basel))
;; This is the merge loop:
;; Locate the first L point (lcand.Dest) to be encountered by the
;; rising bubble, and delete L edges out of basel.Dest that fail the
;; circle test.
(call-with-current-continuation
    (lambda (exit)
        (do () (#f)
            (let ((lcand (onext (sym basel))))
                (if (valid lcand)
                        (do ()
                            ((not (in-circle (dest basel) (org basel) (dest lcand)
                                    (dest (onext lcand)))))
                            (let ((t (onext lcand)))
                                    (delete-edge lcand)
                                    (set! lcand t))))
                    ;; Symmetrically, locate the first R point to be hit, and delete
                    ;; R edges:
                        (let ((rcand (oprev basel)))
                        (if (valid rcand)
                        (do ()
                            ((not (in-circle (dest basel) (org basel) (dest rcand)
                                    (dest (oprev rcand)))))
                                    (let ((t (oprev rcand)))
                                    (delete-edge rcand)
                            (set! rcand t))))
                    ; If both lcand and rcand are invalid, then basel is the
                    ;; upper common tangent:
                        (if (and (not (valid lcand)) (not (valid rcand)))
                        (exit 'done))
                    ;; The next cross edge is to be connected to either lcand.Dest
                    ; ; or rcand.Dest. If both are valid, then choose the
                    ;; appropriate one using the InCircle test:
                        (if (or (not (valid lcand))
                        (and (valid rcand)
                                    (in-circle (dest lcand) (org lcand) (org rcand)
                                    (dest rcand))))
                                    ;; Add cross edge basel from rcand.Dest to basel.Dest:
                                    (set! basel (connect rcand (sym basel)))
                                    ; Else add cross edge basel from basel.Org to lcand.Dest:
                (set! basel (connect (sym basel) (sym lcand))))))))))
```

(list ldo rdo)))))

```
(define (halve 1)
    ;; Split L down the middle
    (let loop ((hcount 0) (tcount (length 1)) (head '()) (tail 1))
        (if (<= tcount hcount)
            (list (reverse head) tail)
            (loop (+ hcount 1) (- tcount 1) (cons (car tail) head) (cdr tail)))))
(define (node= n1 n2)
    (apply equal? (map node:get-coords '(,n1 ,n2))))
;;; Use the above algorithm to compute the convex hull:
(define (convex-hull nodes)
    (let* ((e (cadr (triangulate nodes)))
            (e.org (org e)))
        (let loop ((l (list e)))
            (let ((next (lnext (car l))))
                (if (node= (org next) e.org)
                    l
                    (loop (cons next 1))))))
```


## C.2.13 delaux.scm

```
;; This implements the topological and geometric primitives needed for the
;;; Delaunay triangulation algorithm described in:
;;;
;;; "Primitives for the manipulation of general subdivisions and the
;;; computation of Voronoi diagrams," ACM transactions on graphics, Vol. 4,
;;; No. 2, April 1985, P. 74-123.
;;; Leonidas Guibas and Jorge Stolfi, Xerox PARC and Stanford University.
;;;
;;; This needs the file edge.scm for the edge-algebraic definitions.
;;; The nodal structures are assumed to provide the methods NODE:GET-X and
;;; NODE:GET-Y.
(declare (usual-integrations))
;;; Edge-record methods:
(define (org e-ref)
    (((get-edge-record e-ref) (get-rot-deg e-ref)) 'data))
(define (left e-ref)
    (org (inv-rot e-ref)))
(define (right e-ref)
    (org (rot e-ref)))
(define (dest e-ref)
    (org (sym e-ref)))
(define (set-org! e-ref new)
    ((((get-edge-record e-ref) (get-rot-deg e-ref)) 'set-data!) new))
(define (set-dest! e-ref new)
```

```
    (set-org! (sym e-ref) new))
;;; Topological operators:
(define (connect a b)
    ;; Create an edge E that connects A.Dest to B.Org, such that A.Left = E.Left
    ;; = B.Left after the connection is complete.
    (let ((e (make-edge)))
            (set-org! e (dest a))
            (set-dest! e (org b))
            (splice e (lnext a))
            (splice (sym e) b)
            e))
(define (delete-edge e)
    (splice e (oprev e))
    (splice (sym e) (oprev (sym e)))
    (dynamic-table-set! *delaunay-edges* (get-edge-id e) 'deleted))
(define (swap e)
    (let ((a (oprev e))
            (b (oprev (sym e))))
        (splice e a)
        (splice (sym e) b)
        (splice e (lnext a))
        (splice (sym e) (lnext b))
        (set-org! e (dest a))
        (set-dest! e (dest b))))
;;; Geometric primitives:
(define (in-circle a b c d)
    ;; a, b, c, and d should be 2-vectors.
    (let ((m (make-matrix 4 4)))
        (do ((l (map node:get-coords (list a b c d)) (cdr l))
            (i 0 (+ i 1)))
            ((null? l))
            (let* ((p (car l))
                            (x (vector-ref p 0))
                            (y (vector-ref p 1)))
            (matrix-set! m i O x)
            (matrix-set! m i 1 y)
            (matrix-set! m i 2 (+ (square x) (square y)))
            (matrix-set! m i 3 1)))
        (> (det m) 0)))
(define (ccw a b c)
    ;; a, b, and c should be 2-vectors.
    (let ((m (make-matrix 3 3)))
        (do ((l (map node:get-coords (list a b c)) (cdr l))
            (i 0 (+ i 1)))
            ((null? l))
            (let* ((p (car l))
                (x (vector-ref p 0))
```

```
                    (y (vector-ref p 1)))
            (matrix-set! m i O x)
            (matrix-set! m i 1 y)
            (matrix-set! m i 2 1)))
            (> (det m) 0)))
(define (right-of x e)
    (ccw x (dest e) (org e)))
(define (left-of x e)
    (ccw x (org e) (dest e)))
;;; A procedure that lists all of the mesh elements in a Delaunay triangulation
;;; can be very useful, particularly for our FEM applications.
(define (get-edge-mark e-ref)
    (((get-edge-record e-ref) (get-rot-deg e-ref)) 'mark))
(define (set-edge-mark! e-ref val)
    ((((get-edge-record e-ref) (get-rot-deg e-ref)) 'set-mark!) val))
(define (list-faces)
    (let ((edges (list-edges)))
        (if (< (length edges) 3)
            ,()
                (let ((faces '()))
                    ;; Reset markings:
                    (for-each
                        (lambda (e)
                            (set-edge-mark! e #f)
                (set-edge-mark! (sym e) #f))
                    edges)
                    ;; Begin DFS:
                    (let loop ((e (car edges)))
                    (for-each
                        (lambda (a)
                        (if (false? (get-edge-mark a))
                            (let* ((b (lnext a))
                                    (c (lnext b)))
                                    (set-edge-mark! a #t)
                                    (if (node= (dest c) (org a))
                                    (begin
                                    (set-edge-mark! b #t)
                                    (set-edge-mark! c #t)
                                    (set! faces (cons (list a b c) faces))))
                                    (loop (sym a)))))
                (get-edge-ring e)))
                    faces)))(
```


## C.2.14 dyntable.scm

```
(declare (usual-integrations))
```

```
(define (make-dynamic-table . argl)
    (if (or (null? argl)
                (> (length argl) 0))
            (vector 0 (make-vector 8))
            (vector 0 (make-vector (car argl)))))
(define (dynamic-table-size table)
    (vector-ref table 0))
(define (dynamic-table-add table new-element)
    (let ((size (dynamic-table-size table))
            (real-size (vector-length (vector-ref table 1))))
        (if (= size real-size)
            (let ((old-table (vector-ref table 1))
                    (new-table (make-vector (* 2 real-size))))
                        (do ((i 0 (+ i 1)))
                            ((>= i real-size))
                            (vector-set! new-table i (vector-ref old-table i)))
                (vector-set! table 1 new-table)))
        (vector-set! (vector-ref table 1) size new-element)
        (vector-set! table O (+ size 1))))
(define (dynamic-table-fetch table i)
    (if (>= i (dynamic-table-size table))
            (error "Access out of bound -- DYNAMIC-TABLE-FETCH")
            (vector-ref (vector-ref table 1) i)))
(define (dynamic-table->list table)
    (let loop ((l '()) (n (- (dynamic-table-size table) 1)))
        (if (< n 0)
            l
            (loop (cons (dynamic-table-fetch table n) l) (- n 1)))))
(define (dtableq table element)
    (let loop ((n (- (dynamic-table-size table) 1)))
        (if (< n 0)
            #f
            (if (eq? (dynamic-table-fetch table n) element)
                    n
                    (loop (- n 1))))))
(define (dynamic-table-set! table i val)
    (if (>= i (dynamic-table-size table))
            (error "Access out of bound -- DYNAMIC-TABLE-SET!")
            (vector-set! (vector-ref table 1) i val)))
```


## C.2.15 edge.scm

;; This is a simple implementation of the quad-edge data structure and the
;; associated edge functions on the edge algebra, as described in:
; ; ;
;; ; "Primitives for the manipulation of general subdivisions and the
;;; computation of Voronoi diagrams," ACM transactions on graphics, Vol. 4,
;;; No. 2, April 1985, P. 74-123.
; ; ; Leonidas Guibas and Jorge Stolfi, Xerox PARC and Stanford University.

```
;;;
;;; This is, in particular, meant to be used by the Delaunay triangulation
;;; algorithm.
(declare (usual-integrations))
;;; A global list of edges:
(define *delaunay-edges* (make-dynamic-table))
;;; The quad-edge data structure: This is the basic data structure
;;; representing the edges in an edge algebra. It consists of a representative
;;; edge and its orbit under the operations Rot and Flip.
(define (make-edge-record id)
    (let ((orientation 0)
                (next (make-vector 4))
                ;; Auxiliary data fields, initialized to the null lists:
                (data (make-vector 4 'undefined))
                (mark (make-vector 4 'undefined)))
        ;; Main dispatcher:
        (define (me r)
            (lambda (msg)
                (case msg
                        ((next) (vector-ref next r))
                    ((set-next!) (lambda (val) (vector-set! next r val)))
                    ((data) (vector-ref data r))
                    ((set-data!) (lambda (val) (vector-set! data r val)))
                    ((mark) (vector-ref mark r))
                    ((set-mark!) (lambda (val) (vector-set! mark r val)))
                    (else (error "Unknown request -- EDGE-RECORD")))))
        ;; Initialize the edges, which lie on a 2-sphere.
        (vector-set! next 0 (make-edge-ref me 0 orientation id))
        (vector-set! next 1 (make-edge-ref me 3 orientation id))
        (vector-set! next 2 (make-edge-ref me 2 orientation id))
        (vector-set! next 3 (make-edge-ref me 1 orientation id))
        me))
```

; ; ; An edge-reference is a triplet (e, $r, f$ ), where $e$ is an edge record and $r$
; ; ; and $f$ are the corresponding Rot and Flip degrees.
(define (make-edge-ref erfid)
(vector erfid))
(define (get-edge-record e-ref)
(vector-ref e-ref 0))
(define (get-rot-deg e-ref)
(vector-ref e-ref 1 ))
(define (get-flip-deg e-ref)

```
    (vector-ref e-ref 2))
(define (get-edge-id e-ref)
    (vector-ref e-ref 3))
(define (get-edge-ring e)
    (let ((e.dest (dest e)))
        (let loop ((l (list e)) (e e))
            (let ((e.onext (onext e)))
                (if (node= e.dest (dest e.onext))
                    l
                            (loop (cons e.onext l) e.onext))))))
;;; Basic edge functions on which others are built:
(define (rot e-ref)
    (let ((e (get-edge-record e-ref))
            (r (get-rot-deg e-ref))
            (f (get-flip-deg e-ref))
            (id (get-edge-id e-ref)))
        (make-edge-ref e (modulo (+ r 1 (* 2 f)) 4) f id)))
(define (flip e-ref)
    (let ((e (get-edge-record e-ref))
                (r (get-rot-deg e-ref))
                (f (get-flip-deg e-ref))
                (id (get-edge-id e-ref)))
        (make-edge-ref e r (modulo (+ f 1) 2) id)))
(define (onext e-ref)
    (let ((e (get-edge-record e-ref))
                (r (get-rot-deg e-ref))
            (f (get-flip-deg e-ref)))
        (if (zero? f)
            ((e (modulo (+ r f) 4)) 'next)
            (flip (rot ((e (modulo (+ r f) 4)) 'next))))))
;;; Other edge functions:
(define inv-flip flip)
(define (sym e-ref)
    (rot (rot e-ref)))
(define (inv-rot e-ref)
    (rot (rot (rot e-ref))))
(define (dual e-ref)
    (rot (flip e-ref)))
(define (lnext e-ref)
    (rot (onext (inv-rot e-ref))))
(define (rnext e-ref)
    (inv-rot (onext (rot e-ref))))
(define (dnext e-ref)
```

```
    (sym (onext (sym e-ref))))
(define (oprev e-ref)
    (rot (onext (rot e-ref))))
(define (lprev e-ref)
    (sym (onext e-ref)))
(define (rprev e-ref)
    (onext (sym e-ref)))
(define (dprev e-ref)
    (inv-rot (onext (inv-rot e-ref))))
;;; Vertices are represented by an out-going edge, and faces are represented by
;;; an out-going edge in the dual diagram:
(define (org-ring e-ref)
    e-ref)
(define (left-ring e-ref)
    (org-ring (inv-rot e-ref)))
(define (right-ring e-ref)
    (org-ring (rot e-ref)))
(define (dest-ring e-ref)
    (org-ring (sym e-ref)))
;;; Basic topological operators:
(define (make-edge)
    (let* ((id (dynamic-table-size *delaunay-edges*))
                            (e (make-edge-ref (make-edge-record id) 0 0 id)))
        (dynamic-table-add *delaunay-edges* e)
        e))
(define (splice a b)
    (if (not (equal? b (flip (onext a))))
            (let* ((alpha (rot (onext a)))
                        (beta (rot (onext b)))
                        (a-onext (onext a))
                        (b-onext (onext b))
                        (alpha-onext (onext alpha))
                                (beta-onext (onext beta)))
            (set-onext! a b-onext beta)
            (set-onext! b a-onext alpha)
            (set-onext! alpha beta-onext b)
            (set-onext! beta alpha-onext a))))
(define (set-onext! e-ref new alt)
    (let ((f (get-flip-deg e-ref)))
        (if (zero? f)
            (let ((e (get-edge-record e-ref))
                        (r (get-rot-deg e-ref)))
```

```
    (((e (modulo (+ r f) 4)) 'set-next!) new))
(let* ((e-ref (rot (flip e-ref)))
                    (e (get-edge-record e-ref))
                    (r (get-rot-deg e-ref))
(f (get-flip-deg e-ref)))
(((e (modulo (+ r f) 4)) 'set-next!) (flip alt)))))
```

```
;;; List all edges:
(define (list-edges)
    (if (list? *delaunay-edges*)
        *delaunay-edges*
        (let ((size (dynamic-table-size *delaunay-edges*)))
            (let loop ((l '()) (i 0))
            (if (< i size)
                (let ((e (dynamic-table-fetch *delaunay-edges* i)))
                    (if (eq? e 'deleted)
                        (loop l (+ i 1))
                        (loop (cons e 1) (+ i 1))))
                (begin
                (set! *delaunay-edges* 1)
                    1)))))
```


## C.2.16 fem.scm

; ; ; This program solves linear partial differential equations (with variable ;; ; coefficients) using the finite element method (FEM). It takes as input the ; ; ; source function (the RHS of the equation) and a list of nodes to be ; ; ; used. The boundary nodes should be initialized to the desired values.
;;; Internal dependencies among data structures:
; ; ; * In the final output matrix, each row corresponds to a node. Hence, we
;; ; use nodes to organize the computation of coefficients in each row, and
; ; the actual integrals are computed by the elements. As a result, the
; ; procedures ASSEMBLY and MAKE-NODE are fairly general, and one should
;; seldom need to change them.
;; ; * The differential operator is encapsulated in the constructor for
;; elements, since the operator only affects the computation of the
;; integrals. Many properties of the system (such as the differential
;; operator, element shapes/sizes, dimension of the domain) are parametrized
;;; through this.
;;; Mathematical assumptions:
;;; * Linearity of the differential operator appears to be necessary for these
; ; methods. Otherwise nodal assembly wouldn't work, and we wouldn't be able ; ; to construct linear equations out of the matrix of inner products.
;;; * We cannot handle systems of equations yet.
; ; ; * MAKE-NODE and ELEMENT-MAKER appear to be the only procedures that
; ; restrict the dimensions to which this program applies. The former does
; ; so by requiring $X$ and $Y$ as arguments, while the latter needs to handle

```
;;; higher-dimensional (dim > 1) faces of simplices.
;;; * None of this code actually does anything, of course, since it's
;;; completely abstract. To generalize this to higher dimensions, the
;;; appropriate integrators and constructors would still need to be supplied,
;;; which can be a non-trivial task. How much of what's in 2d-domains.scm
;;; generalized to higher dimensions (assuming regions simple enough to be
;;; triangulated efficiently)?
;;; Our basic reference is:
;;; Vichnevetsky, Robert. _Computer Methods for Partial Differential Equations,
;;; Volume 1: Elliptic Equatins and the Finite-Element Method_. Prentice-Hall:
;;; Englewood Cliffs, New Jersey, 1981.
;;; Note that elements are implicitly accessed through nodes, so that we should
;;; never have to directly refer to elements. Also, the program only computes
;;; the system of linear algebraic equations; it does not attempt to solve
;;; them.
;;; A possible future direction is to extend this program to systems of
;;; equations. How does one handle non-linear equations in general?
(declare (usual-integrations))
;;; Use FEM to produce a matrix:
(define (fem source nodes potential)
    (initialize-values nodes potential)
    (assemble-equations source nodes))
;;; Set boundary values:
(define (initialize-values nodes f)
    (let ((size (vector-length nodes)))
        (do ((i O (+ i 1)))
            ((>= i size))
            (let ((node (vector-ref nodes i)))
                (node:set-value! node (f node))))))
;;; The Rayleigh-Ritz method, as described in Vichnevetsky. Actually, since
;;; MAKE-ELEMENT already incurs most of the cost of discretization up front, we
;;; only need to assemble the equations.
;;; NODE:ASSEMBLE returns a SPARSE-MATRIX data structure, as described in
;;; sparse.scm. It can be used directly as input to SOR, or converted into a
;;; matrix and solved by LU decomposition.
(define (assemble-equations source nodes)
    ;; SOURCE is a function from R^2 to R, and NODES is expected to be a vector.
    (let* ((ncount (vector-length nodes))
            (bcount 0)
            (index-map (make-vector ncount)))
        ;; First, assign each node an index and count the number of boundary nodes.
```

```
(do ((i 0 (+ i 1)))
        ((>= i ncount))
    (node:set-id! (vector-ref nodes i) i)
    (if (node:boundary? (vector-ref nodes i))
            (set! bcount (+ bcount 1))))
;; Next, create a mapping from node indices into matrix row number. (The
;; matrix has one row per interior node.)
;; This enforces the constraint that the equations satisfy the boundary
;; conditions. Note that we can enforce the constraint before *or* after
;; minimizing the action. If we do it before, everything is fine. If we
;; do it in the other order, then we have to justify dropping the
;; constraint equations. (Why does this sound vaguely familiar? Does it
;; have anything at all to do with nonholonomic constraints?)
;; In any case, we can drop the equations associated with boundary nodes by
;; enforcing the constraint *before* we differentiate. Perhaps this cannot
;; be done with the wave equation? Would the Neumann condition along the
;; initial line look like some kind of nonholonomic constraint when we
;; perform the constrained minimization of the action? But ODEs do not
;; have this problem...
```

```
(let loop ((i 0) (row 0))
    (if (< i ncount)
        (if (node:boundary? (vector-ref nodes i))
                        (begin
                        (vector-set! index-map i #f)
                (loop (+ i 1) row))
                (begin
                (vector-set! index-map i row)
                (loop (+ i 1) (+ row 1))))))
;; Loop over the nodes to create row entries:
(let* ((icount (- ncount bcount))
            (big-matrix (make-sparse-matrix icount (1+ icount))))
        (do ((i 0 (+ i 1)))
            ((>= i ncount))
        (if (not (node:boundary? (vector-ref nodes i)))
            (let ((row (vector-ref index-map i)))
            ;; Compute the source term for this row:
                (sparse-matrix-set! big-matrix row icount
                    (node:compute-source (vector-ref nodes i)
                                    source))
```

                ; Combine boundary values:
                (for-each
                    (lambda (pair)
                    (let ((id (car pair))
                            (val (cadr pair)))
                            (if (node:boundary? (vector-ref nodes id))
                        (sparse-matrix-set!
    ```
    big-matrix rov icount
    (- (sparse-matrix-ref big-matrix row icount)
    (* val (node:get-value (vector-ref nodes id)))))
        (sparse-matrix-set! big-matrix row
                            (vector-ref index-map id) val))))
(node:assemble (vector-ref nodes i))))))
```

big-matrix)))

```
;;; These procedures localize some of the assembly process in nodes; they are
;;; defined separately from nodes themselves to isolate the definitions
;;; specific to this particular application, thus increasing the generality of
;;; the definitions in this file.
(define (node:assemble node)
    (let ((l (append-map
                        (lambda (element index)
                            (element:compute-integrals element index))
            (node:get-elements node)
            (node:get-local-ids node))))
        (merge-terms 1 + (lambda (x y) (< (car x) (car y))))))
(define (node:compute-source node source)
    (apply + (map (lambda (element index)
                    (element:compute-source element source index))
                            (node:get-elements node)
                (node:get-local-ids node))))
;;; This is a useful helper procedure: Given a list L of the form L = ((index1
;;; val1) (index2 val2) ...) and a procedure COMBINE, use COMBINE to
;;; concatenate the values of elements of L with the same index.
(define (merge-terms l combine <)
    ; Sort first, then accumulate. This is O(n log n), which is after than the
    ;; obvious O(n^2) algorithm.
    (if (null? 1)
        '()
        (let* ((l (sort 1 <))
            (indices (map car 1))
            (values (map cadr 1)))
            (let loop ((indices (cdr indices))
                    (vals (cdr values))
                    (result '())
                    (id (car indices))
                    (accum (car values)))
                (if (null? indices)
                        (cons (list id accum) result)
                        (if (eq? (car indices) id)
                    (loop (cdr indices) (cdr vals) result id
                    (combine accum (car vals)))
                    (loop (cdr indices) (cdr vals)
                    (cons (list id accum) result)
                    (car indices) (car vals)))))))
```

```
;;; Let's now define elements. To be completely general (in terms of
;;; dimensions of applicability), we should allow the construction of nodes on
;;; higher-dimensional (> 1) faces.
;;; Note that this implicitly assumes that elements are the convex hull of
;;; their vertices.
;;; The constructor for element-constructors:
(define (element-maker make-operator
make-integrator
make-basis-function)
;; MAKE-INTEGRATOR should take as argument a list of nodes, and returns a
; ; procedure that takes a variable number of functions (at least 1) and
; integrates their product over the domain specified implicitly as the
; convex hull of the vertex nodes.
;; MAKE-BASIS-FUNCTION should take as argument a list of nodes and the index
; ; of the node that is to be the center of the basis function, and return
;; some structure representing basis functions.
;; We place no restrictions on the representation of functions over elements,
; ; so long as the particular instances of MAKE-BASIS-FUNCTION and
;; MAKE-INTEGRATOR agree a-priori on the representation.
MAKE-OPERATOR should take a list of nodes and return LEFT-OP, RIGHT-OP,
and COMBINE procedure, satisfying (INTEGRATE (COMBINE (LEFT-OP F)
(RIGHT-OP G))) = (INTEGRATE F (OP G)), i.e. implement integration by parts
so that basis functions can be less smooth.
The list of nodes facilitates the interpolation of variable coefficients
in the operator. This may not be a good interface, as it makes artificial
assumptions on the contract between basis functions and operators (as is
the explicit use of LEFT-OP and RIGHT-OP).
(define (make-element vertex-nodes other-nodes)
;; The first part stores the coefficients, the second part the source
; terms. What about coefficients? Maybe we should incorporate the
;; source term into the differential operator.
(let* ((nodes (append vertex-nodes other-nodes))
(number-of-nodes (length nodes))
(n-choose-2 (choose (+ number-of-nodes 2) 2))
(element
(vector (make-vector n-choose-2 0)
                                    (make-vector n-choose-2 0)
                                    vertex-nodes
                                    other-nodes
                                    (make-vector number-of-nodes #f)))
(op (make-operator nodes)))
;; Add the element to the nodes:
    (let loop ((nodes nodes) (i 0))
        (if (not (null? nodes))
            (begin
```

```
                (node:add-element (car nodes) element i)
                (loop (cdr nodes) (+ i 1)))))
        ;; Initiailize elements (and hiding the hair)...
        (let ((integrate (make-integrator vertex-nodes))
            (local-form (operator:get-local-form op)))
            (do ((i O (+ i 1)))
                ((>= i number-of-nodes))
            (element:set-basis-function!
            element i (make-basis-function nodes i)))
            (do ((i O (+ i 1)))
                ((>= i number-of-nodes))
            (let ((f (element:get-basis-function element i)))
                (do ((j i (+ j 1)))
                    ((>= j number-of-nodes))
                    (let ((g (element:get-basis-function element j)))
                    (element:set-coeff! element i j
                            (integrate (local-form f g)))
                            (element:set-source! element i j (integrate f g))))))
        element))
    make-element)
;;; Methods for accessing the data structure:
(define (element:get-coeff element i j)
    (vector-ref (vector-ref element 0) (symmetric->vector-index i j)))
(define (element:set-coeff! element i j val)
    (vector-set! (vector-ref element 0) (symmetric->vector-index i j) val))
(define (element:get-source element i j)
    (vector-ref (vector-ref element 1) (symmetric->vector-index i j)))
(define (element:set-source! element i j val)
    (vector-set! (vector-ref element 1) (symmetric->vector-index i j) val))
(define (element:get-vertex-nodes element)
    (vector-ref element 2))
(define (element:get-non-vertex-nodes element)
    (vector-ref element 3))
(define (element:get-nodes element)
    (append (element:get-vertex-nodes element)
                (element:get-non-vertex-nodes element)))
(define (element:node-count element)
    (length (element:get-nodes element)))
(define (element:set-basis-function! element i basis-function)
    (vector-set! (vector-ref element 4) i basis-function))
```

```
(define (element:get-basis-function element i)
    (vector-ref (vector-ref element 4) i))
;;; Computing the data needed in assembly:
(define (element:compute-source element source i)
    (let loop ((nodes (element:get-nodes element)) (sum 0.) (j 0))
        (if (null? nodes)
            sum
                        (loop (cdr nodes)
                            (* (source (car nodes)) (element:get-source element i j))
                            (+ j 1)))))
(define (element:compute-integrals element i)
    (let loop ((nodes (element:get-nodes element)) (l '()) (j 0))
        (if (null? nodes)
            l
            (loop (cdr nodes)
                        (cons (list (node:get-id (car nodes))
                                    (element:get-coeff element i j))
                                    1)
                            (+ j 1)))))
;;; It's useful to have constant source functions:
(define (make-constant-function const)
    (lambda (node) const))
(define 0-function (make-constant-function 0))
```


## C.2.17 job.scm

```
(load "load")
(define nodes (make-real-square-domain 3 3))
(define m (sparse->matrix (fem 0-function nodes node:get-x)))
(define v (lu-solve m))
(let ((port (open-output-file "test"))
            (n (vector-length v)))
    (print-matrix m port)
    (newline port)
    (newline port)
    (do ((i 0 (+ i 1)))
                ((>= i n))
        (write-line (vector-ref v i) port))
    (close-output-port port))
```


## C.2.18 job1.scm

(load "collect")
(test-3 '(20 202.1 ) 100001.9 "bigtest.data")

## C.2.19 load.scm

| (load "util") | ; $;$ Useful helper routines. |
| :---: | :---: |
| (load "util-too") | ; Useful routines compatible with ScmUtils. |
| (load "matlib") | ; : Everyone needs the matrix library. |
| (load "sparse") | ; S Sparse matrices. |
| (load "relax") | ; R Relaxation. |
| (load "dyntable") | ; D Dynamic tables. |
| (load "debug") | ; Graphics \& stuff. |
| (load "delaunay") | ; D Delaunay triangulation. |
| (load "delaux") | ; Auxiliary routines for Delaunay. |
| (load "edge") | ; E Edge algebra junk for Delaunay. |
| (load "fem") | ; The main finite-element code. |
| (load "nodes") | ; $;$ Definition of nodes. |
| (load "basis") | ; ; Basis functions. |
| (load "2d-poly-basis") | ; P Polynomial basis functions in two variables. |
| (load "2d-real-diff") | ; $;$ Real differential operators on real functions. |
| (load "2d-real-basis") | ; P Polynomial basis functions in two variables. |
| (load "2d-trapezoid") | ; Numerical integration using trapezoidal rule. |
| (load "operators") | ; Tools for differential operators. |
| (load "2d-operators") | ; Examples. |
| (load "2d-domains") | ; M Making domains and boundaries, etc. |
| (load "2d-examples") | ; ; Some examples. |

## C.2.20 matlib.scm

;; S Some useful ideas from concrete linear algebra. It is pretty poorly
;; ; organized and some implementations need improving.
(declare (usual-integrations))
;;; Vector operations (more to come as needed):
(define (inner-product v1 v2)
(if (and (vector? v1)
(vector? v2)
(= (vector-length v1) (vector-length v2)))
(let ((n (vector-length v1)))
(let loop ((sum 0) (i 0 )) (if (< in)
(loop (+ sum (* (vector-ref v1 i) (vector-ref v2 i))) (+ i 1))
sum)))
\#f))
(define (apply-affine-transformation A b v)
; ; Left-multiplication by a matrix:
(let* ( $n$ (vector-length v))
( $\mathbf{w}$ (make-vector n )))
(do ( i 0 (+ i 1)))
( $(>=\mathrm{i} n$ ) w)
(let loop ( (j 0) (sum 0.))
(if (< j n)
(loop (+ j 1) (+ sum (* (matrix-ref A i j) (vector-ref $\mathbf{v} j$ )))) (vector-set! wi (+ sum (vector-ref bi)))))))

```
(define (apply-linear-transformation A v)
    (let ((m (matrix-row-count A))
            (n (matrix-column-count A))
            (p (vector-length v)))
        (if (not (= n p))
            (error "Wah! A mistake! -- APPLY-LINEAR-TRANSFORMATION"))
        (let ((w (make-vector m)))
            (do ((i 0 (+ i 1)))
                ((>= i m) w)
                    (let loop ((j 0) (sum 0))
                        (if (< j n)
                            (loop (+ j 1) (+ sum (* (matrix-ref A i j) (vector-ref v j))))
                            (vector-set! w i sum)))))))
```

;;; Miscellaneous matrix operations:
(define (transpose A)
(let* ((m (matrix-row-count A))
(n (matrix-column-count A))
(At (make-matrix $n \mathrm{~m}$ )) )
(do ((i) 0 (+i1)))
( $(>=\mathrm{i} m)$ At)
(do ((j 0 (+ j 1)))
( $(>=j n)$ )
(matrix-set! At $j$ i (matrix-ref A $i \quad j))$ )))
(define (matrix:binary* A B)
(let ( $(\mathrm{p}$ (matrix-column-count A)))
(if (not (= p (matrix-row-count B)))
(error "Incompatible matrix sizes!"))
(let* ((m (matrix-row-count A))
( $n$ (matrix-column-count B))
(result (make-matrix m n)))
(do ( $\mathrm{i} 0(+\mathrm{i} 1))$ )
( $(>=\mathrm{i} \mathrm{m})$ )
(do ((j 0 (+ j 1)))
( $(>=\mathrm{j} n)$ )
(let loop ( $(\mathrm{k} 0)$ (sum 0))
(if (< k p)
(loop (+k 1)
(+ sum (* (matrix-ref A i k) (matrix-ref B k j))))
(matrix-set! result i j sum)))))
result)))
(define (matrix:* A . rest)
(let loop ((A A) (l rest))
(if (null? 1)
A
(loop (matrix:binary* A (car l)) (cdr 1)))))
(define (matrix:+ A . rest)
(let loop ((A A) (l rest))
(if (null? 1)

```
        A
            (loop (matrix:binary+ A (car 1)) (cdr 1)))))
(define (matrix:binary+ A B)
    (let* ((m (matrix-row-count A))
                    (n (matrix-column-count A))
                    (C (make-matrix m n)))
        (if (not (and (= m (matrix-rou-count B))
                        (= n (matrix-column-count B))))
            (error "Cannot add matrices of different dimensions!"))
        (do ((i O (+ i 1)))
            ((>= i m) C)
        (do ((j O (+ j 1)))
            ((>= j n))
            (matrix-set! C i j (+ (matrix-ref A i j) (matrix-ref B i j)))))))
(define (matrix:- A . rest)
    (if (null? rest)
        A
        (matrix:binary- A (apply matrix:+ rest))))
(define (matrix:binary- A B)
    (let* ((m (matrix-rou-count A))
            (n (matrix-column-count A))
            (C (make-matrix m n)))
        (if (not (and (= m (matrix-row-count B))
                        (= n (matrix-column-count B))))
            (error "Cannot subtract matrices of different dimensions!"))
        (do ((i O (+ i 1)))
                ((>= i m) C)
            (do ((j 0 (+ j 1)))
                    ((>= j n))
                (matrix-set! C i j (- (matrix-ref A i j) (matrix-ref B i j)))))))
(define (matrix:trace A)
    (let ((m (matrix-row-count A)))
        (if (not (= m (matrix-column-count A)))
            (error "Cannot compute the trace of a non-square matrix!"))
        (let loop ((i 0) (sum 0))
            (if (< i m)
                    (loop (+ i 1) (+ sum (matrix-ref A i i)))
                        sum))))
```

;; Matrix constructors and methods:
(define (make-matrix m n)
(let ((vector-of-rows (make-vector m)))
(do ( $(\mathrm{i} 0$ (+ i 1)))
( $>=\mathrm{im}$ ) )
(vector-set! vector-of-rows i (make-vector n 0 )))
vector-of-rows))

```
(define (list->matrix m n l)
    (if (not (= (length l) (* m n)))
        (error "Incorrect dimensions -- LIST->MATRIX")
        (let ((A (make-matrix m n)))
            (do ((i 0 (+ i 1)))
                    ((>= i m) A)
                    (do ((j 0 (+ j 1)))
                        ((>= j n))
                        (matrix-set! A i j (car l))
                        (set! l (cdr l)))))))
(define (matrix-set! matrix i j newval)
    (vector-set! (vector-ref matrix i) j newval))
(define (matrix-ref matrix i j)
    (vector-ref (vector-ref matrix i) j))
(define matrix-row-count vector-length)
(define (matrix-column-count matrix)
    (vector-length (vector-ref matrix 0)))
(define (matrix-size matrix)
    (list (matrix-row-count matrix) (matrix-column-count matrix)))
(define matrix-dimensions matrix-size)
(define (matrix-copy A)
    (let* ((m (matrix-row-count A))
                            (n (matrix-column-count A))
                            (B (make-matrix m n)))
        (do ((i O (+ i 1)))
            ((>= i m) B)
            (do ((j 0 (+ j 1)))
                ((>= j n))
                    (matrix-set! B i j (matrix-ref A i j)))))
(define (matrix-get-row A i)
    (let* ((n (matrix-column-count A))
                (result (make-vector n)))
        (do ((j 0 (+ j 1)))
            ((>= j n) result)
            (vector-set! result j (matrix-ref A i j)))))
(define (matrix-get-column A j)
    (let* ((m (matrix-row-count A))
                (result (make-vector m)))
        (do ((i 0 (+ i 1)))
            ((>= i m) result)
            (vector-set! result i (matrix-ref A i j)))))
;;; Some predicates that might be useful:
(define (matrix? matrix)
    (and (vector? matrix)
            (> (vector-length matrix) 0)
            (vector? (vector-ref matrix 0))
            (let ((m (vector-length matrix))
```

```
            (n (vector-length (vector-ref matrix 0))))
        (let loop ((i 0))
        (if (< i m)
            (if (and (vector? (vector-ref matrix i))
                    (= (vector-length (vector-ref matrix i)) n))
                    (loop (+ i 1))
                    #f)
            #t))))
(define (diag-dom? matrix)
    (let ((m (matrix-row-count matrix))
            (n (matrix-column-count matrix)))
        (if (> m n) (error "Matrix has more rows than columns - DIAG-DOM?"))
        (call-with-current-continuation
        (lambda (return)
            (let ((sum 0.))
            (do ((i O (+ i 1)))
                ((>= i m))
                    (set! sum 0.)
                    (do ((j 0 (+ j 1)))
                    ((>= j m))
                            (set! sum (+ sum (abs (matrix-ref matrix i j)))))
                            (let ((diag (abs (matrix-ref matrix i i))))
                            (if (not (or (> diag (- sum diag))
                            (almost-zero? (- (* 2 diag) sum))))
                                    (begin
                                    (write-line '(- ,diag ,(- sum diag)))
                                    (return #f)))))
            #t))))
(define (symmetric? matrix)
    (let ((m (matrix-row-count matrix))
            (n (matrix-column-count matrix)))
        (if (> m n)
            #f
            (call-with-current-continuation
                    (lambda (return)
                    (do ((i 1 (+ i 1)))
                        ((>= i m))
                        (do ((j 0 (+ j 1)))
                            ((>= j i))
                        (if (not (almost-zero? (- (matrix-ref matrix i j)
                            (matrix-ref matrix j i))))
                                (return #f))))
                #t))))
```

;;; LU-decomposition, done in a pretty primitive way. It is almost directly
;; ; lifted out of _Numerical Recipes_. It can also be used to compute
;;; determinants, but it mutates its argument. DET does not.
;;
;; ; Note that the matrix may be left in a partially modified state, since the
;;; procedure aborts on singular matrices.
(define (LU-decomp A . aux)

```
(call-with-current-continuation
    (lambda (exit)
        (let ((m (matrix-row-count A))
            (n (matrix-column-count A))
            (det (if (and (not (null? aux))
                        (eq? (car aux) 'no-det))
                #f
                1)))
            ;; Compute the upper part:
            (do ((j O (+ j 1)))
                    ((>= j m))
            (do ((i 0 (+ i 1)))
                    ((>= i j))
                        (let ((sum (matrix-ref A i j)))
                            (do ((k O (+ k 1)))
                            ((>= ki))
                            (set! sum (- sum (* (matrix-ref A i k) (matrix-ref A k j)))))
                    (matrix-set! A i j sum)))
            ;; Compute the lower portion with partial pivoting:
            (let ((pivot 0) (new-j -1))
                (do ((i j (+ i 1)))
                    ((>= i m))
                        (let ((sum (matrix-ref A i j)))
                        (do ((k 0 (+ k 1)))
                        ((>= k j))
                            (set! sum (- sum (* (matrix-ref A i k) (matrix-ref A k j))))
                        (matrix-set! A i j sum)
                        (if (>= (abs sum) (abs pivot))
                        (begin
                        (set! pivot sum)
                        (set! new-j i)))))
                ;; Swap rows, if necessary.
                (if (> new-j j)
                        (begin
                            (if det (set! det (* det -1)))
                                    (do ((k 0 (+ k 1)))
                                    ((>= k n))
                                    (let ((val (matrix-ref A j k)))
                                    (matrix-set! A j k (matrix-ref A new-j k))
                                    (matrix-set! A new-j k val)))))
                ;; If the matrix is singular, return 0 and leave matrix as is.
                (if (almost-zero? pivot) (exit 0))
                (do ((i (+ j 1) (+ i 1)))
                        ((>= i m))
                    (matrix-set! A i j (/ (matrix-ref A i j) pivot)))))
            ; Compute the determinant, if necessary. Note that we kept track of
```

```
        ;; its sign during row swaps.
        (if det
            (do ((k O (+ k 1)))
                ((>= k m) det)
            ;; Look out for underflows:
            (if (or (almost-zero? det) (almost-zero? (matrix-ref A k k)))
                (exit 0)
                (set! det (* det (matrix-ref A k k)))))
        (begin
            (do ((k O (+ k 1)))
                ((>= k m))
                (if (almost-zero? (matrix-ref A k k))
                    (exit 0)))
            (exit 1))))))
;;; Use LU-decomposition to solve a linear system of equations (signals error
;;; if the system is singular).
(define (LU-solve A . aux)
    (if (sparse-matrix? A)
        (set! A (sparse->matrix A)))
    (let ((m (matrix-row-count A))
            (n (matrix-column-count A)))
        (if (or (null? aux)
            (not (eq? (car aux) 'no-copy)))
            (set! A (matrix-copy A)))
        (rref A)
    ;; Form the result:
        (if (> n (+ m 1))
            (let ((result (make-matrix m (-m n))))
                (do ((i 0 (+ i 1)))
                    ((>= i m) result)
                    (do ((j m (+ j 1)))
                                    ((>= j n))
                            (matrix-set! result i (- j m) (matrix-ref A i j))))
            (let ((result (make-vector m)))
                (do ((i O (+ i 1)))
                    ((>= i m) result)
                    (vector-set! result i (matrix-ref A i m))))))
(define (rref A)
    ;; Solve Ax = b.
    (let ((m (matrix-row-count A))
            (n (matrix-column-count A)))
        (if (>= m n) (error "Incorrect dimensions -- LU-SOLVE"))
            ;; Get A = LU.
```

```
    (let ((det (lu-decomp A 'no-det)))
            (if (almost-zero? det)
                    (error (string-append "Singular matrix! (Determinant = "
                                    (number->string det)
                                    ") -- LU-SOLVE"))))
    ;; Forward substitution to solve Ly = b.
    (do ((j 0 (1+ j)))
        ((>= j m))
        (do ((i (1+ j) (1+ i)))
            ((>= i m))
        (do ((k m (1+ k)))
                ((>= k n))
            (matrix-set! A i k
                                    (- (matrix-ref A i k)
                                    (* (matrix-ref A j k) (matrix-ref A i j))))
            (matrix-set! A i j 0))))
    ;; Backward substitution to solve Ux = y.
    (do ((i (-1+m) (-1+ i)))
        ((< i 0))
        (let ((diag (matrix-ref A i i)))
        (do ((k i (1+ k)))
                ((>= k n))
            (matrix-set! A i k (/ (matrix-ref A i k) diag))))
        (do ((j (-1+ i) (-1+ j)))
            ((< j 0))
        (let ((factor (matrix-ref A j i)))
            (do ((k i (1+ k)))
                ((>= k n))
                (matrix-set! A j k
                            (- (matrix-ref A j k)
                                    (* factor (matrix-ref A i k))))))))))
;;; Compute determinants without mutating the argument:
(define (det A)
    (lu-decomp (matrix-copy A)))
;;; A very useful procedure to have around:
(define (print-matrix matrix . argl)
    (if (null? argl)
        (set! argl (list (current-output-port))))
    (if (not (and (null? (cdr argl)) (output-port? (car argl))))
            (error "Invalid argument(s) -- PRINT-MATRIX"))
    (let ((port (car argl)))
        (newline port)
        (let ((m (matrix-row-count matrix))
                        (n (matrix-column-count matrix)))
            (do ((i 0 (+ i 1)))
```

```
((>= i m))
(display (matrix-ref matrix i 0) port)
(do ((j 1 (+ j 1)))
    ((>= j n))
    (display #\tab port)
    (display (matrix-ref matrix i j) port))
(newline port)))))
```

```
;;; Also very useful, especially for printing large matrices in a Emacs buffer:
```

;;; Also very useful, especially for printing large matrices in a Emacs buffer:
(define (round-matrix mat precision)
(define (round-matrix mat precision)
(let* ((m (matrix-row-count mat))
(let* ((m (matrix-row-count mat))
(n (matrix-column-count mat))
(n (matrix-column-count mat))
(out (make-matrix m n)))
(out (make-matrix m n)))
(do ((i O (+ i 1)))
(do ((i O (+ i 1)))
((>= i m) out)
((>= i m) out)
(do ((j O (+ j 1)))
(do ((j O (+ j 1)))
((>= j n))
((>= j n))
(let ((val (matrix-ref mat i j)))
(let ((val (matrix-ref mat i j)))
(matrix-set! out i j (* (round (/ val precision)) precision))))))

```
                (matrix-set! out i j (* (round (/ val precision)) precision))))))
```


## C.2.21 nodes.scm

```
;; ; This file defines nodes, which localize assembly computations. Nodes use
;;; elements to compute source terms and integrals. Linearity of the PDE is
;;; probabliy the only assumption here. This also appears to be the only
;;; definition that restricts this code to two dimensions.
(declare (usual-integrations))
;;; Constructor:
(define (make-node x y . aux)
    (vector x y (if (null? aux) #f (car aux)) '() '() 37 0.))
;;; Access methods for nodes. These are necessary for the Delaunay
;;; triangulation program, and are also useful for testing the algorithm, as
;;; the canonical coordinate functions are harmonic.
(define (node:get-x node) (vector-ref node 0))
(define (node:get-y node) (vector-ref node 1))
(define (node:get-coords node) (vector (node:get-x node) (node:get-y node)))
(define (node:get-value node) (vector-ref node 6))
(define (node:set-value! node val) (vector-set! node 6 val))
(define (node:get-id node) (vector-ref node 5))
(define (node:set-id! node id) (vector-set! node 5 id))
(define (node:boundary? node) (vector-ref node 2))
(define (node:set-boundary! node flag) (vector-set! node 2 flag))
(define (node:get-elements node) (vector-ref node 3))
(define (node:get-local-ids node) (vector-ref node 4))
(define (node:add-element node element index)
```

(vector-set! node 3 (cons element (vector-ref node 3)))
(vector-set! node 4 (cons index (vector-ref node 4))))

## C.2.22 opalg.scm

```
;;; This file defines abstract operator algebra. It turns out to be more
;;; general than what we need. If we're going to be this abstract anyway, why
;;; don't we just use the polynomial code? Anyway, this is the wrong thing, so
;;; we won't continue along this line of work. Let's keep it around, though,
;;; just in case we need it someday.
(declare (usual-integrations))
```

;; We use the multi-index notation:
(define (make-multi-index $n$ )
(make-vector n 0 ) )
(define multi-index vector)
(define multi-index:length vector-length)
(define multi-index:get vector-ref)
(define multi-index:set! vector-set!)
(define (multi-index:sum multind)
(let ((len (multi-inde:length multind)))
(let loop ((sum 0) (i 0))
(if (< i len)
(loop (+ sum (multi-index:get multind i)) (+ i 1))
sum))))
(define (mult-index:fact multind)
(let ((len (multi-index:length multind)))
(let loop ((prod 1) (i 0))
(if (< i len)
(loop (* prod (factorial (mult-index:get multind i))) (+ i 1))
prod)))
(define (multi-index:+ mi1 mi2)
(let* ((len (multi-index:length mi1))
(result (make-multi-index len)))
(do ( $(\mathrm{i} 0$ (+ i 1)))
( $(>=\mathrm{i}$ len) result)
(multi-index:set! result i
(+ (multi-index:get mi1 i) (multi-index:get mi2 i)))))
;;; Make a partial differential operator with one summand:
(define (multi-index->diffop multind)
'((1 ,multind)))
;;; Elementary operations:

```
(define (diffop:+ op1 op2)
    (append op1 op2))
(define (diffop:- op1 op2)
    (append op1 (diffop:negate op2)))
(define diffop:first-term car)
(define diffop:remaining-terms cdr)
(define (diffop:zero? op)
    (null? (diffop:simplify op)))
;;; Composing differential operators:
(define (diffop:compose op1 op2)
    (if (diffop:zero? op1)
            op2
            (diffop:compose (diffop:remaining-terms op1)
                    (let ((op1 (diffop:first-term op1)))
                        (append-map
                                (lambda (op2)
                            (diffop:compose-terms op1 op2))
                op2)))))
```


## C.2.23 operators.scm

```
;;; This file contains some ad-hoc definitions of simple differential
;;; operators, such as the two-dimensional Laplacian or the 1+1-dimensional
;;; d'Alembertian. It also defines some important constructors and operations
;;; on differential operators that compute the adjoint and encapsulate
;;; integration-by-parts. (See ELEMENT-MAKER in fem.scm.)
;;; opalg.scm contains the beginnings of a much more abstract (and complete)
;;; approach.
;;; This file uses various procedures from basis.scm.
(declare (usual-integrations))
;;; Simple constructor and methods for an operator structure:
(define (make-operator left-op right-op combine)
    (vector left-op right-op combine))
(define (operator:get-left-op operator)
    (vector-ref operator 0))
(define (operator:get-right-op operator)
    (vector-ref operator 1))
(define (operator:get-combine operator)
    (vector-ref operator 2))
```

```
(define (operator:get-local-form op)
    (let ((combine (operator:get-combine op))
                (left-op (operator:get-left-op op))
                (right-op (operator:get-right-op op)))
    (lambda (f g)
            (combine (left-op f) (right-op g)))))
```


## C.2.24 relax.scm

;;; Matrix inversion using successive overrelaxation methods (SOR):
;;
;;; We use classical SOR methods to solve linear systems of equations.
;;; The representation for matrices is defined in sparse.scm.
(declare (usual-integrations))
; (load "sparse")
;; ; Note that this procedure first modifies the matrix by dividing through by
;;; the diagonal...
(define (sor sm n . aux)
(let* ((nrows (sparse-matrix-row-count sm))
(ncols-1 (- (sparse-matrix-column-count sm) 1))
(rhs (make-vector nrows))
(state (make-vector nrows 0))
(factor 1)
(residual 0 ))
; ; Parse auxiliary arguments, if any:
;; (The first should be SOR factor, the second an alternate state.)
(if (not (null? aux))
(if (not (null? (cdr aux)))
(set! state (cadr aux))
(set! factor (car aux))))
; ; Normalize the matrix:
(do ((io (+ i 1)))
( $(>=i$ nrows))
(let ((diag (sparse-matrix-ref sm i i)))
(for-each
(lambda (pair)
(sparse-matrix-set! sm i (car pair) (/ (cadr pair) diag)))
(sparse-matrix-get-row sm i))))
; ; Collect the right hand side of the equation $A x=b$ :
(do ( i 0 (+ i 1)))
((>= i nrows))
(vector-set! rhs i (sparse-matrix-ref sm i ncols-1)))
; ; Now perform SOR (note that we have normalized the matrix so that the
; ; diagonal terms are all unity):
(do ( $(\mathrm{n} \operatorname{n}(-\mathrm{n} 1)))$

```
        ((<= n 0))
    (set! residual 0.)
    (do ((i O (+ i 1)))
        ((>= i nrows))
    ;; Compute the row sum:
    (let ((sum 0))
        (for-each
            (lambda (pair)
            (let ((index (car pair))
                        (val (cadr pair)))
                    (if (< index ncols-1)
                            (set! sum (+ sum (* (vector-ref state index) val)))))
            (sparse-matrix-get-ron sm i))
        ;; Step forward:
        (let ((step (- (vector-ref rhs i) sum)))
            (if (> (abs step) residual)
                (set! residual (abs step)))
            (vector-set! state i (+ (vector-ref state i) (* factor step))))))
(write-line '(residual: ,residual))
state))
```


## C.2.25 sparse.scm

;; This file describes a data structure useful for describing sparse matrices.
;;; It is geared towards saving space, and is rather handy for performing SOR
;; ; on large matrices or assembling finite-element equations.

```
(declare (usual-integrations))
```

```
;;; Basic data structure and associated methods:
(define (make-sparse-matrix nrows ncols)
    (list (make-vector nrows '()) nrows ncols))
(define sparse-matrix-row-count cadr)
(define sparse-matrix-column-count caddr)
(define sparse-matrix-size cdr)
(define (sparse-matrix-ref sm i j)
    (let ((m (sparse-matrix-row-count sm))
            (n (sparse-matrix-column-count sm)))
        (if (or (< i 0) (>= i m) (< j 0) (>= j n))
            (error "Matrix access out of bound. -- SPARSE-MATRIX-REF"))
        (let ((result (assq j (vector-ref (car sm) i))))
            (if result
                        (cadr result)
```

0)) )

```
(define (sparse-matrix-set! sparse i j val)
    (let ((m (sparse-matrix-row-count sparse))
            (n (sparse-matrix-column-count sparse))
            (sm (car sparse)))
        (if (or (< i 0) (>= i m) (< j 0) (>= j n))
            (error "Matrix access out of bound. -- SPARSE-MATRIX-SET!"))
        (let ((result (assq j (vector-ref sm i))))
            (if (zero? val)
                (if result
                        (vector-set! sm i (all-but (vector-ref sm i) result)))
                (if result
                        (set-cdr! result (list val))
                        (vector-set! sm i (cons (list j val) (vector-ref sm i)))))))
(define (sparse-matrix-get-row sm i)
    (let ((m (sparse-matrix-row-count sm)))
        (if (or (< i 0) (>= i m))
            (error "Matrix access out of bound. -- SPARSE-MATRIX-GET-ROW"))
        (vector-ref (car sm) i)))
(define sparse-matrix-get-rows car)
(define (sparse-matrix-get-column sm j)
    (let ((m (sparse-matrix-row-count sm)))
        (let next-row ((i 0) (result '()))
            (if (< i m)
                        (let next-term ((row (sparse-matrix-get-row sm i)))
                        (if (null? row)
                        (next-row (+ i 1) result)
                        (if (= (caar row) j)
                            (next-row (+ i 1) (cons (list i (cadar row)) result))
                    (next-term (cdr row)))))
                            result)))(
(define (sparse-matrix-get-columns sm)
    (let ((m (sparse-matrix-row-count sm))
            (v (make-vector (sparse-matrix-column-count sm) '())))
        (let next-row ((i 0))
            (if (< i m)
                        (let next-term ((row (sparse-matrix-get-row sm i)))
                                (if (null? row)
                            (next-row (+ i 1))
                            (let ((j (caar row)))
                                    (vector-set! v j (cons (list i (cadar row))
                                    (vector-ref v j)))
                                (next-term (cdr row)))))
                        v))))
;;; A predicate that can come in handy:
(define (sparse-matrix? sm)
    (call-with-current-continuation
```

```
    (lambda (exit)
        (if (and (list? sm) (= (length sm) 3))
            (let ((m (cadr sm))
                    (n (caddr sm))
                    (sm (car sm)))
                (if (and (integer? m) (integer? n) (> m 0) (> n 0) (vector? sm))
                    (do ((i 0 (+ i 1)))
                        ((>= i m))
                        (let ((l (vector-ref sm i))) .
                        (if (or (not (list? l))
                                    (memq #f (map list? 1)))
                                    (exit #f))))
                #f)
                #t)
            #f)))(
```

```
;;; Converters:
```

;;; Converters:
(define (sparse->matrix sm)
(define (sparse->matrix sm)
(let* ((m (sparse-matrix-row-count sm))
(let* ((m (sparse-matrix-row-count sm))
(n (sparse-matrix-column-count sm))
(n (sparse-matrix-column-count sm))
(matrix (make-matrix m n)))
(matrix (make-matrix m n)))
(do ((i O (+ i 1)))
(do ((i O (+ i 1)))
((>= i m) matrix)
((>= i m) matrix)
(for-each
(for-each
(lambda (pair)
(lambda (pair)
(matrix-set! matrix i (car pair) (cadr pair)))
(matrix-set! matrix i (car pair) (cadr pair)))
(sparse-matrix-get-row sm i)))))
(sparse-matrix-get-row sm i)))))
(define (matrix->sparse matrix)
(define (matrix->sparse matrix)
(let* ((m (matrix-row-count matrix))
(let* ((m (matrix-row-count matrix))
(n (matrix-column-count matrix))
(n (matrix-column-count matrix))
(sm (make-sparse-matrix m n)))
(sm (make-sparse-matrix m n)))
(do ((i O (+ i 1)))
(do ((i O (+ i 1)))
((>= i m) sm)
((>= i m) sm)
(do ((j O (+ j 1)))
(do ((j O (+ j 1)))
((>= j n))
((>= j n))
(sparse-matrix-set! sm i j (matrix-ref matrix i j))))))
(sparse-matrix-set! sm i j (matrix-ref matrix i j))))))
;;; Very useful routine:
(define (print-sparse-matrix matrix . argl)
(if (null? argl)
(set! argl (list (current-output-port))))
(if (not (and (null? (cdr argl)) (output-port? (car argl))))
(error "Invalid argument(s) -- PRINT-MATRIX"))
(let ((port (car argl)))
(newline port)
(let ((m (sparse-matrix-row-count matrix))
(n (sparse-matrix-column-count matrix)))
(do ((i 0 (+ i 1)))
((>= i m))
(display (sparse-matrix-ref matrix i 0) port)
(do ((j 1 (+ j 1)))

```
```

            ((>= j n))
        (display #\tab port)
        (display (sparse-matrix-ref matrix i j) port))
    (newline port)))))
    ```
```

;;; Prepare for least squares on sparse matrices:

```
;;; Prepare for least squares on sparse matrices:
(define (sparse-normal-equations mat)
(define (sparse-normal-equations mat)
    (let* ((m (sparse-matrix-row-count mat))
    (let* ((m (sparse-matrix-row-count mat))
            (n+1 (sparse-matrix-column-count mat))
            (n+1 (sparse-matrix-column-count mat))
            (n (- n+1 1))
            (n (- n+1 1))
            (out (make-sparse-matrix n n+1))
            (out (make-sparse-matrix n n+1))
            (columns (sparse-matrix-get-columns mat)))
            (columns (sparse-matrix-get-columns mat)))
            ;; Compute the normal equations:
            ;; Compute the normal equations:
        (do ((i O (+ i 1)))
        (do ((i O (+ i 1)))
            ((>= i n) out)
            ((>= i n) out)
                (let ((ith-column (vector-ref columns i)))
                (let ((ith-column (vector-ref columns i)))
                    ;; First, the diagonal:
                    ;; First, the diagonal:
            (sparse-matrix-set! out i i (sparse-dot ith-column ith-column))
            (sparse-matrix-set! out i i (sparse-dot ith-column ith-column))
            ;; Next, the off-diagonal terms:
            ;; Next, the off-diagonal terms:
            (do ((j (+ i 1) (+ j 1)))
            (do ((j (+ i 1) (+ j 1)))
                ((>= j n))
                ((>= j n))
                (let ((val (sparse-dot ith-column (vector-ref columns j))))
                (let ((val (sparse-dot ith-column (vector-ref columns j))))
                        (sparse-matrix-set! out i j val)
                        (sparse-matrix-set! out i j val)
                        (sparse-matrix-set! out j i val)))
                        (sparse-matrix-set! out j i val)))
            ;; Finally, the RHS:
            ;; Finally, the RHS:
            (sparse-matrix-set! out i n (sparse-dot ith-column
            (sparse-matrix-set! out i n (sparse-dot ith-column
                                    (vector-ref columns n)))))))
(define (sparse-dot a b)
    (let a-loop ((a a) (sum 0))
        (if (null? a)
            sum
            (let ((id (caar a)))
                (let b-loop ((b b))
                    (if (null? b)
                        (a-loop (cdr a) sum)
                            (if (= id (caar b))
                            (a-loop (cdr a) (+ sum (* (cadar a) (cadar b))))
                                (b-loop (cdr b)))))))))
```


## C.2.26 thesis.scm

(load "collect")
; ; ; Test cases:

```
(test-1 '(3 3 2.) "Data/thesis/test1a")
(test-1 '(4 4 2.) "Data/thesis/test1b")
(test-1 '(5 5 2.) "Data/thesis/test1c")
(test-1 '(6 6 2.) "Data/thesis/test1d")
(test-1 '(7 7 2.) "Data/thesis/test1e")
(test-1 '(8 8 2.) "Data/thesis/test1f")
(test-1 '(9 9 2.) "Data/thesis/test1g")
(test-1 '(10 10 2.) "Data/thesis/test1h")
(test-1 '(11 11 2.) "Data/thesis/test1i")
(test-1 '(12 12 2.) "Data/thesis/test1j")
(test-1 '(13 13 2.) "Data/thesis/test1k")
(test-1 '(14 14 2.) "Data/thesis/test1l")
(test-1 '(15 15 2.) "Data/thesis/test1m")
(test-2 '(3 3) "Data/thesis/test2a")
(test-2 '(4 4) "Data/thesis/test2b")
(test-2 '(5 5) "Data/thesis/test2c")
(test-2 '(6 6) "Data/thesis/test2d")
(test-2 '(7 7) "Data/thesis/test2e")
(test-2 '(8 8) "Data/thesis/test2f")
(test-2 '(9 9) "Data/thesis/test2g")
(test-2 '(10 10) "Data/thesis/test2h")
(test-2 '(11 11) "Data/thesis/test2i")
(test-2 '(12 12) "Data/thesis/test2j")
(test-2 '(13 13) "Data/thesis/test2k")
(test-2 '(14 14) "Data/thesis/test2l")
(test-2 '(15 15) "Data/thesis/test2m")
(test-1 '(15 15 2.1) "Data/thesis/test3m")
(test-1 '(15 15 2.2) "Data/thesis/test4m")
(test-1 '(15 15 2.3) "Data/thesis/test5m")
(test-1 '(15 15 2.4) "Data/thesis/test6m")
(test-1 '(15 15 2.5) "Data/thesis/test7m")
(test-1 '(15 15 2.6) "Data/thesis/test8m")
(test-1 '(15 15 2.7) "Data/thesis/test9m")
(test-1 '(15 15 2.8) "Data/thesis/test10m")
(test-1 '(15 15 2.9) "Data/thesis/test11m")
(test-1 '(15 15 3.0) "Data/thesis/test12m")
```


## C.2.27 thesis1.scm

```
(load "collect")
```

; ; ; Test cases (the first run somehow stopped in the middle, probably because
;;; of errors in LU-solve):

```
(test-3 '(15 15 2.7) 30000 1.5 "Data/thesis/test9m")
(test-3 '(15 15 2.8) 30000 1.5 "Data/thesis/test10m")
(test-3 '(15 15 2.9) 30000 1.5 "Data/thesis/test11m")
(test-3 '(15 15 3.0) 30000 1.5 "Data/thesis/test12m")
```


## C.2.28 util-too.scm

;;; Miscellaneous mathematical helpers that are useful:
(declare (usual-integrations))

```
;;; Some combinatorial things:
(define (choose n r)
    ;; Compute nCr:
    (/ (factorial n) (factorial r) (factorial (- n r))))
(define (slow-factorial n)
    (let loop ((n n) (result 1))
        (if (> n 1)
                            (loop (- n 1) (* result n))
            result)))
(define factorial (simple-memoize slow-factorial 100))
(define (pairs 1)
    (let loop ((1 1) (result '()))
        (if (null? 1)
            result
            (loop (cdr 1)
                (let ((a (car l)))
                    (let loop ((l (cdr 1)) (result result))
                        (if (null? 1)
                        result
                        (loop (cdr l) (cons (list a (car l)) result))))))))
;;; Forming the list of all sublists of L of length N is a bit more
;;; complicated.
(define (choose-sublists 1 n)
    (if (or (null? l) (<= n 0))
            '(())
            (let loop ((l l) (n n) (k (- (length l) n)) (result '()))
                    (cond ((< k 0) result)
                        ((zero? k) (cons l result))
                        ((=n 1) (append (map list 1) result))
                        (else
                        (let ((first (car l))
                            (rest (cdr l)))
                                (append
                                result
                                (loop rest n (- k 1)
                            (map (lambda (sublist) (cons first sublist))
                                    (loop rest (- n 1) k '())))))))))
```

;;; This procedure converts references to entries in an NxN symetric matrix
;; into a vector representation.
(define (symmetric->vector-index $i \quad j$ )
(if (>= i j )
(if (= i 0 )
0
(+ (choose (+ i 1) 2) j))
(if (= j 0)

```
            0
            (+ (choose (+ j 1) 2) i))))
```

```
;;; Why isn't this built into Scheme?
```

;;; Why isn't this built into Scheme?
(define (all-but list item)
(define (all-but list item)
(let loop ((head '()) (tail list))
(let loop ((head '()) (tail list))
(if (null? tail)
(if (null? tail)
list
list
(if (eq? (car tail) item)
(if (eq? (car tail) item)
(append (reverse head) (cdr tail))
(append (reverse head) (cdr tail))
(loop (cons (car tail) head) (cdr tail))))))
(loop (cons (car tail) head) (cdr tail))))))
;;; It's useful to find the bounding box of a finite subset of an Euclidean
;;; space:
(define (bounding-box nodes get-coords)
(let* ((p (get-coords (car nodes)))
(dim (vector-length p))
(best (let ((l (vector->list p)))
(list->vector (map cons 1 l)))))
(let loop ((nodes (cdr nodes)))
(if (null? nodes)
(let ((l (vector->list best)))
(append (map car 1) (map cdr 1)))
(let ((p (get-coords (car nodes))))
(do ((i O (+ i 1)))
((>= i dim))
(let ((pair (vector-ref best i))
(val (vector-ref p i)))
(cond ((< val (car pair)) (set-car! pair val))
((> val (cdr pair)) (set-cdr! pair val)))))
(loop (cdr nodes))))))

```

\section*{C.2.29 util.scm}
```

;;; Memoization is frequently useful:

```
;;; Memoization is frequently useful:
(declare (usual-integrations))
;;; ALWAYS useful:
(define (square z)
    (* z z))
(define (cube z)
    (* z z z))
;;; Even more useful:
(define (simple-memoize proc size)
    ;; Memoize a function whose argument is a non-negative integer:
    (let ((cache (make-vector size 'undefined)))
```

```
    (lambda (n)
    (if (>= n size)
            (proc n)
            (let ((val (vector-ref cache n)))
                (if (eq? val 'undefined)
                    (let ((val (proc n)))
                        (vector-set! cache n val)
                        val)
                    val)))())
(define almost-zero?
    (let ((*tolerance* 1e-14))
        (lambda (z)
            (< (magnitude z) *tolerance*))))
;;; Why is this not defined elsewhere?
(define (compose f . rest)
    (let loop ((f f) (l rest))
        (if (null? 1)
            f
                            (loop (binary-compose f (car l)) (cdr l)))))
(define (binary-compose f g)
    (lambda (first . rest)
        (f (apply g (cons first rest)))))
```


## C. 3 Finite differences

The following programs implement the finite difference algorithms discussed in Chapter 3. They have been written in C for speed, and hence may be even less coherent than the previous sections.

## C.3.1 farray.h

```
typedef struct {
    int size;
    double *array;
} farray;
farray *farray_cons(int);
void farray_free(farray *);
double fa_ref(farray *, int);
void fa_set(farray *, int, double);
int fa_size(farray *);
```


## C.3.2 gunk.h

```
#include <stdio.h>
#define STRLEN 256
#define pi 3.141592653589793
#define TOLERANCE 1e-16
```

```
#define SQUARE(x) ((x)*(x))
```

```
/**
    * Currently supported node types include interior and boundary.
    **/
```

\#define INTERIOR_NODE 0
\#define BOUNDARY_NODE 1
/**
* This is a kluge. It is especially useful for keeping track of the evolution
* of file formats.
**/
\#define GUNK_VERSION 1

## /****************************************

* These macros decide which potential *
* function we use.


```
/* #define POT1 */
```

/* \#define POT2 */
typedef struct CELL \{
int index;
struct CELL *next;
\} cell;
typedef struct \{
double $\mathrm{x}, \mathrm{y}, \mathrm{z}$;
cell *neighbors;
int type;
\} node;
typedef struct \{
cell **list;
int size;
$\}$ partition;

```
/***************************
    * Function prototypes *
    ************************/
void old_write_elements(FILE *, int, node *);
void write_elements(FILE *, int, node *);
void old_read_elements(FILE *, int, node *);
void read_elements (FILE *, int, node *);
int read_neighbor(FILE *);
void putlf(FILE *, double);
void getlf(FILE *, double *);
void putint(FILE *, int);
void getint(FILE *, int *);
```

```
void free_memory(int, node *);
double potential(double, double);
double fsquare(double);
double distance(node *, node *);
partition *make_partition(int);
void partition_add(partition *, int, int, int);
cell *partition_get(partition *, int, int);
void free_partition(partition *);
void introduce(node *, int, int, double);
int interior_node(node *);
int boundary_node(node *);
void set_boundary_node(node *);
void set_interior_node(node *);
```


## C.3.3 matrix.h

```
#include "farray.h"
#include <stdio.h>
#define TOL tolerance()
static double tolerance_value = 1e-10;
typedef struct {
    int nrows, ncols;
    double **el;
} matrix;
matrix *matrix_cons(int, int);
void matrix_free(matrix *);
double mref(matrix *, int, int);
void mset(matrix *, int, int, double);
void exchange_rows(matrix *, int, int);
void scale_row(matrix *, int, double);
void add_rows(matrix *, int, double, int);
void madd(matrix *, matrix *, matrix *);
void mmult(matrix *, matrix *, matrix *);
matrix *transpose(matrix *);
double rref(matrix *);
double ludecomp(matrix *);
void svdecomp(matrix *, farray *, matrix *);
void mprint(FILE *, matrix *);
void mprintf(FILE *, char *, matrix *);
double tolerance(void);
void set_tolerance(double);
```


## C.3.4 random.h

```
#define R 43
#define R_1 42
#define S 22
#define R_S 21
#define B 4294967291
#define B_1 4294967290
#define double_B ((double)4294967291)
#define TRANSIENT 100000
static struct {
    unsigned long vector[R];
    unsigned long borrow;
    int index;
} random_state = {{
    417256956, 1236137991, 1100630827, 2433022217, 2575904056,
    177405986, 526582298, 3028657187, 2171035790, 2723919184,
    2986817232, 1113054914, 626934711, 19344552, 731146319,
    914274156, 548850110, 1944853448, 3801326350, 2719133088,
    3577421839, 86246253, 2765784422, 561513528, 3851288197,
    2574098718, 744615398, 1598791186, 2262866255, 2248738083,
    1670448839, 985226499, 4058327925, 177492605, 3322737975,
    1918398670, 3765485260, 629347060, 553460448, 278339973,
    1688376297, 1125926514, 2267411077}, 1, 35};
#endif
double frandom(void);
unsigned long lrandom(void);
void initialize_random(void);
```


## C.3.5 stat.h

```
void sqstat(FILE *, int, node *);
void nstat(FILE *, int, node *);
```


## C.3.6 accum.c

```
/**************
```

    * accum.c *
    *************
    Convert a list of tuples of the form ( $x, y$, val, ref) into a rectangular grid of something-or-other, readable by MATLAB or Maple. Just like list2grid.c or slice.c.

## */

```
#include <stdio.h>
#include <math.h>
#include "matrix.h"
matrix *accum(FILE *, int, int, int);
```

```
main(int argc, char *argv[])
{
    FILE *fin, *fout;
    int m, n, opt = 0;
    matrix *mat;
    if (argc != 5 && argc != 6) {
        fprintf(stderr, "Usage: %s input-file output-file x-count y-count [opt]\n",
                    argv[0]);
        exit(1);
    }
    /* Open input file. */
    fin = fopen(argv[1], "r");
    if (fin == NULL) {
        fprintf(stderr, "Error: Cannot read file \"%s\".\n", argv[1]);
        exit(2);
    }
    sscanf(argv[3], "%d", &m);
    sscanf(argv[4], "%d", &n);
    if (argc == 6)
        sscanf(argv[5], "%d", &opt);
    printf("Constructing %dx%d grid and accumulating data...\n", m, n);
    getc(fin); /* Pop the first character in the file. */
    mat = accum(fin, m, n, opt);
    fclose(fin);
    /* Open output file. */
    fout = fopen(argv[2], "w");
    if (fout == NULL) {
        fprintf(stderr, "Error: Cannot write to file \"%s\".\n", argv[2]);
        exit(3);
    }
    mprint(fout, mat);
    fclose(fout);
    matrix_free(mat);
    return 0;
}
matrix *accum(FILE *fin, int m, int n, int opt)
{
    double x, y, val, ref, x_max, x_min, y_max, y_min, dx, dy;
    int i, j;
    matrix *mat, *count;
    if (opt < 0 || opt > 3) {
        fprintf(stderr, "Valid options:\n\n");
        fprintf(stderr, "0. Save absolute error.\n");
        fprintf(stderr, "1. Save relative error.\n");
        fprintf(stderr, "2. Save reference value.\n");
        fprintf(stderr, "3. Save computed value.\n\n");
```

```
    exit(4);
}
mat = matrix_cons(m, n);
count = matrix_cons(m, n);
fscanf(fin, "%lg\t%lg\t%lg\t%lg\n", &x, &y, &val, &ref);
x_max = x_min = x;
y_max = y_min = y;
while (fscanf(fin, "%lg\t%lg\t%lg\t%lg\n", &x, &y, &val, &ref) != E0F) {
    if (x > x_max)
        x_max = x;
    else if (x < x_min)
            x_min = x;
        if (y > y_max)
            y_max = y;
    else if (y < y_min)
            y_min = y;
}
dx = (x_max - x_min)/(m + 1);
dy = (y_max - y_min)/(n + 1);
fprintf(stderr, "x_max = %lg, x_min = %lg, y_max = %lg, y_min = %lg.\n",
                    x_max, x_min, y_max, y_min);
rewind(fin);
while (fscanf(fin, "%lg\t%lg\t%lg\t%lg\n", &x, &y, &val, &ref) != EOF) {
    i = (int) ((x - x_min)/dx);
    j = (int)((y - y_min)/dy);
    if (i >= m) i = m - 1;
    if (j >= n) j = n - 1;
    switch(opt) {
    case 0:
        mset(mat, i, j, mref(mat, i, j) + fabs(val - ref));
        break;
    case 1:
        mset(mat, i, j, mref(mat, i, j) + (ref == 0.0 ? 0.0 : (val - ref)/ref));
        break;
    case 2:
        mset(mat, i, j, mref(mat, i, j) + ref);
        break;
    case 3:
        mset(mat, i, j, mref(mat, i, j) + val);
        break;
    default:
        mset(mat, i, j, mref(mat, i, j) + val);
        break;
    }
    mset(count, i, j, mref(count, i, j) + 1);
}
for (i = 0; i < m; i++)
    for (j = 0; j < n; j++)
```

```
            if (mref(count, i, j) > 0)
                mset(mat, i, j, mref(mat, i, j)/mref(count, i, j));
    matrix_free(count);
    return mat;
}
```


## C.3.7 approx.c

```
/**************
* approx.c *
***************
```

Input: A gunk file, such as the output of mkbd or mkgrid.
Output: Local finite-difference approximations of the Laplacian operator, obtained via polynomial interpolation.

```
*/
```

```
#include <stdio.h>
#include <math.h>
#include "gunk.h"
#include "matrix.h"
/* #define NORMALIZE */
#define CACHE_SIZE 105
void init_op(int, node *, farray **);
void gen_coeff(int, node *, farray **);
void check_coeff(int, farray **);
void normalize_coeff(int, farray **);
void lsqfit(int, node *, farray *);
void memo_walk(int, int *, int *);
void walk(int, int *, int *);
double monomial(double, double, int, int);
double laplace_monomial(double, double, int, int);
int term_count(int);
void average(int, node *, farray *);
void diamond(int, node *, farray *);
main(int argc, char *argv[])
{
    farray **coeff;
    int size, i, j, N;
    FILE *fp;
    node *elements;
    if (argc != 3 && !(argc == 4 && (strcmp(argv[3], "-ascii") == 0))) {
        fprintf(stderr, "Usage: %s input-file output-file [-ascii]\n",
                            argv[0]);
        exit(1);
```

```
    fp = fopen(argv[1], "r");
    if (fp == NULL) {
        fprintf(stderr, "Error: Cannot open file \"%s\"\n", argv[1]);
        exit(2);
    }
    fscanf(fp, "%d", &size);
    elements = (node *)calloc(sizeof(node), size);
    fprintf(stderr, "Reading input file (%d nodes)...\n", size);
    read_elements(fp, size, elements);
    fclose(fp);
    fprintf(stderr, "Initializing differential operator coefficients...\n");
    coeff = (farray **)calloc(sizeof(farray *), size);
    init_op(size, elements, coeff);
    gen_coeff(size, elements, coeff);
#ifdef NORMALIZE
    normalize_coeff(size, coeff);
#endif
    check_coeff(size, coeff);
    fp = fopen(argv[2], "w");
    fprintf(stderr, "Writing output file...\n");
    if (argc == 4) {
        for (i = 0; i < size; i++) {
        N = fa_size(coeff[i]);
        fprintf(fp, "%d", N);
        for (j = 0; j < N; j++)
            fprintf(fp, " %.16lg", fa_ref(coeff[i], j));
        fprintf(fp, "\n");
    }
}
else {
    for (i = 0; i < size; i++) {
            N = fa_size(coeff[i]);
            putint(fp,N);
            for (j = 0; j < N; j++)
            putlf(fp, fa_ref(coeff[i], j));
    }
}
fclose(fp);
fprintf(stderr, "Freeing memory...\n");
free_memory(size, elements);
for (i = 0; i < size; i++)
    farray_free(coeff[i]);
free(coeff);
return 0;
```

```
}
void init_op(int size, node *elements, farray **coeff)
{
    cell *cp;
    int i, length;
    for (i = 0; i < size; i++)
        if (interior_node(elements + i)) {
            length = 0;
                for (cp = elements[i].neighbors; cp != NULL; cp = cp->next)
                length++;
            coeff[i] = farray_cons(length + 1);
        }
}
void gen_coeff(int size, node *elements, farray **coeff)
{
    int n;
    for (n = 0; n < size; n++)
        if (interior_node(elements + n))
            lsqfit(n, elements, coeff[n]);
}
void check_coeff(int size, farray **coeff)
{
    double sum;
    int n, i, N, count = 0;
    for (n = 0; n < size; n++) {
        N = fa_size(coeff[n]);
        if (N > 0) {
            sum = 0.0;
            for (i = 1; i < N; i++)
                sum += fabs(fa_ref(coeff[n], i));
            if (sum > fabs(fa_ref(coeff[n], 0)))
                count++;
        }
    }
    if (count > 0)
        fprintf(stderr, "Warning: Matrix not diagonally dominant! (%d/%d)\n",
                        count, size);
}
void memo_walk(int n, int *x, int *y)
{
    static int xi[CACHE_SIZE], yi[CACHE_SIZE], filled[CACHE_SIZE];
    static int first_time = 1;
    int k;
    if (n < CACHE_SIZE) {
        if (first_time) {
```

```
            for (k = 0; k < CACHE_SIZE; k++)
                filled[k] = 0;
            first_time = 0;
        }
        if (!filled[n]) {
            walk(n, xi + n, yi + n);
            filled[n] = 1;
        }
        *x = xi[n];
        *y = yi[n];
    }
    else
        walk(n, x, y);
}
void walk(int n, int *x, int *y)
{
    int i=0,j=0,k;
    for (; n > 0; n--)
        if (i == j) {
            i++;
            j = 0;
        }
        else if (i > j) {
            k = i;
            i=j;
            j = k;
        }
        else {
            k = i + 1;
            i = j;
            j = k;
        }
    *x = i;
    *y = j;
}
```

double monomial (double $x$, double $y$, int $i$, int $j$ )
$\{$
double product $=1.0$;
for (; i > 0 ; i--) product *= $x$;
for (; $j>0$; $j--$ ) product *= $y$;
return product;
\}
double laplace_monomial(double $x$, double $y$, int $i$, int $j$ )
\{
double $\operatorname{xprod}=\mathrm{i} *(\mathrm{i}-1), \operatorname{yprod}=j *(j-1)$;
int $k$;
for ( $k=2$; $k<i ; k++$ ) xprod *= $x$;
for ( $k=0 ; k<j ; k++$ ) xprod $*=y$;

```
    for (k = 2; k < j; k++) yprod *= y;
    for (k = 0; k < i; k++) yprod *= x;
    return xprod + yprod;
}
int term_count(int n)
{
    /* n = (int)ceil(sqrt(8.0*n + 1.0)/2.0-1.5);
        n = ((n + 1)*(n + 2))/2; */
    return (n > 6) ? n : 6;
}
void lsqfit(int index, node *elements, farray *coeff)
{
    FILE *fp;
    static int print = 0, first_time = 0;
    cell *cp;
    double x = elements[index].x, y = elements[index].y, sum;
    farray *W;
    int i, j, M, N, p, q, other, bad_count = 0;
    matrix *U, *V;
    N = fa_size(coeff);
    M = term_count(N);
    U = matrix_cons(M, N);
    V = matrix_cons(N,N);
    W = farray_cons(N);
    if (first_time) {
        fp = fopen("matrix", "w");
        fprintf(fp, "%.16lg", elements[index].z);
        for (cp = elements[index].neighbors; cp != NULL; cp = cp->next)
            fprintf(fp, " %.16lg", elements[cp->index].z);
        fprintf(fp, "\n");
    }
    for (i = 0; i < M; i++) {
        memo_walk(i, &p, &q);
        other = index;
        cp = elements[index] .neighbors;
        for (j = 0; j < N; j++) {
            mset(U, i, j, monomial(elements[other].x - x, elements[other].y - y,
                                    p, q));
            if (first_time) fprintf(fp, "%.16lg ", mref(U, i, j));
            other = cp->index;
            cp = cp->next;
        }
        if (first_time) fprintf(fp, "\n");
    }
    svdecomp(U, W, V); /* use _Numerical Recipes_ */
```

```
    if (print) {
    printf("\nSingular values:\n");
    for (i = 0; i < N; i++)
        printf("%.16lg\n", fa_ref(W, i));
    printf("\nU =\n");
    mprint(stdout, U);
    printf("\nV =\n");
    mprint(stdout, V);
}
    for (i = 0; i < N; i++)
    if (fabs(fa_ref(W, i)) < TOL)
        fa_set(W, i, 0.0);
    else {
        sum = 0.0;
        for (j = 0; j < M; j++) {
            memo_walk(j, &p, &q);
            sum += laplace_monomial(0.0, 0.0, p, q)*mref(U, j, i);
        }
        fa_set(W, i, sum/fa_ref(W, i));
    }
for (i = 0; i < N; i++) {
    sum = 0.0;
    for (j = 0; j < N; j++)
        sum += mref(V, i, j)*fa_ref(W, j);
    fa_set(coeff, i, sum);
}
if (print) {
    print--;
    printf("\nCoefficients:\n");
    printf("(%lf, %lf)\t%.16lg\n", 0.0, 0.0, fa_ref(coeff, 0));
    for (cp = elements[index].neighbors, i = 1; i < N; i++, cp = cp->next)
        printf("(%.16lg, %.16lg)\t%.16lg\n",
                elements[cp->index].x - elements[index].x,
                elements[cp->index].y - elements[index].y,
                fa_ref(coeff, i));
    }
    if (first_time) {
        first_time = 0;
        fclose(fp);
        fprintf(stderr, "Done dumping.\n");
}
matrix_free(U);
matrix_free(V);
farray_free(W);
}
```

```
/*
** The following procedures are alternatives to lsqfit.
*/
void average(int index, node *elements, farray *coeff)
{
    int i, N = fa_size(coeff);
    for (i = 1; i < N; i++)
        fa_set(coeff, i, 1);
    fa_set(coeff, 0, -N + 1);
}
void diamond(int index, node *elements, farray *coeff)
{
    cell *cp;
    double max_r = 0.0, dist, x = elements[index].x, y = elements[index].y;
    int i,N = 0;
    if (interior_node(elements + index)) {
        for (cp = elements[index].neighbors; cp != NULL; cp = cp->next) {
            dist = distance(elements + index, elements + cp->index);
            if (dist > max_r)
                max_r = dist;
        }
        dist = max_r/sqrt(2);
        for (cp = elements[index].neighbors, i = 1;
                    cp != NULL;
                    cp = cp->next, i++)
            if (fabs(elements[cp->index].x - x) + fabs(elements[cp->index].y - y) <=
                    dist + TOLERANCE) {
                fa_set(coeff, i, 1);
                N++;
            }
            else
                fa_set(coeff, i, 0);
        fa_set(coeff, 0, -N);
    }
}
/**
    * This function divides through each row of the matrix by the diagonal term,
    * to help reduce the magnitude of coefficients.
    **/
void normalize_coeff(int size, farray **coeff)
{
    double val;
    int i, j, n;
    for (i = 0; i < size; i++)
        if (coeff[i] != NULL) {
```

```
        n = fa_size(coeff[i]);
        val = fa_ref(coeff[i], 0);
        for (j = 0; j < n; j++)
            fa_set(coeff[i], j, fa_ref(coeff[i], j)/val);
    }
}
```


## C.3.8 band.c

```
/************
* band.c *
*************
This program computes the bandwidth of the matrix
generated by a gunk system.
*/
#include <stdio.h>
#include <math.h>
#include "gunk.h"
#include "matrix.h"
void band(int, node *);
main(int argc, char *argv[])
{
    int size;
    node *elements;
    FILE *fp;
    if (argc != 2) {
        fprintf(stderr, "Usage: %s input-file\n", argv[0]);
        exit(1);
    }
    fp = fopen(argv[1], "r");
    if (fp == NULL) {
        fprintf(stderr, "Error: Cannot open file \"%s\"\n", argv[1]);
        exit(1);
    }
    fscanf(fp, "%d", &size);
    fprintf(stderr, "Reading input (%d nodes)...\n", size);
    elements = (node *)calloc(sizeof(node), size);
    read_elements(fp, size, elements);
    fclose(fp);
    band(size, elements);
    fprintf(stderr, "Freeing memory...\n");
    free_memory(size, elements);
    return 0;
}
```

```
void band(int size, node *elements)
{
    cell *cp;
    int real_size = 0, i, *index, band = 0, row, max_index1 = -1;
    int max_index2 = -1, max_index3 = -1;
    for (i = 0; i < size; i++)
        if (interior_node(elements + i))
            real_size++;
    index = (int *)calloc(sizeof(int), size);
    if (interior_node(elements))
        index[0] = 0;
    else
        index[0] = -1;
    for (i = 1; i < size; i++)
        if (interior_node(elements + i))
            index[i] = index[i - 1] + 1;
        else
            index[i] = index[i - 1];
    for (i = 0; i < size; i++)
        if (interior_node(elements + i)) {
            row = 0;
                for (cp = elements[i].neighbors; cp != NULL; cp = cp->next)
                    if (elements[cp->index].neighbors != NULL &&
                    fabs(index[i] - index[cp->index]) > row) {
                    row = fabs(index[i] - index[cp->index]);
                        max_index3 = cp->index;
                    }
            if (row > band) {
                band = row;
                        max_index1 = i;
                max_index2 = max_index3;
                }
        }
    free(index);
    printf("Node %d: at (%lf, %lf)\n",
            max_index1, elements[max_index1].x, elements[max_index1].y);
    printf("Node %d: at (%lf, %lf)\n",
                max_index2, elements[max_index2].x, elements[max_index2].y);
    printf("Bandwidth = %d\n", 2*band + 1);
}
```


## C.3.9 blud.c

## /************

* blud.c *
************

```
Load a file of gunk and operator coefficients and try to
invert the matrix using LU decomposition. This tries to take
advantage of banded matrices, so gsort might come in handy.
*/
#include <stdio.h>
#include <math.h>
#include "gunk.h"
#include "matrix.h"
#define MAX(X,Y) (((X) > (Y)) ? (X) : (Y))
#define MIN(X,Y) (((X) < (Y)) ? (X) : (Y))
int band(int, node *);
void solve(int, node *, farray **);
void read_coeff(FILE *, int, farray **);
int entry(int, int, int, int);
main(int argc, char *argv[])
{
    farray **coeff;
    int size, i;
    node *elements;
    FILE *fp;
    if (argc != 4) {
        fprintf(stderr, "Usage: %s input-file coeff output-file\n",
                argv[0]);
        exit(1);
    }
    fprintf(stderr, "Warning: This program has not been thoroughly tested\n");
    fprintf(stderr, " and may not operate correctly!\n");
    fp = fopen(argv[1], "r");
    if (fp == NULL) {
        fprintf(stderr, "Error: Cannot open file \"%s\"\n", argv[1]);
        exit(1);
    }
    fscanf(fp, "%d", &size);
    fprintf(stderr, "Reading input (%d nodes)...\n", size);
    elements = (node *)calloc(sizeof(node), size);
    read_elements(fp, size, elements);
    close(fp);
    fp = fopen(argv[2], "r");
    if (fp == NULL) {
        fprintf(stderr, "Error: Cannot open file \"%s\"\n", argv[2]);
        exit(1);
    }
    fprintf(stderr, "Reading coefficients...\n");
    coeff = (farray **)calloc(sizeof(farray *), size);
    read_coeff(fp, size, coeff);
```

```
    fclose(fp);
    fprintf(stderr, "Solving system of equations...\n");
    solve(size, elements, coeff);
    fprintf(stderr, "Writing output...\n");
    fp = fopen(argv[3], "w");
    write_elements(fp, size, elements);
    fclose(fp);
    fprintf(stderr, "Freeing memory...\n");
    free_memory(size, elements);
    for (i = 0; i < size; i++)
        farray_free(coeff[i]);
    free(coeff);
    return 0;
}
void read_coeff(FILE *fp, int size, farray **coeff)
{
    double val;
    int i, j, N;
    for (i = 0; i < size; i++) {
        getint(fp, &N);
        if (N > 0) {
            coeff[i] = farray_cons(N);
            for (j = 0; j < N; j++) {
                    getlf(fp, &val);
                    fa_set(coeff[i], j, val);
            }
        }
    }
}
void solve(int size, node *elements, farray **coeff)
{
    cell *cp;
    double sum, scale, mem = 0;
    farray **rows, *rhs;
    int real_size = 0, i, j, k, bw, B, index[size];
    /* solving Ax = u */
    /* set up the banded matrix A */
    for (i = 0; i < size; i++)
        if (interior_node(elements + i))
            real_size++;
    bw = band(size, elements);
    B = (bw - 1)/2;
    fprintf(stderr, " %dx%d matrix, bandwith = %d\n", real_size, real_size, bw);
    for (i = 0; i < size; i++)
```

```
if (bw > real_size) {
    fprintf(stderr, "Error: Matrix not banded\n");
    exit(1);
}
if (interior_node(elements))
    index[0] = 0;
else
    index[0] = -1;
for (i = 1; i < size; i++)
    if (interior_node(elements + i))
        index[i] = index[i - 1] + 1;
    else
            index[i] = index[i - 1];
rows = (farray **)calloc(sizeof(farray *), real_size);
rhs = farray_cons(real_size);
mem = fa_size(rhs)*sizeof(double) + real_size*sizeof(farray *);
for (i = 0; i < real_size; i++) {
    if (i + B < real_size)
        rows[i] = farray_cons(entry(real_size, B, i, i + B) + 1);
    else
        rows[i] = farray_cons(entry(real_size, B, i, real_size - 1) + 1);
    mem += sizeof(farray) + fa_size(rows[i])*sizeof(double);
}
fprintf(stderr, " Using approximately %.1lfK...\n", mem/1024);
for (i = 0; i < size; i++)
    if (interior_node(elements + i)) {
        sum = 0;
        fa_set(rows[index[i]],
                    entry(real_size, B, index[i], index[i]),
                    fa_ref(coeff[i], 0));
        for (cp = elements[i].neighbors, j = 1; cp != NULL; cp = cp->next, j++)
            if (interior_node(elements + cp->index))
                    fa_set(rows[index[i]],
                        entry(real_size, B, index[i], index[cp->index]),
                        fa_ref(coeff[i], j));
            else
                sum == fa_ref(coeff[i], j)*elements[cp->index].z;
        fa_set(rhs, index[i], sum);
    }
/* LU decomposition (without pivoting): A = LU */
fprintf(stderr, " Performing LU decomposition...\n");
for (j = 0; j < real_size; j++) {
    for (i = MAX(0, j - B); i <= j; i++) {
        sum = fa_ref(rows[i], entry(real_size, B, i, j));
        for (k = MaX(MAX (0, j - B), i - B); k < MIN(i, j + B); k++)
```

```
        sum -= fa_ref(rows[i], entry(real_size, B, i, k))*
                        fa_ref(rows[k], entry(real_size, B, k, j));
        fa_set(rows[i], entry(real_size, B, i, j), sum);
    }
    for (i = j + 1; i <= MIN(real_size - 1, j + B); i++) {
        sum = fa_ref(rows[i], entry(real_size, B, i, j));
        for (k = MAX (MAX (0, j - B), i - B); k < MIN(j, i + B); k++)
            sum -= fa_ref(rows[i], entry(real_size, B, i, k))*
            fa_ref(rows[k], entry(real_size, B, k, j));
        fa_set(rows[i], entry(real_size, B, i, j),
                sum/fa_ref(rows[j], entry(real_size, B, j, j)));
    }
}
/* forward substitution: Ly = u */
fprintf(stderr, " Forward substitution...\n");
for (j = 0; j < real_size; j++)
    for (i = j + 1; i <= MIN(real_size - 1, j + B); i++) {
        fa_set(rhs, i, fa_ref(rhs, i) -
            fa_ref(rows[i], entry(real_size, B, i, j))*fa_ref(rhs, j));
        fa_set(rows[i], entry(real_size, B, i, j), 0.0);
    }
/* backward substitution: Ux = y */
fprintf(stderr, " Backward substitution...\n");
for (j = real_size - 1; 0 <= j; j--) {
    scale = fa_ref(rows[j], entry(real_size, B, j, j));
    for (k = j; k <= MIN(real_size - 1, j + B); k++)
        fa_set(rows[j], entry(real_size, B, j, k),
                            fa_ref(rows[j], entry(real_size, B, j, k))/scale);
    fa_set(rhs, j, fa_ref(rhs, j)/scale);
    for (i = j - 1; MAX(0, j - B) <= i; i--) {
        scale = fa_ref(rows[i], entry(real_size, B, i, j));
        for (k = j; k <= MIN(real_size - 1, i + B); k++)
            fa_set(rows[i], entry(real_size, B, i, k),
                        fa_ref(rows[i], entry(real_size, B, i, k)) -
                            scale*fa_ref(rows[j], entry(real_size, B, j, k)));
        fa_set(rhs, i, fa_ref(rhs, i) - scale*fa_ref(rhs, j));
    }
}
/* store answers back */
fprintf(stderr, " Storing answers...\n");
for (i = 0; i < size; i++)
```

```
        if (interior_node(elements + i))
            elements[i].z = fa_ref(rhs, index[i]);
    /* free up memory */
    for (i = 0; i < real_size; i++)
        farray_free(rows[i]);
    farray_free(rhs);
}
int entry(int real_size, int B, int i, int j)
{
    if (fabs(i - j) <= B)
        if (i < B)
            return j;
        else
            return j - i + B;
    else {
        fprintf(stderr, "Error: Banded matrix reference out of bound (%d, %d)\n",
                    i, j);
        exit(1);
    }
}
int band(int size, node *elements)
{
    cell *cp;
    int real_size = 0, i, *index, band = 0, row;
    for (i = 0; i < size; i++)
        if (interior_node(elements + i))
            real_size++;
    index = (int *)calloc(sizeof(int), size);
    if (interior_node(elements))
        index[0] = 0;
    else
        index[0] = -1;
    for (i = 1; i < size; i++)
        if (interior_node(elements + i))
            index[i] = index[i - 1] + 1;
        else
            index[i] = index[i - 1];
    for (i = 0; i < size; i++)
        if (interior_node(elements + i)) {
        row = 0;
            for (cp = elements[i].neighbors; cp != NULL; cp = cp->next)
            if (interior_node(elements + cp->index) &&
                        fabs(index[i] - index[cp->index]) > row)
                row = fabs(index[i] - index[cp->index]);
        if (row > band) band = row;
        }
```

```
    free(index);
    return 2*band + 1;
}
```


## C.3.10 check.c

```
/************
* check.c *
*************
```

Input: A file of gunk, such as the output of mkbd or mkgrid.
This program checks the consistency between node positions and
neighborhood information.
*/
\#include <stdio.h>
\#include <math.h>
\#include "gunk.h"
void check(FILE *, int, node *, int);
int check_neighbors(node *, int, int);
int check_nhood(node *, int, int);
main(int argc, char *argv[])
\{
int size, cell_count;
node *elements;
FILE *fp;
if (argc != 3) \{
fprintf(stderr, "Usage: \%s file 1/radius\n",
$\operatorname{argv}[0]$ );
exit(1) ;
\}
sscanf(argv[2], "\%d", \&cell_count);
$f p=f o p e n(\operatorname{argv}[1], \quad " r ") ;$
if ( $f p==$ NULL) \{
fprintf(stderr, "Error: Cannot open file \"\%s\"\n", argv[1]);
exit(1);
\}
fscanf(fp, "\%d", \&size);
fprintf(stderr, "Reading input file (\% nodes)...\n", size);
elements = (node *) calloc (sizeof(node), size);
read_elements (fp, size, elements);
fclose(fp);
fprintf(stderr, "Checking neighbors...\n");
check(stderr, size, elements, cell_count);
fprintf(stderr, "Freeing memory...\n");

```
    free_memory(size, elements);
    return 0;
}
void check(FILE *fp, int size, node *elements, int cell_count)
{
    cell *cp;
    double radius = 1.0/cell_count, d;
    int n, i, j, p, q, case1_count = 0.0, case2_count = 0.0;
    partition *part = make_partition(cell_count);
    for (n = 0; n < size; n++)
        if (interior_node(elements + n)) {
            i = (int)floor(elements[n].x*cell_count);
            j = (int)floor(elements[n].y*cell_count);
            for (p = i - 1; p <= i + 1; p++)
                for (q = j - 1; q<= j + 1; q++)
                if ( }0<=p\mathrm{ && p < cell_count && 0 <= q && q < cell_count)
                        for (cp = partition_get(part, p, q); cp != NULL; cp = cp->next) {
                                d = distance(elements + n, elements + cp->index);
                        if (d < radius && !check_neighbors(elements, n, cp->index))
                        case1_count++;
                        else if (d >= radius && check_nhood(elements, n, cp->index))
                        case2_count++;
                        }
                partition_add(part, i, j, n);
        }
    fprintf(fp, "Number of missing neighbors = %d ", case1_count);
    fprintf(fp, "(average = %f)\n", (float)case1_count/size);
    fprintf(fp, "Number of illegal neighbors = %d ", case2_count);
    fprintf(fp, "(average = %f)\n", (float)case2_count/size);
    free_partition(part);
}
int check_nhood(node *elements, int i, int j)
{
    cell *cp;
    for (cp = elements[i].neighbors; cp != NULL; cp = cp->next)
        if (cp->index == j) {
            for (cp = elements[j].neighbors; cp != NULL; cp = cp->next)
                if (cp->index == i)
                    return 1;
            break;
        }
    return 0;
}
```

```
int check_neighbors(node *elements, int i, int j)
```

int check_neighbors(node *elements, int i, int j)
{
{
cell *cp;

```
    cell *cp;
```

```
    for (cp = elements[i].neighbors; cp != NULL; cp = cp->next)
    if (cp->index == j || check_nhood(elements, cp->index, j)) {
            for (cp = elements[j].neighbors; cp != NULL; cp = cp->next)
                if (cp->index == i || check_nhood(elements, cp->index, i))
                    return 1;
        break;
        }
    return 0;
}
```


## C.3.11 circle.c

```
/**************
* circle.c *
***************
```

Input: A file of gunk, such as the output of mkbd or mkgrid.
Output: The result of attempting to solve Laplace's equation
using approximations of line integrals. Pretty bad.
*/
\#include <stdio.h>
\#include <math.h>
\#include "gunk.h"
\#include "random.h"
double circle(int, node *, unsigned long);
main(int argc, char *argv[])
\{
int size;
unsigned long count;
FILE *fp;
node *elements;
if (argc $!=4$ ) \{
fprintf(stderr, "Usage: \%s input-file number-of-iterations output-file\n",
$\operatorname{argv}[0]$ ) ;
exit(1);
$\}$
$f p=$ fopen(argv[1], "r");
sscanf(argv[2], "\%u", \&count);
if (fp == NULL) \{
fprintf(stderr, "Error: Cannot open file \"\%s\"\n", argv[1]);
exit(2);
$\}$
fscanf(fp, "\%d", \&size);
elements $=$ (node *)calloc(sizeof(node), size);
fprintf(stderr, "Reading input file (\% nodes)... ln ", size);
read_elements (fp, size, elements);

```
    fclose(fp);
    fprintf(stderr, "Running %u iterations...\n", count);
    fprintf(stderr, "Maximum residual = %.16lf\n",
                circle(size, elements, count));
    fp = fopen(argv[3], "w");
    fprintf(stderr, "Writing output file...\n");
    write_elements(fp, size, elements);
    fclose(fp);
    fprintf(stderr, "Freeing memory...\n");
    free_memory(size, elements);
    return 0;
}
double circle(int size, node *elements, unsigned long count)
{
    cell *cp, *beginning;
    int n, i, length, index;
    double sum, residual, max_residual;
    initialize_random();
    for (n = 0; n < count; n++) {
        max_residual = 0.0;
        for (i = 0; i < size; i++) {
            index = lrandom()%size;
            if (interior_node(elements[index])) {
            beginning = elements[index].neighbors;
            sum = 0.0;
            length = 0;
            for (cp = beginning; cp->next != NULL; cp = cp->next) {
                sum += (elements[cp->index].z + elements[(cp->next)->index].z)/2.0*
                distance(elements + cp->index, elements + (cp->next)->index);
                length++;
            }
            sum += (elements[cp->index].z + elements[beginning->index].z)/2.0*
                distance(elements + cp->index, elements + beginning->index);
            residual = sum - elements[index].z;
            elements[index].z = sum;
            if (fabs(residual) > max_residual)
                max_residual = fabs(residual);
        }
        }
    }
    return max_residual;
}
```


## C.3.12 dot.c

```
/***********
* dot.c *
***********
Let's test the inner product routine in norm_eqs.c.
*/
#include <stdio.h>
#include <math.h>
#include "gunk.h"
#include "farray.h"
void read_coeff(FILE *, int, farray **);
void write_coeff(FILE *, int, farray **);
void free_coeff(int, farray **);
void dot(int, node *, farray **, int, int);
main(int argc, char *argv[])
{
    farray **coeff;
    int size, i, j;
    node *elements;
    FILE *fp;
    if (argc != 5) {
        fprintf(stderr, "Usage: %s gunk coeff i j\n", argv[0]);
        exit(1);
    }
    fp = fopen(argv[1], "r");
    if (fp == NULL) {
        fprintf(stderr, "Error: Cannot open file \"%s\"\n", argv[1]);
        exit(2);
    }
    fscanf(fp, "%d", &size);
    elements = (node *)calloc(sizeof(node), size);
    fprintf(stderr, "Reading input file (%d nodes)...\n", size);
    read_elements(fp, size, elements);
    fclose(fp);
    fp = fopen(argv[2], "r");
    if (fp == NULL) {
        fprintf(stderr, "Error: Cannot open file \"%s\"\n", argv[2]);
        exit(3);
    }
    fprintf(stderr, "Reading coefficients...\n");
    coeff = (farray **)calloc(sizeof(farray *), size);
    read_coeff(fp, size, coeff);
    fclose(fp);
```

```
    /* Generate new coefficients and neighborhoods. */
    sscanf(argv[3], "%d", &i);
    sscanf(argv[4], "%d", &j);
    dot(size, elements, coeff, i, j);
    fprintf(stderr, "Freeing gunk memory...\n");
    free_memory(size, elements);
    fprintf(stderr, "Freeing coefficient memory...\n");
    free_coeff(size, coeff);
    return 0;
}
void free_coeff(int size, farray **coeff)
{
    int i;
    for (i = 0; i < size; i++)
        farray_free(coeff[i]);
    free(coeff);
}
void write_coeff(FILE *fp, int size, farray **coeff)
{
    int i, j, N;
    for (i = 0; i < size; i++) {
        N = fa_size(coeff[i]);
        putint(fp, N);
        for (j = 0; j < N; j++)
            putlf(fp, fa_ref(coeff[i], j));
    }
}
void read_coeff(FILE *fp, int size, farray **coeff)
{
    double val;
    int i, j, N;
    for (i = 0; i < size; i++) {
            getint(fp, &N);
        if (N > 0) {
            coeff[i] = farray_cons(N);
            getlf(fp, &val);
            fa_set(coeff[i], 0, val);
            for (j = 1; j < N; j++) {
                getlf(fp, &val);
                fa_set(coeff[i], j, val);
            }
        }
    }
}
```

```
/**
    * Generate normal equations for the system of equations generated by
    * gen_coeff. This has the distinct advantage that the result matrix is
    * symmetric positive-definite.
    *
    * The current implementation is *wrong*. It actually gets the transpose of
    * the normal equations, and it doesn't deal properly with boundaries (it's
    * hard to deal with boundaries using these data structures).
    **/
void dot(int size, node *elements, farray **coeff, int p, int q)
{
    cell **new_neighbors, *cp, *cq, *new, *prev, head;
    double val;
    farray **new_coeff, *temp;
    int count, i, j, k;
    double inner_product(int, int, node *, farray **);
    /* First step: For each node, compute the union of its neighbor-set and its
    * neighbors' neighbor-sets. */
    fprintf(stderr, "Computing and sorting neighbor sets...\n");
    new_neighbors = (cell **)calloc(sizeof(cell *), size);
    new_coeff = (farray **)calloc(sizeof(farray *), size);
    head.index = -1;
    /* We need this array to sort the old lists. */
    for (i = 0; i < size; i++) {
        /* Need some temporary space for sorting coefficients. */
        if (coeff[i] != NULL) {
        temp = farray_cons(fa_size(coeff[i]));
        fa_set(temp, 0, fa_ref(coeff[i], 0));
        }
        /* Copy neighbors first, taking care to sort the new list. */
        head.next = NULL;
        for (j = 0, cp = elements[i].neighbors; cp != NULL; j++, cp = cp->next) {
        if (coeff[i] != NULL)
            val = fa_ref(coeff[i], j);
        for (prev = &head, cq = prev; cq != NULL; cq = cq->next)
            if (cq->index > cp->index)
                break;
            else
                        prev = cq;
        if (coeff[i] != NULL)
            fa_set(temp, k, val);
        new = (cell *)malloc(sizeof(cell));
        new->next = prev->next;
```

```
        prev->next = new;
        new->index = cp->index;
        }
        /* Sort old list, too. */
        for (j = 0, cp = elements[i].neighbors, cq = head.next;
            cp != NULL;
            j++, cp = cp->next, cq = cq->next)
        cp->index = cq->index;
    if (coeff[i] != NULL) {
        farray_free(coeff[i]);
        coeff[i] = temp;
    }
    /* Then copy neighbors' neighbors, still sorting. */
        for (cp = elements[i].neighbors; cp != NULL; cp = cp->next)
        for (cq = elements[cp->index].neighbors; cq != NULL; cq = cq->next)
            if (cq->index != i) {
            for (prev = &head, new = prev; new != NULL; new = new->next)
                if (new->index > cq->index)
                    break;
                else
                    prev = new;
            if (prev->index != cq->index) {
                new = (cell *)malloc(sizeof(cell));
                new->next = prev->next;
                prev->next = new;
                new->index = cq->index;
            }
        }
        new_neighbors[i] = head.next;
    }
    printf("The dot product of columns %d and %d is %.16lf.\n",
            p, q, inner_product(p, q, elements, coeff));
    /* Free temporary memory. */
    free(new_neighbors);
    free(new_coeff);
}
/**
* This procedure assumes that the neighbor lists have been sorted in
* descending order by index. This implementation does *not* do the right
* thing with boundary nodes: The inner product between an interior and a
* boundary node is automatically zero, and no boundary condition comes in.
*
* But why should this introduce instability into the system? *Which* matrix
* needs to be positive-definite?
**/
double inner_product(int i, int j, node *elements, farray **coeff)
{
```

```
cell *cp = elements[i].neighbors, *cq = elements[j].neighbors, *cr;
double sum = 0, prod;
int count, k, l;
while (1)
    if (cp == NULL && cq == NULL) {
        if (i == j && interior_node(elements + i))
            sum += fa_ref(coeff[i], 0)*fa_ref(coeff[j], 0);
        return sum;
    }
    else if (cq == NULL || (cp != NULL && cp->index < cq->index)) {
        if (cp->index == j && interior_node(elements + j)) {
            prod = fa_ref(coeff[j], 0);
            for (l = 1, cr = elements[j].neighbors; cr != NULL; l++, cr = cr->next)
                if (cr->index == i) {
                    prod *= fa_ref(coeff[j], l);
                    break;
            }
        sum += prod;
        }
        cp = cp->next;
    }
    else if (cp == NULL || (cq != NULL && cp->index > cq->index)) {
        if (cq->index == i && interior_node(elements + i)) {
        prod = fa_ref(coeff[i], 0);
        for (l = 1, cr = elements[i].neighbors; cr != NULL; l++, cr = cr->next)
            if (cr->index == j) {
                    prod *= fa_ref(coeff[i], l);
                    break;
            }
        sum += prod;
        }
        cq = cq->next;
    }
    else {
        k = cp->index;
        if (interior_node(elements + k)) {
        prod = 1.0;
        for (count = 2, l = 1, cr = elements[k].neighbors;
                cr != NULL && count > 0;
                l++, cr = cr->next) {
            if (cr->index == i) {
                prod *= fa_ref(coeff[k], l);
                count--;
            }
            if (cr->index == j) {
                prod *= fa_ref(coeff[k], 1);
                count--;
```

```
            }
            }
        sum += prod;
        }
        cp = cp->next;
        cq = cq->next;
    }
}
```


## C.3.13 estimate.c

```
/*****************
* estimate.c *
******************
Input: A file of gunk, such as the output of mkbd or mkgrid.
Given the coordinates of the boundary nodes, this program attempts to
establish a crude coordinate system using relaxation.
*/
#include <stdio.h>
#include <math.h>
#include "gunk.h"
#include "random.h"
const double omega = 1.9390892311242500;
double estimate(int, node *, int);
main(int argc, char *argv [])
{
    int size, count;
    FILE *fp;
    node *elements;
    if (argc != 4) {
        fprintf(stderr, "Usage: %s input-file number-of-iterations output-file\n",
            argv[0]);
        exit(1);
    }
    fp = fopen(argv[1], "r");
    sscanf(argv[2], "%d", &count);
    if (fp == NULL) {
        fprintf(stderr, "Error: Cannot open file \"%s\"\n", argv[1]);
        exit(2);
    }
    fscanf(fp, "%d", &size);
    elements = (node *)calloc(sizeof(node), size);
```

```
    fprintf(stderr, "Reading input file (%d nodes)...\n", size);
    read_elements(fp, size, elements);
    fclose(fp);
    fprintf(stderr, "Running %d iterations...\n", count);
    fprintf(stderr, "Maximum residual = %.16lf\n",
        estimate(size, elements, count));
    fp = fopen(argv[3], "w");
    fprintf(stderr, "Writing output file...\n");
    write_elements(fp, size, elements);
    fclose(fp);
    fprintf(stderr, "Freeing memory...\n");
    free_memory(size, elements);
    return 0;
}
double estimate(int size, node *elements, int count)
{
    cell *cp;
    int n, i, length, index;
    double sum_x, sum_y, residual_x, residual_y, max_residual;
    initialize_random();
    for (n = 0; n < count; n++) {
        max_residual = 0.0;
        for (i = 0; i < size; i++)
            if (interior_node(elements + i)) {
                sum_x = sum_y = 0.0;
            length = 0;
                index = lrandom()%size;
                for (cp = elements[index].neighbors; cp != NULL; cp = cp->next) {
                sum_x += elements[cp->index].x;
                sum_y += elements[cp->index].y;
                length++;
            }
                if (length > 0) {
                    residual_x = sum_x - length*elements[index].x;
                        residual_y = sum_y - length*elements[index].y;
                elements[index].x += omega*residual_x/length;
                elements[index].y += omega*residual_y/length;
                        if (fabs(residual_x) > max_residual)
                max_residual = fabs(residual_x);
                if (fabs(residual_y) > max_residual)
                        max_residual = fabs(residual_y);
            }
        }
    }
    return max_residual;
```


## C.3.14 farray.c

```
/**************
* farray.c *
***************
This file defines the data type farray, which is an array/vector
of double-precision floating point numbers.
*/
```

```
#include <stdio.h>
```

\#include <stdio.h>
\#include "farray.h"
\#include "farray.h"
farray *farray_cons(int size)
farray *farray_cons(int size)
{
{
farray *fap;
farray *fap;
if (size <= 0) {
if (size <= 0) {
fprintf(stderr, "Error: Illegal array size\n");
fprintf(stderr, "Error: Illegal array size\n");
exit(1);
exit(1);
}
}
fap = (farray *)malloc(sizeof(farray));
fap = (farray *)malloc(sizeof(farray));
fap->size = size;
fap->size = size;
fap->array = (double *)calloc(sizeof(double), size + 1);
fap->array = (double *)calloc(sizeof(double), size + 1);
return fap;
return fap;
}
}
void farray_free(farray *fap)
void farray_free(farray *fap)
{
{
if (fap != NULL) {
if (fap != NULL) {
if (fap->array != NULL)
if (fap->array != NULL)
free(fap->array);
free(fap->array);
free(fap);
free(fap);
}
}
}
}
double fa_ref(farray *fap, int i)
double fa_ref(farray *fap, int i)
{
{
if (i < 0 || fap->size <= i) {
if (i < 0 || fap->size <= i) {
fprintf(stderr, "Error: Array access out of bound (%d/%d; action = ref)\n",
fprintf(stderr, "Error: Array access out of bound (%d/%d; action = ref)\n",
i, fap->size);
i, fap->size);
exit(1);
exit(1);
}
}
return *(fap->array + i + 1);
return *(fap->array + i + 1);
}
}
void fa_set(farray *fap, int i, double val)
void fa_set(farray *fap, int i, double val)
{
{
if (i < 0 || fap->size <= i) {

```
    if (i < 0 || fap->size <= i) {
```

```
        fprintf(stderr, "Error: Array access out of bound (%d/%d; action = set)\n",
                    i, fap->size);
        exit(1);
    }
    *(fap->array + i + 1) = val;
}
int fa_size(farray *fap)
{
    return fap->size;
}
```


## C.3.15 femcompact.c

```
/*********************
    * femcompact.c *
    *******************
```

Given a list of tuples of the form ( $x, y$, val, ref), remove all tuples of the form ( $0,0,0,0$ ).
*/
\#include <stdio.h>
void femcompact (FILE *, FILE *);
main(int argc, char *argv[])
\{
FILE *fin;
if (argc $!=2$ ) \{
fprintf(stderr, "Usage: \%s input-file\n", argv[0]); exit(1);
\}
/* Open input file. */
fin $=$ fopen $(\operatorname{argv}[1], \quad " r ")$;
if (fin == NULL) \{
fprintf(stderr, "Error: Cannot read file \"\%s\".\n", argv[1]); exit(2);
\}
getc (fin); /* Pop the first character in the file. */
femcompact(fin, stdout);
fclose(fin);
return 0 ;
\}
void femcompact(FILE *fin, FILE *fout)
$\{$
double $x, y, v a l$, ref;

```
    fprintf(fout, "\n");
    while (fscanf(fin, "%lg\t%lg\t%lg\t%lg\n", &x, &y, &val, &ref) != EOF)
    if (! (x == 0 && y == 0 && ref == 0 && val == 0))
        fprintf(fout, "%.16lg\t%.16lg\t%.16lg\t%.16lg\n", x, y, val, ref);
}
```


## C.3.16 femstats.c

```
/*****************
    * femstats.c *
    ****************
```

Given a list of tuples of the form ( $x, y$, val, ref), compute statistics.
*/
\#include <stdio.h>
\#include <math.h>
void femstats(FILE *, FILE *);
main(int arge, char *argv[])
$\{$
FILE *fin;
if (argc != 2) \{
fprintf(stderr, "Usage: \%s input-file\n", argv[0]);
exit(1);
\}
/* Open input file. */
fin $=$ fopen(argv[1], "r");
if (fin == NULL) \{
fprintf(stderr, "Error: Cannot read file \"\%s\".\n", argv[1]);
exit(2) ;
\}
getc(fin); /* Pop the first character in the file. */
femstats(fin, stdout);
fclose(fin);
return 0 ;
\}
void femstats(FILE *fin, FILE *fout)
$\{$
double $x, y$, val, ref, max_abs, min_abs, sum_abs, max_rel, min_rel, abs, rel;
int count = 1;
fscanf(fin, " $\% 1 g \backslash t \% 1 g \backslash t \% 1 g \backslash t \% 1 g \backslash n ", \& x, \& y, \& v a l, \& r e f) ;$
max_abs = min_abs = sum_abs = fabs(val - ref);
max_rel = min_rel $=($ ref $==0.0$ ) ? 0.0 : (val - ref)/ref;
while (fscanf(fin, "\%1g\t\%lg\t\% $1 \mathrm{~g} \backslash t \% 1 g \backslash n ", \& x, \& y, \& v a l, \& r e f) \quad!=E O F)\{$

```
        abs = fabs(val - ref);
        rel = (ref == 0.0) ? 0.0 : (val - ref)/ref;
        count++;
        sum_abs += abs;
        if (abs > max_abs)
        max_abs = abs;
        else if (abs < min_abs)
        min_abs = abs;
        if (rel > max_rel)
        max_rel = rel;
        else if (rel < min_rel)
        min_rel = rel;
    }
    fprintf(fout, "Maximum absolute error: %.16lg\n", max_abs);
    fprintf(fout, "Minimum absolute error: %.16lg\n", min_abs);
    fprintf(fout, "Average absolute error: %.16lg\n\n", sum_abs/count);
    fprintf(fout, "Maximum relative error: %.16lg\n", max_rel);
    fprintf(fout, "Minimum relative error: %.16lg\n", min_rel);
}
```


## C.3.17 fill.c

```
/***********
* fill.c *
************
```

This program appears to be a test of the pseudorandom number generator, which was ported directly from MIT Scheme.
*/

```
#include <stdio.h>
#include "random.h"
#include <math.h>
#include "gunk.h"
#include <time.h>
#include <stdlib.h>
main(int argc, char *argv[])
{
    double end, mean, stddev = 0.0;
    int size, count = 0, n, index, max, *list, i;
    if ((argc != 2) && (argc != 3)) {
        fprintf(stderr, "Usage: %s sample-size [percentage]\n", argv[0]);
        exit(1);
    }
    sscanf(argv[1], "%d", &size);
    if (argc == 3)
        sscanf(argv[2], "%lf", &end);
```

```
    else
        end = 1.0;
/* initialize_random();*/
    list = (int *)calloc(sizeof(int), size);
    if (end > 1.0)
        end = 1.0;
    end *= size;
    for (n = 0; count < end; n++) {
        index = (int)(frandom()*size);
        if (list[index] == 0)
            count++;
        if (list[index] == max)
            max = ++(list[index]);
        else
            (list[index])++;
    }
    mean = (double)n/size;
    for (i = 0; i < size; i++)
        stddev += fsquare(list[i] - mean);
    stddev = sqrt(stddev/size);
    printf("Took %d samples to fill %lf%% of %d slots;\n",
            n, 100*end/size, size);
    printf("on the average %.16lf per slot.\n", mean);
    printf("Maximum is %d.\n", max);
    printf("Standard deviation = %.16lf\n", stddev);
    free(list);
    return 0;
}
```


## C.3.18 filt.c

```
/************
* filt.c *
*************
```

This program seems to initialize the nodes according to some
odd criterion. Can't recall writing this for any good reason.
*/
\#include <stdio.h>
\#include "gunk.h"
void filter(int, node *, double);

```
main(int argc, char *argv[])
{
    double threshold;
    int size;
    node *elements;
    FILE *fp;
    if (argc != 4) {
        fprintf(stderr, "Usage: %s input-file threshold output-file\n", argv[0]);
        exit(1);
    }
    fp = fopen(argv[1], "r");
    sscanf(argv[2], "%lf", &threshold);
    if (fp == NULL) {
        fprintf(stderr, "Error: Cannot open file \"%s\"\n", argv[1]);
        exit(2);
    }
    fscanf(fp, "%d", &size);
    fprintf(stderr, "Reading file (%d nodes)...\n", size);
    elements = (node *)calloc(sizeof(node), size);
    read_elements(fp, size, elements);
    fclose(fp);
    fprintf(stderr, "Computing output...\n");
    filter(size, elements, threshold);
    fprintf(stderr, "Writing output...\n");
    fp = fopen(argv[3], "w");
    write_elements(fp, size, elements);
    fclose(fp);
    free_memory(size, elements);
    return 0;
}
void filter(int size, node *elements, double threshold)
{
    int i;
    node corner1, corner2;
    corner1.x = 0.0;
    corner1.y = 1.0;
    corner2.x = 1.0;
    corner2.y = 1.0;
    for (i = 0; i < size; i++)
        if (interior_node(elements + i) &&
            (distance(elements + i, &corner1) < threshold ||
            distance(elements + i, &corner2) < threshold))
            elements[i].z = potential(elements[i].x, elements[i].y);
}
```


## C.3.19 grad.c

```
/************
* grad.c *
*************
This program attempts to estimate the magnitude of the gradient at
each sample point, I think.
*/
#include <stdio.h>
#include "gunk.h"
void grad(int, node *);
main(int argc, char *argv[])
{
    int size;
    node *elements;
    FILE *fp;
    if (argc != 3) {
        fprintf(stderr, "Usage: %s input-file output-file\n", argv[0]);
        exit(1);
    }
    fp = fopen(argv[1], "r");
    if (fp == NULL) {
        fprintf(stderr, "Error: Cannot open file \"%s\"\n", argv[1]);
        exit(2);
    }
    fscanf(fp, "%d", &size);
    fprintf(stderr, "Reading file (%d nodes)...\n", size);
    elements = (node *)calloc(sizeof(node), size);
    read_elements(fp, size, elements);
    fclose(fp);
    fprintf(stderr, "Computing output...\n");
    grad(size, elements);
    fprintf(stderr, "Writing output file...\n");
    fp = fopen(argv[2], "w");
    write_elements(fp, size, elements);
    fclose(fp);
    free_memory(size, elements);
    return 0;
}
void grad(int size, node *elements)
{
    cell *cp;
    double dist, *grad;
    int i, j;
```

```
    grad = (double *)calloc(sizeof(double), size);
    for (i = 0; i < size; i++) {
        j = i;
        for (cp = elements[i].neighbors; cp != NULL; cp = cp->next)
            if (elements[cp->index].z > elements[j].z)
                j = cp->index;
        dist = distance(elements + i, elements + j);
        grad[i] = (dist > 0) ? (elements[j].z - elements[i].z)/dist : 0.0;
    }
    for (i = 0; i < size; i++)
    elements[i].z = grad[i];
    free(grad);
}
```


## C.3.20 gs_mat.c

```
/**************
* gs_mat.c *
**************
```

Generate the matrix actually used in Gauss-Seidel iteration. We can then use MATLAB or something like that to analyze the damn thing. Won't work for large systems due to memory constraints.

```
*/
```

```
#include <stdio.h>
#include <math.h>
#include "gunk.h"
#include "matrix.h"
void gs_mat(FILE *, int, node *, farray **, double);
main(int argc, char *argv[])
{
    double val, omega = 1.0;
    farray **coeff;
    int size, i, j, N, count = 0;
    node *elements;
    FILE *fp;
    if (argc != 4 && argc != 5) {
        fprintf(stderr, "Usage: %s input-file coeff output-file [omega]\n",
            argv[0]);
        exit(1);
    }
    fp = fopen(argv[1], "r");
```

```
    if (fp == NULL) {
    fprintf(stderr, "Error: Cannot open file \"%s\"\n", argv[1]);
    exit(1);
}
fscanf(fp, "%d", &size);
fprintf(stderr, "Reading input (%d nodes)...\n", size);
elements = (node *)calloc(sizeof(node), size);
read_elements(fp, size, elements);
close(fp);
fp = fopen(argv[2], "r");
if (fp == NULL) {
    fprintf(stderr, "Error: Cannot open file \"%s\"\n", argv[2]);
    exit(1);
}
fprintf(stderr, "Reading coefficients...\n");
coeff = (farray **)calloc(sizeof(farray *), size);
for (i = 0; i < size; i++) {
    getint(fp, &N);
    if (N > 0) {
        count++;
        coeff[i] = farray_cons(N);
        for (j = 0; j < N; j++) {
            getlf(fp, &val);
            fa_set(coeff[i], j, val);
        }
    }
}
fclose(fp);
fprintf(stderr, "Composing %dx%d Gauss-Seidel iteration matrix...\n",
                count, count);
    if (argc == 5)
    sscanf(argv[4], "%lf", &omega);
fp = fopen(argv[3], "w");
gs_mat(fp, size, elements, coeff, omega);
fclose(fp);
    fprintf(stderr, "Freeing memory...\n");
free_memory(size, elements);
    for (i = 0; i < size; i++)
    farray_free(coeff[i]);
    free(coeff);
    return 0;
}
void gs_mat(FILE *fp, int size, node *elements, farray **coeff, double omega)
{
```

```
cell *cp;
int real_size = 0, i, j, *index, count;
matrix *mat, *big_mat;
for (i = 0; i < size; i++)
    if (interior_node(elements + i))
        real_size++;
big_mat = matrix_cons(real_size, 2*real_size);
index = (int *)calloc(sizeof(int), size);
count = 0;
for (i = 0; i < size; i++)
    if (interior_node(elements + i))
        index[i] = count++;
    else
        index[i] = -1;
/* Construct the original matrix to be inverted: */
for (i = 0; i < size; i++)
    if (interior_node(elements + i) && coeff[i] != NULL) {
        for (j = 1, cp = elements[i].neighbors;
                        cp != NULL;
                        j++, cp = cp->next)
                if (coeff[cp->index] != NULL)
                    mset(big_mat, index[i], index[cp->index], fa_ref(coeff[i], j));
        mset(big_mat, index[i], index[i], fa_ref(coeff[i], 0));
    }
/* Free up space as we go along. */
free(index);
/* Collect the diagonal and lower-triangular entries, and multiply the upper
    triangular part by the overrelaxation factor. */
for (i = 0; i < real_size; i++)
    for (j = i + 1; j < real_size; j++) {
        mset(big_mat, i, j + real_size, -omega*mref(big_mat, i, j));
        mset(big_mat, i, j, 0);
    }
rref(big_mat);
mat = matrix_cons(real_size, real_size);
for (i = 0; i < real_size; i++)
    for (j = 0; j < real_size; j++)
        mset(mat, i, j, mref(big_mat, i, j + real_size));
matrix_free(big_mat);
for (i = 0; i < real_size; i++)
    mset(mat, i, i, 1 - omega + mref(mat, i, i));
mprint(fp, mat);
matrix_free(mat);
```


## \}

## C.3.21 gsort.c

```
/**************
* gsort.c *
**************
```

Sort gunk, in the hope of reducing bandwidth.

```
*/
```

```
#include <stdio.h>
#include <math.h>
#include "gunk.h"
void sort_nodes(int, node *);
int compare_nodes(int, int, node *);
void sort(int *, int, int, node *);
int sort_part(int *, int, int, node *);
main(int argc, char *argv[])
{
    FILE *fp;
    int size;
    node *elements;
    if (argc != 3) {
        fprintf(stderr, "Usage: %s input-file output-file\n", argv[0]);
        exit(1);
    }
```

    \(f p=\) fopen(argv[1], "r");
    if ( \(f p==\mathrm{NULL}\) ) \(\{\)
        fprintf(stderr, "Error: Cannot open file \"\%s\"\n", argv[1]);
        exit(1);
    \(\}\)
    fscanf(fp, "\% ", \&size);
    fprintf(stderr, "Reading input (\% nodes)...\n", size);
    elements = (node *)calloc(sizeof(node), size);
    read_elements (fp, size, elements);
    fclose(fp);
    fprintf(stderr, "Sorting gunk...\n");
    sort_nodes(size, elements);
    \(f p=f o p e n(\operatorname{argv}[2], \quad\) "r");
    fprintf(stderr, "Writing output...\n");
    write_elements(fp, size, elements);
    fclose(fp);
    fprintf(stderr, "Freeing memory...\n");
    free_memory(size, elements);
    ```
    return 0;
}
void sort_nodes(int size, node *elements)
{
    cell *cp;
    int index[size], invert[size], i;
    node temp[size];
    for (i = 0; i < size; i++)
        invert[i] = i;
    sort(invert, 0, size - 1, elements);
    for (i = 0; i < size; i++)
        index[invert[i]] = i;
    for (i = 0; i < size; i++) {
        temp[i] = elements[i];
        for (cp = elements[i].neighbors; cp != NULL; cp = cp->next)
            cp->index = index[cp->index];
    }
    for (i = 0; i < size; i++)
        elements[index[i]] = temp[i];
}
void sort(int *index, int p, int r, node *elements)
{
    int q;
    if (p < r) {
        q = sort_part(index, p, r, elements);
        sort(index, p, q, elements);
        sort(index, q + 1, r, elements);
    }
}
int sort_part(int *index, int p, int r, node *elements)
{
    int x = index[p], i = p - 1, j = r + 1, temp;
    while (1) {
        do {j--;} while (compare_nodes(index[j], x, elements) > 0);
        do {i++;} while (compare_nodes(index[i], x, elements) < 0);
        if (i< j) {
            temp = index[i];
            index[i] = index[j];
            index[j] = temp;
        }
        else return j;
    }
}
int compare_nodes(int n1, int n2, node *elements)
{
    if (elements[n1].neighbors == NULL && elements[n2].neighbors != NULL)
```

```
        return -1;
    else if (elements[n1].neighbors != NULL && elements[n2].neighbors == NULL)
        return 1;
    else if (elements[n1].y > elements[n2].y)
        return 1;
    else if (elements[n1].y < elements[n2].y)
        return -1;
    else if (elements[n1].x > elements[n2].x)
        return 1;
    else if (elements[n1].x < elements[n2].x)
        return -1;
    else
        return 0;
}
```


## C.3.22 gunk.c

```
/************
* gunk.c *
*************
```

Common routines used by gunk programs, including read/write, etc.
*/
\#include <stdio.h>
\#include <math.h>
\#include "gunk.h"
\#define SERIES_LIMIT 1000
\#define POT2
/**
* Some artifacts, left here just so some old programs would compile (I'm not
* sure what those programs are good for, actually).
**/
void old_read_elements (FILE *fp, int size, node *elements)
f
cell **cp;
int $n$, index;
for ( $n=0 ; n<s i z e ; n++$ ) \{
fscanf(fp, "\%lf", \&(elements[n].x));
fscanf(fp, "\%lf", \&(elements[n].y));
fscanf(fp, " $\% 1 \mathrm{fl}$ ", \&(elements[n].z));
$\mathrm{cp}=2$ (elements [n] .neighbors);
*cp = NULL;
while ((index = read_neighbor $(f p)$ ) >= 0) \{
*cp $=$ (cell *) malloc (sizeof (cell));
(*cp)->index = index;
$c p=\&((* c p)->$ next $)$;
*CP = NULL;
\}

```
    }
}
int read_neighbor(FILE *fp)
{
    char s[STRLEN], c;
    int n = 0;
    while (1) {
        fscanf(fp, "%c", &c);
        if (c == 'x')
            return -1;
        else if (c != , ') {
            while (c != , ') {
                s[n++] = c;
                fscanf(fp, "%c", &c);
            }
            s[n] = '\0';
            sscanf(s, "%d", &n);
            return n;
        }
    }
}
void old_write_elements(FILE *fp, int size, node *elements)
{
    cell *cp;
    int n;
    fprintf(fp, "%d\n", size);
    for (n = 0; n < size; n++) {
        fprintf(fp, "%.17lg %.17lg %.17lg ",
                elements[n].x, elements[n].y, elements[n].z);
            cp = elements[n].neighbors;
            while (cp != NULL) {
            fprintf(fp, "%d ", cp->index);
            cp = cp->next;
        }
        fprintf(fp, "x\n");
    }
}
/**
    * A real mess, these file formats.
    **/
void write_elements(FILE *fp, int size, node *elements)
{
    cell *cp;
    int n, count;
    fprintf(fp, "%d\n", size);
```

```
    fprintf(fp, "Gunk file version %d\n", GUNK_VERSION);
    for (n = 0; n < size; n++) {
        putlf(fp, elements[n].x);
        putlf(fp, elements[n].y);
        putlf(fp, elements[n].z);
        putint(fp, elements[n].type);
        count = 0;
        for (cp = elements[n].neighbors; cp != NULL; cp = cp->next)
            count++;
        putint(fp, count);
        for (cp = elements[n].neighbors; cp != NULL; cp = cp->next)
            putint(fp, cp->index);
    }
}
void read_elements(FILE *fp, int size, node *elements)
{
    cell **cp;
    int n, count, i, version, return_val;
    return_val = fscanf(fp, "\nGunk file version %d\n", &version);
    if (return_val == EOF || return_val == 0) {
        fprintf(stderr, "Error: Not a gunk file!\n");
        exit(1);
    }
    for (n = 0; n < size; n++) {
        getlf(fp, &(elements[n].x));
        getlf(fp, &(elements[n].y));
        getlf(fp, &(elements[n].z));
        if (version >= 1)
            getint(fp, &(elements[n].type));
        getint(fp, &count);
        elements[n].neighbors = NULL;
        cp = &(elements[n].neighbors);
        for (i = 0; i < count; i++) {
            *cp = (cell *)malloc(sizeof(cell));
            getint(fp, &((*cp)->index));
            (*cp)->next = NULL;
            cp = &((*cp)->next);
        }
        if (version < 1)
            if (elements[n].neighbors == NULL)
                set_boundary_node(elements + n);
            else
                set_interior_node(elements + n);
    }
}
```

```
void putlf(FILE *fp, double x)
{
    int i;
    unsigned char *p = (unsigned char *)(&x);
    for (i = 0; i < sizeof(double); i++)
        putc(*(p++), fp);
}
void getlf(FILE *fp, double *x)
{
    int i;
    unsigned char *p = (unsigned char *)x;
    for (i = 0; i < sizeof(double); i++)
        *(p++) = (unsigned char)getc(fp);
}
void putint(FILE *fp, int n)
{
    int i;
    unsigned char *p = (unsigned char *)(&n);
    for (i = 0; i < sizeof(int); i++)
        putc(*(p++), fp);
}
void getint(FILE *fp, int *n)
{
    int i;
    unsigned char *p = (unsigned char *)n;
    for (i = 0; i < sizeof(int); i++)
        *(p++) = (unsigned char)getc(fp);
}
void free_memory(int size, node *elements)
{
    cell *cp, *cp_next;
    int i;
    for (i = 0; i < size; i++) {
        cp = elements[i].neighbors;
        while (cp != NULL) {
            cp_next = cp->next;
            free(cp);
            cp = cp_next;
        }
    }
    free(elements);
}
/********************************************
* This is the old potential; the new *
* one's simpler...
******************************************/
```

```
#ifdef POT1
double potential(double x, double y)
{
    double a = 1.0, b = 1.0, v = 1.0;
    double m2 = a*(-log(1e-18))/(2.0*pi);
    double m1 = m2/b;
    double terms[SERIES_LIMIT];
    int i, n;
    double new_term = 2*TOLERANCE, ratio, sum = 0.0;
    for (i = 0; (i < SERIES_LIMIT) && (fabs(new_term) >= TOLERANCE); i++) {
        n = 2*i + 1;
        if (n <= m1)
            ratio = sinh(pi*n*y/a)/sinh(pi*n*b/a);
        else if ( }n*y<=m2\mathrm{ )
            ratio = exp(pi*n*(y - b)/a) - exp(-pi*n*(y + b)/a);
        else
            ratio = exp(pi*n*(y - b)/a);
        new_term = 4*v*ratio*sin(pi*n*x/a)/(pi*n);
        terms[i] = new_term;
    }
    for (n = i - 1; n >= 0; n--)
        sum += terms[n];
    return 1.0 + sum;
}
#else
#ifdef POT2
/*********************************
* This is the nev potential: *
*********************************/
double potential(double x, double y)
{
    double a = 1.0, b = 1.0, A = 1.0/sinh(pi*b/a);
    return 1.0 + A*sinh(pi*y/a)*sin(pi*x/a);
}
/******************************************
* Let's try estimating the ordinate: *
******************************************/
#else
double potential(double x, double y)
{
    return y;
}
#endif
#endif
double fsquare(double x)
```

```
{
    return x*x;
}
double distance(node *node1, node *node2)
{
    return sqrt(fsquare(node1->x - node2->x) + fsquare(node1->y - node2->y));
}
void introduce(node *elements, int i, int j, double radius)
{
    cell *new_cell1, *new_cell2;
    if (distance(elements + i, elements + j) < radius) {
        new_cell1 = (cell *)malloc(sizeof(cell));
        new_cell2 = (cell *)malloc(sizeof(cell));
        new_cell1->next = elements[i].neighbors;
        new_cell2->next = elements[j].neighbors;
        elements[i].neighbors = new_cell1;
        elements[j].neighbors = new_cell2;
        new_cell1->index = j;
        new_cell2->index = i;
    }
}
partition *make_partition(int size)
{
    partition *new_partition = (partition *)malloc(sizeof(partition));
    new_partition->list = (cell **)calloc(sizeof(cell *), SQUARE(size));
    new_partition->size = size;
    return new_partition;
}
void partition_add(partition *partition, int i, int j, int index)
{
    cell **list = partition->list, *new_cell;
    int size = partition->size;
    if (i == size) i--;
    if (j == size) j--;
    if ((i >= size) || (j >= size)) {
        fprintf(stderr, "Error: Partition access out of bounds\n");
        exit(2);
    }
    new_cell = (cell *)malloc(sizeof(cell));
    new_cell->next = *(list + i*size + j);
    *(list + i*size + j) = new_cell;
    new_cell->index = index;
}
cell *partition_get(partition *partition, int i, int j)
{
```

```
    cell **list = partition->list;
    int size = partition->size;
    if (i == size) i--;
    if (j == size) j--;
    if ((i >= size) || (j >= size)) {
        fprintf(stderr, "Error: Partition access out of bounds\n");
        exit(3);
    }
    return *(list + i*size + j);
}
void free_partition(partition *partition)
{
    cell **list = partition->list, *cp, *cp_next;
    int size = partition->size, i, j;
    for (i = 0; i < size; i++) {
        for (j = 0; j < size; j++) {
            cp = *(list + i*size + j);
            while (cp != NULL) {
                cp_next = cp->next;
                    free(cp);
                    cp = cp_next;
            }
        }
    }
    free(partition);
}
int interior_node(node *n)
{
    if (n->type == INTERIOR_NODE)
        return 1;
    else
        return 0;
}
int boundary_node(node *n)
{
    if (n->type == BOUNDARY_NODE)
        return 1;
    else
        return 0;
}
void set_boundary_node(node *n)
{
    n->type = BOUNDARY_NODE;
}
void set_interior_node(node *n)
{
    n->type = INTERIOR_NODE;
}
```


## C.3.23 gunk2mat.c

```
/****************
* gunk2mat.c *
****************
Generate the matrix we're trying to invert. We can then use MATLAB or something
like that to analyze the damn thing. Won't work for large systems due to
memory constraints.
*/
#include <stdio.h>
#include <math.h>
#include "gunk.h"
#include "matrix.h"
void gunk2mat(FILE *, int, node *, farray **);
main(int argc, char *argv[])
{
    double val;
    farray **coeff;
    int size, i, j, N;
    node *elements;
    FILE *fp;
    if (argc != 4) {
        fprintf(stderr, "Usage: %s input-file coeff output-file\n", argv[0]);
        exit(1);
    }
    fp = fopen(argv[1], "r");
    if (fp == NULL) {
        fprintf(stderr, "Error: Cannot open file \"%s\"\n", argv[1]);
        exit(1);
    }
    fscanf(fp, "%d", &size);
    fprintf(stderr, "Reading input (%d nodes)...\n", size);
    elements = (node *)calloc(sizeof(node), size);
    read_elements(fp, size, elements);
    close(fp);
    fp = fopen(argv[2], "r");
    if (fp == NULL) {
        fprintf(stderr, "Error: Cannot open file \"%s\"\n", argv[2]);
        exit(1);
    }
    fprintf(stderr, "Reading coefficients...\n");
    coeff = (farray **)calloc(sizeof(farray *), size);
    for (i = 0; i < size; i++) {
```

```
        getint(fp,&N);
        if (N > 0) {
        coeff[i] = farray_cons(N);
        for (j = 0; j < N; j++) {
            getlf(fp, &val);
            fa_set(coeff[i], j, val);
        }
        }
    }
    fclose(fp);
    fp = fopen(argv[3], "w");
    gunk2mat(fp, size, elements, coeff);
    fclose(fp);
    fprintf(stderr, "Freeing memory...\n");
    free_memory(size, elements);
    for (i = 0; i < size; i++)
        farray_free(coeff[i]);
    free(coeff);
    return 0;
}
void gunk2mat(FILE *fp, int size, node *elements, farray **coeff)
{
    cell *cp;
    double boundary;
    int real_size = 0, i, j, *index, count;
    matrix *mat;
    for (i = 0; i < size; i++)
        if (interior_node(elements + i))
            real_size++;
    mat = matrix_cons(real_size, real_size + 1);
    index = (int *)calloc(sizeof(int), size);
    count = 0;
    for (i = 0; i < size; i++)
        if (interior_node(elements + i))
            index[i] = count++;
        else
            index[i] = -1;
    for (i = 0; i < size; i++)
        if (interior_node(elements + i) && coeff[i] != NULL) {
        boundary = 0.0;
            for (j = 1, cp = elements[i].neighbors;
                    cp != NULL;
                    j++, cp = cp->next)
            if (coeff[cp->index] != NULL)
                    mset(mat, index[i], index[cp->index], fa_ref(coeff[i], j));
            else
```

```
                boundary -= fa_ref(coeff[i], j)*elements[cp->index].z;
        mset(mat, index[i], index[i], fa_ref(coeff[i], 0));
        mset(mat, index[i], real_size, boundary);
        }
    mprint(fp, mat);
    matrix_free(mat);
    free(index);
}
```


## C.3.24 hilbert.c

```
/***************
* hilbert.c *
****************
Generate a Hilbert matrix for testing matrix routines.
*/
#include <stdio.h>
#include "matrix.h"
void init_hilbert(matrix *);
main(int argc, char *argv[])
{
    int i, n;
    farray *W;
    matrix *U, *V;
    if (argc != 2) {
        fprintf(stderr, "Usage: %s n\n", argv[0]);
        exit(1);
    }
    sscanf(argv[1], "%d", &n);
    U = matrix_cons(n, n);
    V = matrix_cons(n, n);
    W = farray_cons(n);
    init_hilbert(U);
    svdecomp(U, W, V);
    printf("Singular values for the %dx%d Hilbert matrix:\n", n, n);
    for (i = 0; i < n; i++)
        printf("%.16lg\n", fa_ref(W, i));
    printf("Condition number = %.16lg\n", fa_ref(W, 0)/fa_ref(W, n - 1));
    matrix_free(U);
    matrix_free(V);
    farray_free(W);
```

```
    return 0;
}
void init_hilbert(matrix *mat)
{
    int i, j, n = mat->nrows;
    if (n != mat->ncols) {
        fprintf(stderr, "Error: Cannot create non-square Hilbert matrix\n");
        exit(1);
    }
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
        mset(mat, i, j, 1.0/(2.0 + i + j));
}
```


## C.3.25 improve.c

## /**************

```
* improve.c *
```

***************

Input: A file of gunk, such as the output of mkbd or mkgrid, and a file of operator coefficients, such as the output of approx.

Output: An attempt to get a more accurate solution using the coefficients.
*/

```
#include <stdio.h>
#include <math.h>
#include "gunk.h"
#include "random.h"
#include "farray.h"
/* #define RANDOM */
const double mixture_ratio = 1.;
double improve(int, node *, int, farray **, double);
double sync(int, node *, int, farray **, double);
double laplace(int, node *, int, farray **);
double checksum(int, node *, int, farray **);
void read_coeff(FILE *, int, farray **);
main(int argc, char *argv[])
{
    double val, omega = 1.0;
    farray **coeff;
    int size, i, count;
    FILE *fp;
    node *elements;
    if (argc != 5 && argc != 6) {
```

```
    fprintf(stderr,
                    "Usage: %s input-file coeff iteration-count output-file [omega]\n",
                    argv[0]);
    exit(1);
    }
    fp = fopen(argv[1], "r");
    if (fp == NULL) {
    fprintf(stderr, "Error: Cannot open file \"%s\"\n", argv[1]);
    exit(2);
    }
    fscanf(fp, "%d", &size);
    elements = (node *)calloc(sizeof(node), size);
    fprintf(stderr, "Reading input file (%d nodes)...\n", size);
    read_elements(fp, size, elements);
    fclose(fp);
    fp = fopen(argv[2], "r");
    if (fp == NULL) {
        fprintf(stderr, "Error: Cannot open file \"%s\"\n", argv[2]);
        exit(1);
    }
    fprintf(stderr, "Reading coefficients...\n");
    coeff = (farray **)calloc(sizeof(farray *), size);
    read_coeff(fp, size, coeff);
    fclose(fp);
    sscanf(argv[3], "%d", &count);
    if (argc == 6) sscanf(argv[5], "%lf", &omega);
    fprintf(stderr, "Running %d iterations...\n", count);
    fprintf(stderr, "Maximum residual = %.16lf\n",
        improve(size, elements, count, coeff, omega));
    fp = fopen(argv[4], "ъ");
fprintf(stderr, "Writing output file...\n");
write_elements(fp, size, elements);
fclose(fp);
fprintf(stderr, "Freeing memory...\n");
free_memory(size, elements);
for (i = 0; i < size; i++)
        farray_free(coeff[i]);
    free(coeff);
    return 0;
}
double improve(int size, node *elements, int count, farray **coeff,
                    double omega)
{
    cell *cp;
    int n, i, j, index;
```

```
    double sum, max_residual;
#ifdef RANDOM
    initialize_random();
#endif
    for (n = 0; n < count; n++) {
        max_residual = 0.0;
        for (i = 0; i < size; i++) {
#ifdef RANDOM
            index = lrandom()%size;
#else
            index = i;
#endif
            if (interior_node(elements + index)) {
                sum = fa_ref(coeff[index], 0)*elements[index].z;
                for (cp = elements[index].neighbors, j = 1;
                    cp != NULL;
                    cp = cp->next, j++)
                sum += fa_ref(coeff[index], j)*elements[cp->index].z;
                elements[index].z -= omega*sum/fa_ref(coeff[index], 0);
                if (fabs(sum) > max_residual) max_residual = fabs(sum);
            }
        }
    }
    return max_residual;
}
double sync(int size, node *elements, int count, farray **coeff, double omega)
{
    cell *cp;
    int n, i, j;
    double sum, max_residual, *temp;
    temp = (double *)calloc(size, sizeof(double));
    for (i = 0; i < size; i++)
        temp[i] = elements[i].z;
    for (n = 0; n < count; n++) {
        max_residual = 0.0;
        for (i = 0; i < size; i++)
            if (interior_node(elements + i)) {
                sum = fa_ref(coeff[i], 0)*elements[i].z;
                for (cp = elements[i].neighbors, j = 1;
                        cp != NULL;
                            cp = cp->next, j++)
                sum += fa_ref(coeff[i], j)*elements[cp->index].z;
                temp[i] = elements[i].z - omega*sum/fa_ref(coeff[i], 0);
            }
```

```
        for (i = 0; i < size; i++)
            if (interior_node(elements + i)) {
            sum = fa_ref(coeff[i], 0)*temp[i];
                for (cp = elements[i].neighbors, j = 1;
                    cp != NULL;
                    cp = cp->next, j++)
                sum += fa_ref(coeff[i], j)*temp[cp->index];
            elements[i].z = temp[i] - omega*sum/fa_ref(coeff[i], 0);
            if (fabs(sum) > max_residual) max_residual = fabs(sum);
        }
    }
    free(temp);
    return max_residual;
}
void read_coeff(FILE *fp, int size, farray **coeff)
{
    double val;
    int i, j, N;
    for (i = 0; i < size; i++) {
        getint(fp, &N);
        if (N > 0) {
            coeff[i] = farray_cons(N);
            getlf(fp, &val);
            fa_set(coeff[i], 0, -(1 - mixture_ratio)*N + mixture_ratio*val);
            for (j = 1; j < N; j++) {
                getlf(fp, &val);
                fa_set(coeff[i], j, 1 - mixture_ratio + mixture_ratio*val);
            }
        }
    }
}
/*********************************************************************
* The following procedures exist for debugging purposes: *
* laplace() replaces the value at each node with an estimate of *
* the laplacian at that node, using the linear approximation. *
* checksum() computes the sum of the coefficients.
******************************************************************/
double laplace(int size, node *elements, int count, farray **coeff)
{
    cell *cp;
    int n, length, index;
    double sum, max_residual = 0.0;
    double *temp;
    temp = (double *)calloc(sizeof(double), size);
    for (index = 0; index < size; index++)
        if (interior_node(elements + index)) {
```

```
            length = 0;
            sum = fa_ref(coeff[index], 0)*elements[index].z;
            for (cp = elements[index].neighbors; cp != NULL; cp = cp->next)
                    sum += fa_ref(coeff[index], ++length)*elements[cp->index].z;
            temp[index] = sum;
            if (fabs(sum) > max_residual)
            max_residual = fabs(sum);
        }
        else
            temp[index] = 0.0;
    for (index = 0; index < size; index++)
        elements[index].z = temp[index];
    free(temp);
    return max_residual;
}
double checksum(int size, node *elements, int count, farray **coeff)
{
    cell *cp;
    int index, i;
    for (index = 0; index < size; index++)
        elements[index].z = 0.0;
    for (index = 0; index < size; index++)
        if (interior_node(elements + index)) {
            elements[index].z += fa_ref(coeff[index], 0);
            for (i = 1, cp = elements[index].neighbors;
                    cp != NULL;
                    i++, cp = cp->next)
            elements[cp->index].z += fa_ref(coeff[index], i);
        }
    return pi;
}
```


## C.3.26 jac_mat.c

## /**************

* jac_mat.c *
***************

Generate the matrix actually iterated in Jacobi iteration. We can then use MATLAB or something like that to analyze the damn thing. Won't work for large systems due to memory constraints.
*/
\#include <stdio.h>
\#include <math.h>

```
#include "gunk.h"
#include "matrix.h"
void jac_mat(FILE *, int, node *, farray **, double);
main(int argc, char *argv[])
{
    double val, omega = 1.0;
    farray **coeff;
    int size, i, j, N, count = 0;
    node *elements;
    FILE *fp;
    if (argc != 4 && argc != 5) {
        fprintf(stderr, "Usage: %s input-file coeff output-file [omega]\n",
                    argv[0]);
        exit(1);
    }
    fp = fopen(argv[1], "r");
    if (fp == NULL) {
        fprintf(stderr, "Error: Cannot open file \"%s\"\n", argv[1]);
        exit(1);
    }
    fscanf(fp, "%d", &size);
    fprintf(stderr, "Reading input (%d nodes)...\n", size);
    elements = (node *)calloc(sizeof(node), size);
    read_elements(fp, size, elements);
    close(fp);
    fp = fopen(argv[2], "r");
    if (fp == NULL) {
        fprintf(stderr, "Error: Cannot open file \"%s\"\n", argv[2]);
        exit(1);
    }
    fprintf(stderr, "Reading coefficients...\n");
    coeff = (farray **)calloc(sizeof(farray *), size);
    for (i = 0; i < size; i++) {
        getint(fp,&N);
        if (N > 0) {
            count++;
            coeff[i] = farray_cons(N);
            for (j = 0; j < N; j++) {
                getlf(fp, &val);
                fa_set(coeff[i], j, val);
            }
        }
    }
    fclose(fp);
```

```
    fprintf(stderr, "Composing %dx%d Jacobi iteration matrix...\n",
                count, count);
    if (argc == 5)
        sscanf(argv[4], "%lf", &omega);
    fp = fopen(argv[3], "w");
    jac_mat(fp, size, elements, coeff, omega);
    fclose(fp);
    fprintf(stderr, "Freeing memory...\n");
    free_memory(size, elements);
    for (i = 0; i < size; i++)
        farray_free(coeff[i]);
    free(coeff);
    return 0;
}
void jac_mat(FILE *fp, int size, node *elements, farray **coeff, double omega)
{
    cell *cp;
    double diag;
    int real_size = 0, i, j, *index, count;
    matrix *mat;
    for (i = 0; i < size; i++)
        if (interior_node(elements + i))
            real_size++;
    mat = matrix_cons(real_size, real_size);
    index = (int *)calloc(sizeof(int), size);
    count = 0;
    for (i = 0; i < size; i++)
        if (interior_node(elements + i))
            index[i] = count++;
        else
            index[i] = -1;
    for (i = 0; i < size; i++)
        if (interior_node(elements + i) 㛶 coeff[i] != NULL) {
            diag = fa_ref(coeff[i], 0);
            for (j = 1, cp = elements[i].neighbors;
                    cp != NULL;
                    j++, cp = cp->next)
            if (coeff[cp->index] != NULL)
                mset(mat, index[i], index[cp->index],
                    -omega*fa_ref(coeff[i], j)/diag);
        mset(mat, index[i], index[i], 1 - omega);
    }
    mprint(fp, mat);
    matrix_free(mat);
    free(index);
```


## C.3.27 list2grid.c

```
/*****************
* list2grid.c *
*****************
```

Takes a gunk file and produces average sample values over a regular grid of specified size. The output is sent to standard output. This can be used to plot a solution on Maple or Matlab, for example.

```
*/
```

\#include <stdio.h>
\#include "gunk.h"
void list2grid(FILE *, int, node *, int);
main(int argc, char *argv[])
$\{$
int grid_size, size;
node *elements;
FILE *fp;
if (argc $!=3$ ) \{
fprintf(stderr, "Usage: \%s input-file grid-size\n", argv[0]);
exit(1);
\}
fp $=$ fopen(argv[1], "r");
sscanf(argv[2], "\%d", \&grid_size);
if (fp == NULL) \{
fprintf(stderr, "Error: Cannot open file $\backslash " \% s \backslash " \backslash n ", ~ a r g v[1]) ;$
exit(2);
\}
fscanf(fp, "\%d", \&size);
fprintf(stderr, "Reading file (\% nodes)... \n", size);
elements $=$ (node *)calloc (sizeof (node), size);
read_elements (fp, size, elements);
fclose(fp);
fprintf(stderr, "Printing output... ${ }^{\text {n }}$ ");
list2grid(stdout, size, elements, grid_size);
fprintf(stderr, "Freeing memory... ${ }^{\prime}$ ");
free_memory (size, elements);
return 0;
\}
void list2grid(FILE *fp, int size, node *elements, int grid_size)
\{
double *value, *count, $x, y$;
int i, $j, n$;

```
    value = (double *)calloc(sizeof(double), SQUARE(grid_size));
    count = (double *)calloc(sizeof(double), SQUARE(grid_size));
    for (n = 0; n < size; n++) {
        x = elements[n].x;
        y = elements[n].y;
        i = (int)(x*grid_size);
        j = (int)(y*grid_size);
        if (i == grid_size) i--;
        if (j == grid_size) j--;
        *(count + i*grid_size + j) += 1.0;
        *(value + i*grid_size + j) += elements[n].z;
    }
    fprintf(fp, "%d\n", grid_size);
    for (j = 0; j < grid_size; j++) {
        for (i = 0; i < grid_size; i++) {
            n = *(count + i*grid_size + j);
            if (n > 0.0)
                fprintf(fp, "%.16lf ", *(value + i*grid_size + j)/n);
            else
                fprintf(fp, "%.16lf ", 0.0);
        }
        fprintf(fp, "\n");
    }
    free(value);
    free(count);
}
```


## C.3.28 lud.c

```
/***********
* lud.c *
************
Load a file of gunk and operator coefficients and try to
invert the matrix using LU decomposition. Right now we do
it the dumb way, so it takes lots and lots of memory.
*/
```

```
#include <stdio.h>
#include <math.h>
#include "gunk.h"
#include "matrix.h"
void solve(int, node *, farray **);
void read_coeff(FILE *, int, farray **);
```

```
main(int argc, char *argv[])
{
    farray **coeff;
    int size, i;
    node *elements;
    FILE *fp;
    if (argc != 4) {
        fprintf(stderr, "Usage: %s input-file coeff output-file\n",
                argv[0]);
        exit(1);
    }
    fp = fopen(argv[1], "r");
    if (fp == NULL) {
        fprintf(stderr, "Error: Cannot open file \"%s\"\n", argv[1]);
        exit(1);
    }
    fprintf(stderr, "Warning: This program has not been thoroughly tested\n");
    fprintf(stderr, " and may not operate correctly!\n");
    fscanf(fp, "%d", &size);
    fprintf(stderr, "Reading input (%d nodes)...\n", size);
    elements = (node *)calloc(sizeof(node), size);
    read_elements(fp, size, elements);
    close(fp);
    fp = fopen(argv[2], "r");
    if (fp == NULL) {
        fprintf(stderr, "Error: Cannot open file \"%s\"\n", argv[2]);
        exit(1);
    }
    fprintf(stderr, "Reading coefficients...\n");
    coeff = (farray **)calloc(sizeof(farray *), size);
    read_coeff(fp, size, coeff);
    fclose(fp);
    fprintf(stderr, "Solving system of equations...\n");
    solve(size, elements, coeff);
    fprintf(stderr, "Writing output...\n");
    fp = fopen(argv[3], "w");
    write_elements(fp, size, elements);
    fclose(fp);
    fprintf(stderr, "Freeing memory...\n");
    free_memory(size, elements);
    for (i = 0; i < size; i++)
        farray_free(coeff[i]);
    free(coeff);
    return 0;
}
```

```
void read_coeff(FILE *fp, int size, farray **coeff)
{
    double val;
    int i, j, N;
    for (i = 0; i < size; i++) {
        getint(fp, &N);
        if (N > 0) {
            coeff[i] = farray_cons(N);
            for (j = 0; j < N; j++) {
            getlf(fp, &val);
                fa_set(coeff[i], j, val);
            }
        }
    }
}
void solve(int size, node *elements, farray **coeff)
{
    cell *cp;
    double boundary, diag;
    int real_size = 0, i, j, N, *index;
    matrix *mat;
    for (i = 0; i < size; i++)
        if (interior_node(elements + i))
            real_size++;
    fprintf(stderr, "Generating %dx%d matrix...\n", real_size, real_size + 1);
    mat = matrix_cons(real_size, real_size + 1);
    index = (int *)calloc(sizeof(int), size);
    if (interior_node(elements))
        index[0] = 0;
    else
        index[0] = -1;
    for (i = 1; i < size; i++)
        if (interior_node(elements + i))
            index[i] = index[i - 1] + 1;
        else
            index[i] = index[i - 1];
    for (i = 0; i < size; i++)
        if (interior_node(elements + i) 统 coeff[i] != NULL) {
            N = fa_size(coeff[i]);
            diag = fa_ref(coeff[i], 0);
            mset(mat, index[i], index[i], 1.0);
            boundary = 0.0;
            for (j = 1, cp = elements[i].neighbors;
                    cP != NULL;
                    j++, cp = cp->next) {
            if (interior_node(elements + cp->index))
                mset(mat, index[i], index[cp->index], fa_ref(coeff[i], j)/diag);
```

```
            else
                boundary == fa_ref(coeff[i], j)*elements[cp->index].z/diag;
            mset(mat, index[i], real_size, boundary);
        }
        }
    fprintf(stderr, "Inverting matrix...\n");
    fprintf(stderr, "Determinant = %lf\n", rref(mat));
    for (i = 0; i < size; i++)
        if (interior_node(elements + i))
            elements[i].z = mref(mat, index[i], real_size);
    matrix_free(mat);
    free(index);
}
```


## C.3.29 maplegrid.c

```
/*****************
* maplegrid.c *
*****************
Input: A grid file, such as the output of list2grid.
Output: A file suitable for Maple consumption. This is to get around
    a stupid input limitation in Maple's matrix-input mechanism.
*/
#include <stdio.h>
void maplegrid(FILE *, FILE *, int, char *);
main(int argc, char *argv[])
{
    int ncols, i;
    FILE *fin, *fout;
    if (argc != 3) {
        fprintf(stderr, "Usage: %s input-file output-name\n", argv[0]);
        exit(1);
    }
    fin = fopen(argv[1], "r");
    fout =fopen(argv[2], "w");
    if ((fin == NULL) || (fout == NULL)) {
        fprintf(stderr, "Error: Cannot open file\n");
        exit(2);
    }
    fscanf(fin, "%d", &ncols);
    maplegrid(fin, fout, ncols, "foo");
    fclose(fin);
```

```
    fclose(fout);
    return 0;
}
void maplegrid(FILE *fin, FILE *fout, int ncols, char *name)
{
    double x;
    int i, j;
    fprintf(fout, "%s := [\n[", name);
    for (j = 0; ; j++) {
        if (fscanf(fin, "%lf", &x) == EOF)
            break;
        else if (j > 0)
            fprintf(fout, "],\n[");
        fprintf(fout, "%.16lf", x);
        for (i = 1; i < ncols; i++) {
            if (fscanf(fin, "%lf", &x) == EOF) {
                        fprintf(stderr, "Error: Premature EOF\n");
                exit(3);
            }
            fprintf(fout, ",%.16lf", x);
        }
    }
    fprintf(fout, "]\n]:\n");
    fprintf(fout, "%s := array(%s):\n", name, name);
    fprintf(fout, "%s := [seq([seq([i, j, %s[i, j]], i=1..%d)], j=1..%d)]:\n",
                name, name, ncols, j);
    fprintf(fout, "plots[surfdata] (%s, axes=frame, style=wireframe);\n", name);
}
```


## C.3.30 matrix.c

## /*************

* matrix.c *
**************

Matrix routines. LU decomposition for banded matrices should be implemented and explored, as well as Cholesky decomposition and conjugate gradient methods.

## */

```
#include <stdio.h>
#include <math.h>
#include "matrix.h"
#include "recipes/nr.h"
```

/****************************

* Constructor/destructor: *

```
******************************/
matrix *matrix_cons(int nrows, int ncols)
{
    int i;
    matrix *mat = (matrix *)malloc(sizeof(matrix));
    if (!(nrows > 0 && ncols > 0)) {
        fprintf(stderr, "Error: Illegal matrix dimensions\n");
        exit(1);
    }
    mat->nrows = nrows;
    mat->ncols = ncols;
    mat->el = (double **)calloc(sizeof(double *), nrows + 1);
    for (i = 0; i <= nrows; i++)
        mat->el[i] = (double *)calloc(sizeof(double), ncols + 1);
    return mat;
}
void matrix_free(matrix *mat)
{
    int i;
    if (mat != NULL) {
        if (mat->el != NULL) {
            for (i = 0; i <= mat->nrows; i++)
                if (mat->el[i] != NULL)
                    free(mat->el[i]);
            free(mat->el);
        }
        free(mat);
    }
}
/**************
* Methods: *
**************/
double mref(matrix *mat, int i, int j)
{
    if (i < 0 || mat->nrows <= i || j < 0 || mat->ncols <= j) {
        fprintf(stderr, "Error: Matrix access out of bounds\n");
        exit(1);
    }
    return mat->el[i + 1][j + 1];
}
void mset(matrix *mat, int i, int j, double val)
{
    if (i < 0 || mat->nrows <= i || j < 0 || mat->ncols <= j) {
        fprintf(stderr, "Error: Matrix access out of bounds\n");
        exit(1);
```

```
    }
    mat->el[i + 1][j + 1] = val;
}
/********************************
void exchange_rows(matrix *mat, int i1, int i2)
{
    double *vect;
    int nrows = mat->nrows, ncols = mat->ncols, j;
    if (i1 < 0 || nrows <= i1 || i2 < 0 || nrous <= i2) {
        fprintf(stderr, "Error: Illegal row exchange\n");
        exit(1);
    }
    if (i1 != i2) {
        vect = (double *)calloc(sizeof(double), ncols);
        for (j = 0; j < ncols; j++)
            vect[j] = mref(mat, i1, j);
        for (j = 0; j < ncols; j++)
            mset(mat, i1, j, mref(mat, i2, j));
        for (j = 0; j < ncols; j++)
            mset(mat, i2, j, vect[j]);
        free(vect);
    }
}
void scale_row(matrix *mat, int i, double a)
{
    int nrows = mat->nrous, ncols = mat->ncols, j;
    if (i < 0 || nrows <= i) {
        fprintf(stderr, "Error: Illegal row scaling\n");
        exit(1);
    }
    for (j = 0; j < ncols; j++)
        mset(mat, i, j, a*mref(mat, i, j));
}
void add_rows(matrix *mat, int i1, double a, int i2)
{
    int nrows = mat->nrows, ncols = mat->ncols, j;
    if (i1 < 0 || nrows <= i1 || i2 < 0 || nrows <= i2) {
        fprintf(stderr, "Error: Illegal row addition\n");
        exit(1);
    }
    for (j = 0; j < ncols; j++)
```

```
    mset(mat, i2, j, mref(mat, i2, j) + a*mref(mat, i1, j));
```

$\}$

```
/************************
* Matrix operations: *
***********************/
void madd(matrix *sum, matrix *mat1, matrix *mat2)
{
    int nrows, ncols, i, j;
    if (!(sum->nrows == mat1->nrows && mat1->nrows == mat2->nrows &&
            sum->ncols == mat1->ncols && mat1->ncols == mat2->ncols)) {
        fprintf(stderr, "Error: Can only add matrices of same dimensions\n");
        exit(1);
    }
    nrows = sum->nrows;
    ncols = sum->ncols;
    for (i = 0; i < nrows; i++)
        for (j = 0; j < ncols; j++)
            mset(sum, i, j, mref(mat1, i, j) + mref(mat2, i, j));
}
void mmult(matrix *product, matrix *mat1, matrix *mat2)
{
    double sum;
    int i, j, k, m = product->nrows, n = product->ncols, p = mat1->ncols;
    if (!(mat1->ncols == mat2->nrows &&
                    mat1->nrows == product->nrows &&
            mat2->ncols == product->ncols)) {
        fprintf(stderr, "Error: Invalid matrix dimensions for multiplication\n");
        exit(1);
    }
    for (i = 0; i < m; i++)
        for (j = 0; j< n; j++) {
            sum = 0.0;
            for (k = 0; k < p; k++)
                sum += mref(mat1, i, k)*mref(mat2, k, j);
            mset(product, i, j, sum);
        }
}
matrix *transpose(matrix *mat)
{
    int i, j, m = mat->nrows, n = mat->ncols;
    matrix *matt = matrix_cons(n, m);
    for (i = 0; i < m; i++)
        for (j = 0; j < n; j++)
            mset(matt, j, i, mref(mat, i, j));
    return matt;
```

```
}
double rref(matrix *mat)
{
    double det;
    int i, j, k, N = mat->nrows, ncols = mat->ncols;
    det = ludecomp(mat);
    for (j = 0; j < N; j++)
            for (i = j + 1; i < N; i++) {
                for (k = N; k < ncols; k++)
                    mset(mat, i, k, mref(mat, i, k) - mref(mat, j, k)*mref(mat, i, j));
                mset(mat, i, j, 0.0);
            }
    for (j = N - 1; 0 <= j; j--) {
            scale_row(mat, j, 1.0/mref(mat, j, j));
        for (i = j - 1; 0 <= i; i--)
            add_rows(mat, j, -mref(mat, i, j), i);
    }
    return det;
}
double ludecomp(matrix *mat)
{
    double sum, pivot, det = 1.0;
    int N = mat->nrows, i, j, k, pivot_index;
    if (mat->ncols < N) {
        fprintf(stderr, "Error: Matrix has more rows than columns\n");
        exit(1);
    }
    for (j = 0; j < N; j++) {
        pivot_index = j;
        pivot = fabs(mref(mat, j, j));
        for (i = j + 1; i< < N; i++)
            if (fabs(mref(mat, i, j)) > pivot) {
                    pivot_index = i;
                pivot = fabs(mref(mat, i, j));
            }
        if (j != pivot_index) {
            exchange_rows(mat, j, pivot_index);
            det *= -1.0;
        }
        for (i= 0; i <= j; i++) {
            sum = mref(mat, i, j);
            for (k = 0; k < i; k++)
            sum == mref(mat, i, k)*mref(mat, k, j);
            mset(mat, i, j, sum);
```

```
        }
        pivot = mref(mat, j, j);
        det *= pivot;
        for (i = j + 1; i< N; i++) {
            sum = mref(mat, i, j);
            for (k = 0; k < j; k++)
            sum -= mref(mat, i, k)*mref(mat, k, j);
            mset(mat, i, j, sum/pivot);
        }
    }
    if (fabs(det) < TOL)
        fprintf(stderr, "Warning: Nearly singular matrix (determinant = %.16lg)\n",
                        det);
    return det;
}
void svdecomp(matrix *mat, farray *w, matrix *v)
{
    if (w->size != mat->ncols ||
        v->nrows != v->ncols ||
            v->nrows != w->size) {
        fprintf(stderr, "Error: SVD input matrices have incorrect dimensions\n");
        exit(1);
    }
    dsvdcmp(mat->el, mat->nrows, mat->ncols, w->array, v->el);
}
/***********************************
* Print out a matrix nicely: *
*******************************/
void mprint(FILE *fp, matrix *mat)
{
    int nrows = mat->nrows, ncols = mat->ncols, i, j;
    for (i = 0; i < nrows; i++) {
        fprintf(fp, "%.16lf", mref(mat, i, 0));
        for (j = 1; j < ncols; j++)
            fprintf(fp, "\t%.16lf", mref(mat, i, j));
        fprintf(fp, "\n");
    }
}
void mprintf(FILE *fp, char *format, matrix *mat)
{
    int nrows = mat->nrows, ncols = mat->ncols, i, j;
    for (i = 0; i < nrows; i++) {
        fprintf(fp, format, mref(mat, i, 0));
```

```
        for (j = 1; j < ncols; j++) {
        fprintf(fp, "\t");
        fprintf(fp, format, mref(mat, i, j));
        }
        fprintf(fp, "\n");
    }
}
/************************
* Little utilities: *
*************************/
double tolerance(void)
{
    return tolerance_value;
}
void set_tolerance(double new_val)
{
    tolerance_value = new_val;
}
```


## C.3.31 migrate.c

```
/****************
* migrate.c *
***************
```

Change the density of nodes.
*/

```
#include <stdio.h>
#include <math.h>
#include "gunk.h"
#include "random.h"
void migrate(int, node *, int, int);
void create_boundary(int, node *, int);
double remap_x(double, double);
double remap_y(double, double);
double multiplier(double, double);
main(int argc, char *argv[])
{
    int size, cell_count, real_size;
    node *elements;
    FILE *fp;
    initialize_random();
    if (argc != 4) {
        fprintf(stderr, "Usage: %s file number-of-new-nodes 1/radius\n",
            argv[0]);
```

```
        exit(1);
    }
    sscanf(argv[2], "%d", &real_size);
    sscanf(argv[3], "%d", &cell_count);
    fp = fopen(argv[1], "r");
    if (fp == NULL) {
        fprintf(stderr, "Error: Cannot open file \"%s\"\n", argv[1]);
        exit(1);
    }
    fscanf(fp, "%d", &size);
    fprintf(stderr, "Reading input file (%d nodes)...\n", size);
    real_size += size;
    elements = (node *)calloc(sizeof(node), real_size);
    read_elements(fp, size, elements);
    fclose(fp);
    fprintf(stderr, "Adding %d new nodes...\n", real_size - size);
    migrate(real_size, elements, cell_count, size);
    create_boundary(real_size, elements, cell_count);
    fprintf(stderr, "Writing output file (%d nodes)...\n", real_size);
    fp = fopen(argv[1], "w");
    write_elements(fp, real_size, elements);
    fclose(fp);
    fprintf(stderr, "Freeing memory...\n");
    free_memory(real_size, elements);
    return 0;
}
void migrate(int real_size, node *elements, int cell_count, int size)
{
    cell *cp, *cp_next;
    double x, y, xr, yr, radius = 1.0/cell_count, search_radius;
    int n, i, j, p, q;
    partition *partition = make_partition(cell_count);
    for (n = 0; n < size; n++) {
        i = (int)floor(elements[n].x*cell_count);
        j = (int)floor(elements[n].y*cell_count);
        partition_add(partition, i, j, n);
    }
    for (n = size; n < real_size; n++) {
        xr = frandom();
        yr = frandom();
        x = remap_x(xr, yr);
        y = remap_y(xr, yr);
        elements[n].x = x;
        elements[n].y = y;
        elements[n].z = potential(x, y);
        /* Free cells first. */
```

```
        cp = elements[n].neighbors;
        for (cp = elements[n].neighbors; cp != NULL; ) {
            cp_next = cp->next;
            free(cp);
            cp = cp_next;
        }
        elements[n].neighbors = NULL;
        set_interior_node(elements + n);
        i = (int)floor(x*cell_count);
        j = (int)floor(y*cell_count);
        search_radius = ceil(multiplier(x, y));
        for (p = i - search_radius; p <= i + search_radius; p++)
            for (q = j - search_radius; q <= j + search_radius; q++)
                if (0 <= p && p < cell_count 奴 0 <= q && q < cell_count)
                    for (cp = partition_get(partition, p, q); cp != NULL; cp = cp->next)
                        introduce(elements, n, cp->index, radius*multiplier(x, y));
        partition_add(partition, i, j, n);
    }
    free_partition(partition);
}
void create_boundary(int size, node *elements, int cell_count)
{
    double x, y;
    int n;
    for (n = 0; n < size; n++) {
        x = elements[n].x;
        y = elements[n].y;
        if ((x <= 0.0) || (1.0<= x) ||
            (y<= 0.0) || (1.0<= y))
            set_boundary_node(elements + n);
    }
}
double remap_x(double x, double y)
{
    return x;
}
double remap_y(double x, double y)
{
    return sqrt(y);
}
double multiplier(double x, double y)
{
    return 1.0;
}
```


## C.3.32 mkbd.c

```
/************
* mkbd.c *
*************
Creates a square of gunk, as in the rectangular slot problem in
electrostatics. The boundary nodes are given the correct values,
while the interior nodes can either be initialized with the correct
value or randomized. The nodes are randomly distributed.
```

*/
\#include <stdio.h>
\#include <math.h>
\#include "gunk.h"
\#include "random.h"
/* \#define INIT_TO_TRUE_VALS */
void make_nodes (int, node $*$, int, int, double);
void create_boundary (int, node *, int);
/* This stuff is used to make the node lists more readable. */
void sort_neighbors(int, node *);
double compute_angle(node *, int, int);
main(int argc, char *argv[])
\{
double min_dist $=0.0$;
int size, cell_count, bd_size;
node *elements;
FILE *fp;
initialize_random();
if (argc != 4 \&\& argc != 5) \{
fprintf(stderr,
"Usage: \%s output-file number-of-nodes $1 /$ radius [min-dist] $\backslash n$ ",
argv[0]) ;
exit(1);
\}
sscanf(argv[2], " $1 / \mathrm{d}$ ", \&size);
sscanf(argv[3], "\%d", \&cell_count);
bd_size $=8 *(($ int $)$ sqrt (size) +1$)$;
size += bd_size;
elements $=$ (node *) calloc(sizeof(node), size);
fprintf(stderr, "Creating a square containing \%d nodes... \n", size);
if $(\operatorname{argc}==5)$
sscanf(argv[4], "\%1f", \&min_dist);
if (min_dist > 0)

```
        fprintf(stderr, "(Maximum number: %d nodes)\n",
                        (int)(1/(2*pi*fsquare(min_dist/2))));
    make_nodes(size, elements, cell_count, size - bd_size, min_dist);
    create_boundary(size, elements, cell_count);
    /* sort_neighbors(size, elements); */
    fprintf(stderr, "Writing output file...\n");
    fp = fopen(argv[1], "w");
    write_elements(fp, size, elements);
    fclose(fp);
    fprintf(stderr, "Freeing memory...\n");
    free_memory(size, elements);
    return 0;
}
void make_nodes(int size, node *elements, int cell_count, int int_size,
                double min_dist)
{
    cell *cp;
    double x, y, radius = 1.0/cell_count, bd = 0.0;
    double delta = 4.0/(size - int_size);
    int n, i, j, p, q, int_part, too_close, close_encounter_count = 0;
    partition *partition = make_partition(cell_count);
    for (n = 0; n < size; n += (too_close ? 0 : 1)) {
        if (n >= int_size) {
            bd = (n - int_size)*delta;
            int_part = (int)bd;
            bd -= int_part;
            x = int_part%2 ? int_part%3 : bd;
            y = int_part%2 ? bd : (int_part + 1)%3;
        }
        else {
            x = frandom();
            y = frandom();
#ifdef INIT_TO_TRUE_VALS
            elements[n].z = potential(x, y);
#else
            elements[n].z = 0.0;
#endif
    }
    i = (int)floor(x*cell_count);
    j = (int)floor(y*cell_count);
    elements[n].neighbors = NULL;
    elements[n].x = x;
    elements[n].y = y;
    set_interior_node(elements + n);
    too_close = 0;
    if (min_dist > 0.0 && n < int_size) {
        for (p = i - 1; p <= i + 1 && !too_close; p++)
            for (q = j - 1; q <= j + 1 && !too_close; q++)
                if ((0 <= p) && (p < cell_count) && ( }0<=q\mathrm{ ) && (q < cell_count))
```

```
                for (cp = partition_get(partition, p, q);
                cp != NULL;
                cp = cp->next)
                            if (distance(elements + n, elements + cp->index) < min_dist) {
                        too_close = 1;
                        close_encounter_count++;
                        break;
                    }
        }
        if (!too_close) {
            for (p = i - 1; p <= i + 1; p++)
                for (q = j - 1; q<= j + 1; q++)
                    if (( }0<= p)&& (p<cell_count) && ( 0 <= q) && (q < cell_count)
                for (cp = partition_get(partition, p, q);
                    cp != NULL;
                            cp = cp->next)
                        introduce(elements, n, cp->index, radius);
            partition_add(partition, i, j, n);
        }
    }
    fprintf(stderr, "(%d close encounters.)\n", close_encounter_count);
    free_partition(partition);
}
void create_boundary(int size, node *elements, int cell_count)
{
    cell *cp;
    double x, y, d, max_d, radius = 1.0/cell_count;
    int n;
    for (n = 0; n < size; n++) {
        x = elements[n].x;
        y = elements[n].y;
        if ((x<= 0.0) || (1.0<= x) ||
            (y<= 0.0) || (1.0<= y)) {
            max_d = 0.0;
            for (cp = elements[n].neighbors; cp != NULL; cp = cp->next) {
                d = distance(elements + n, elements + cp->index);
                if (d > max_d) max_d = d;
            }
            if (x == 0.0)
                x -= frandom()*(radius - max_d);
            else if ( }x==1.0\mathrm{ )
                x += frandom()*(radius - max_d);
            else if (y == 0.0)
                y == frandom()*(radius - max_d);
            else if (y == 1.0)
                y += frandom()*(radius - max_d);
            elements[n].x = x;
            elements[n].y = y;
            elements[n].z = potential(x, y);
            set_boundary_node(elements + n);
```

```
        }
    }
}
void sort_neighbors(int size, node *elements)
{
    cell *cp;
    double *keys, val;
    int i, m, n, length, *cells, index;
    for (i = 0; i < size; i++)
        if (interior_node(elements + i)) {
            length = 0;
            for (cp = elements[i].neighbors; cp != NULL; cp = cp->next)
                length++;
            cells = (int *)calloc(sizeof(int), length);
            keys = (double *)calloc(sizeof(double), length);
            length = 0;
            for (cp = elements[i].neighbors; cp != NULL; cp = cp->next) {
                cells[length] = cp->index;
                /* keys[length] = compute_angle(elements, i, cp->index); */
                keys[length] = -(cp->index);
                length++;
            }
            for (m = 1; m < length; m++) {
                val = keys[m];
                index = cells[m];
                for (n = m - 1; keys[n] < val && n >= 0; n--) {
                    keys[n + 1] = keys[n];
                    cells[n + 1] = cells[n];
                }
                keys[n + 1] = val;
                cells[n + 1] = index;
            }
            length = 0;
            for (cp = elements[i].neighbors; cp != NULL; cp = cp->next)
                cp->index = cells[length++];
            free(keys);
            free(cells);
        }
}
double compute_angle(node *elements, int i, int j)
{
    double x = elements[j].x - elements[i].x, y = elements[j].y - elements[i].y;
    double result;
    if (x > 0.0)
        result = atan(y/x);
    else
```

```
        result = atan(y/x) + pi;
    if (result < 0.0)
        result = 2*pi + result;
    return result;
}
```


## C.3.33 mkgrid.c

```
/*************
* mkgrid.c *
**************
```

Serves the same function as mkbd, only the nodes are placed on a regular grid.
*/

```
#include <stdio.h>
#include <math.h>
#include "gunk.h"
#define INIT_TO_TRUE_VALS
void make_nodes(int, int, node *);
main(int argc, char *argv[])
{
    int size, count, radius;
    node *elements;
    FILE *fp;
    if (argc != 4) {
        fprintf(stderr, "Usage: %s output-file number-of-nodes/side radius\n",
            argv[0]);
        exit(1);
    }
```

    sscanf(argv[2], "\%d", \&count);
    sscanf(argv[3], "\%d", \&radius);
    size = SQUARE (count);
    elements \(=\) (node \(*\) ) calloc (sizeof(node), size);
    fprintf(stderr, "Creating a \%dx\%d grid (\% nodes)... \({ }^{n}\) ", count, count, size);
    make_nodes(count, radius, elements);
    fprintf(stderr, "Writing output file... \(\ln\) ");
    \(\mathrm{fp}=\) fopen (argv[1], "w");
    write_elements(fp, size, elements);
    fclose(fp);
    fprintf(stderr, "Freeing memory...\n");
    free_memory(size, elements);
    return 0 ;
    ```
}
void make_nodes(int count, int radius, node *elements)
{
    cell *cp;
    double delta = 1.0/(count - 1), x, y, a = (radius - 0.5)*delta, b = 1.0 - a;
    int i, j, p, q, index, neighbor_index, size = SQUARE(count);
    node *new;
    for (j = 0; j < count; j++)
        for (i = 0; i < count; i++) {
            index = j*count + i;
        x = i*delta;
        y = j*delta;
        elements[index].x = x;
        elements[index].y = y;
#ifdef INIT_TO_TRUE_VALS
        elements[index].z = potential(x, y);
#else
        elements[index].z = 0.0;
#endif
            elements[index] .neighbors = NULL;
        if (a<x && x < b && a < y && y < b)
            for (p = -radius; p <= radius; p++)
                for (q = -radius; q <= radius; q++) {
                    neighbor_index = (j + q)*count + i + p;
                    set_interior_node(elements + index);
                    if (0 <= neighbor_index 炈 neighbor_index < size 㛶
                        neighbor_index != index) {
                        cp = (cell *)malloc(sizeof(cell));
                        cp->next = elements[index].neighbors;
                        elements[index].neighbors = cp;
                        cp->index = neighbor_index;
                }
                }
        else {
            elements[index].z = potential(x, y);
            set_boundary_node(elements + index);
        }
    }
}
```


## C.3.34 mksq.c

```
/*************
* mksq.c *
```

************

An obsolete, older version of mkbd.c.

## */

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include "gunk.h"
typedef struct {
    cell **list;
    int size;
} grid;
void make_nodes(int, node *, int);
grid *make_grid(int);
void grid_add(grid *, int, int, int);
cell *grid_get(grid *, int, int);
void free_grid(grid *);
void introduce(node *, int, int, double);
int grid_pos_x(int);
int grid_pos_y(int);
void rmbd(int, node *, int);
void create_boundary(int, node *, int);
main(int argc, char *argv[])
{
    int size, cell_count;
    node *elements;
    FILE *fp;
    srand48(time(NULL));
    if (argc != 4) {
        fprintf(stderr, "Usage: %s output-file number-of-nodes 1/radius\n",
                    argv[0]);
        exit(1);
    }
    sscanf(argv[2], "%d", &size);
    sscanf(argv[3], "%d", &cell_count);
    size = (int)floor((double)size*fsquare(cell_count + 2)/fsquare(cell_count));
    cell_count += 2;
    elements = (node *)calloc(sizeof(node), size);
    fprintf(stderr, "Creating a square containing %d nodes...\n", size);
    make_nodes(size, elements, cell_count);
    create_boundary(size, elements, cell_count);
    rmbd(size, elements, cell_count);
    fprintf(stderr, "Writing output file...\n");
    fp = fopen(argv[1], "w");
    write_elements(fp, size, elements);
    fclose(fp);
    fprintf(stderr, "Freeing memory...\n");
    free_memory(size, elements);
    return 0;
}
void make_nodes(int size, node *elements, int cell_count)
```

```
{
    cell *cp;
    double x, y, radius = 1.0/cell_count;
    int n, i, j, k, p, q;
    grid *partition = make_grid(cell_count);
    for (n = 0; n < size; n++) {
        x = drand48();
        y = drand48();
        i = (int)floor(x*cell_count);
        j = (int)floor(y*cell_count);
        elements[n].neighbors = NULL;
        elements[n].x = x;
        elements[n].y = y;
        elements[n].z = 0.0;
        for (k = 0; k < 9; k++) {
            p = i + grid_pos_x(k);
            q = j + grid_pos_y(k);
            if (( }0<= p) && (p< cell_count) && ( 0 <= q) && (q < cell_count)
                for (cp = grid_get(partition, p, q); cp != NULL; cp = cp->next)
                introduce(elements, n, cp->index, radius);
        }
        grid_add(partition, i, j, n);
    }
    free_grid(partition);
}
void create_boundary(int size, node *elements, int cell_count)
{
    cell *cp, *cp_next;
    int n, i, j;
    for (n = 0; n < size; n++) {
        i = (int)floor(elements[n].x*cell_count);
        j = (int)floor(elements[n].y*cell_count);
        if ((i == 0) || (i == cell_count - 1) ||
            (j == 0) || (j == cell_count - 1)) {
            cp = elements[n].neighbors;
        while (cp != NULL) {
            cp_next = cp->next;
            free(cp);
            cp = cp_next;
        }
            elements[n].neighbors = NULL;
            if (j == cell_count - 1)
                elements[n].z = 1.0;
            else
            elements[n].z = 0.0;
        }
    }
```

```
}
void introduce(node *elements, int i, int j, double radius)
{
    cell *new_cell1, *new_cell2;
    if (distance(elements + i, elements + j) < radius) {
        new_cell1 = (cell *)malloc(sizeof(cell));
        new_cell2 = (cell *)malloc(sizeof(cell));
        new_cell1->next = elements[i].neighbors;
        new_cell2->next = elements[j].neighbors;
        elements[i].neighbors = new_cell1;
        elements[j].neighbors = new_cell2;
        new_cell1->index = j;
        new_cell2->index = i;
    }
}
grid *make_grid(int size)
{
    grid *new_grid = (grid *)malloc(sizeof(grid));
    new_grid->list = (cell **)calloc(sizeof(cell *), SQUARE(size));
    new_grid->size = size;
    return new_grid;
}
void grid_add(grid *partition, int i, int j, int index)
{
    cell **list = partition->list, *new_cell;
    int size = partition->size;
    if ((i >= size) || (j >= size)) {
        fprintf(stderr, "Error: Grid access out of bounds\n");
        exit(2);
    }
    new_cell = (cell *)malloc(sizeof(cell));
    new_cell->next = *(list + i*size + j);
    *(list + i*size + j) = new_cell;
    new_cell->index = index;
}
cell *grid_get(grid *partition, int i, int j)
{
    cell **list = partition->list;
    int size = partition->size;
    if ((i >= size) || (j >= size)) {
        fprintf(stderr, "Error: Grid access out of bounds\n");
        exit(3);
    }
    return *(list + i*size + j);
}
```

```
void free_grid(grid *partition)
{
    cell **list = partition->list, *cp, *cp_next;
    int size = partition->size, i, j;
    for (i = 0; i < size; i++) {
        for (j = 0; j < size; j++) {
            cp = *(list + i*size + j);
            while (cp != NULL) {
                cp_next = cp->next;
                free(cp);
                cp = cp_next;
            }
        }
    }
    free(partition);
}
int grid_pos_x(int n)
{
    return n/3 - 1;
}
int grid_pos_y(int n)
{
    return n%3-1;
}
void rmbd(int size, node *elements, int cell_count)
{
    int n;
    double ratio;
    if (cell_count > 2) {
        ratio = cell_count/(cell_count - 2.0);
        for (n = 0; n < size; n++) {
            elements[n].x = (elements[n].x - .5)*ratio + .5;
            elements[n].y = (elements[n].y - .5)*ratio + .5;
            if ((0.0< elements[n].x) && (elements[n].x < 1.0) &&
                (0.0< elements[n].y) && (elements[n].y < 1.0))
            elements[n].z = potential(elements[n].x, elements[n].y);
        }
    }
}
```


## C.3.35 new2old.c

```
/***************
* new2old.c *
****************
```

Converts a binary gunk file to an ASCII file.
*/

```
#include <stdio.h>
#include "gunk.h"
main(int argc, char *argv[])
{
    int size;
    node *elements;
    FILE *fp;
    if (argc != 3) {
        fprintf(stderr, "Usage: %s input-file output-file\n", argv[0]);
        exit(1);
    }
    fp = fopen(argv[1], "r");
    if (fp == NULL) {
        fprintf(stderr, "Error: Cannot open file \"%s\"\n", argv[1]);
        exit(1);
    }
    fscanf(fp, "%d", &size);
    fprintf(stderr, "Reading input file (%d nodes)...\n", size);
    elements = (node *)calloc(sizeof(node), size);
    read_elements(fp, size, elements);
    fclose(fp);
    fprintf(stderr, "Writing output file (%d nodes)...\n", size);
    fp = fopen(argv[2], "w");
    old_write_elements(fp, size, elements);
    fclose(fp);
    fprintf(stderr, "Freeing memory...\n");
    free_memory(size, elements);
    return 0;
}
```


## C.3.36 norm_eqs.c

## /***************

* norm_eqs.c *
*****************

Input: A gunk file, such as the output of mkbd or mkgrid, and a file of coefficients, such as the output of approx.

Output: A new gunk file and new coefficients, arranged so that the relaxation matrix is symmetric.
*/
\#include <stdio.h>

```
#include <math.h>
#include "gunk.h"
#include "farray.h"
/* Print debug info? */
/* #define DEBUG */
/* #define TEST_KLUGE */
void read_coeff(FILE *, int, farray **);
void write_coeff(FILE *, int, farray **);
void free_coeff(int, farray **);
void normal_eqs(int, node *, farray **);
void sort_neighbors(int, node *, farray **);
void check_symmetry(int, node *, farray **);
main(int argc, char *argv[])
{
    farray **coeff;
    int size;
    node *elements;
    FILE *fp;
    if (argc != 5) {
        fprintf(stderr, "Usage: %s gunk-in coeff-in gunk-out coeff-out\n",
                argv[0]);
        exit(1);
    }
    fp = fopen(argv[1], "r");
    if (fp == NULL) {
        fprintf(stderr, "Error: Cannot open file \"%s\"\n", argv[1]);
        exit(2);
    }
    fscanf(fp, "%d", &size);
    elements = (node *)calloc(sizeof(node), size);
    fprintf(stderr, "Reading input file (%d nodes)...\n", size);
    read_elements(fp, size, elements);
    fclose(fp);
    fp = fopen(argv[2], "r");
    if (fp == NULL) {
        fprintf(stderr, "Error: Cannot open file \"%s\"\n", argv[2]);
        exit(3);
    }
    fprintf(stderr, "Reading coefficients...\n");
    coeff = (farray **)calloc(sizeof(farray *), size);
    read_coeff(fp, size, coeff);
    fclose(fp);
    /* Sort old neighborhoods first. */
    fprintf(stderr, "Sorting neighbor sets...\n");
    sort_neighbors(size, elements, coeff);
```

```
    /* Compute normal equations. */
    normal_eqs(size, elements, coeff);
    fprintf(stderr, "Checking symmetry...\n");
    check_symmetry(size, elements, coeff);
    fprintf(stderr, "Writing new gunk file...\n");
    fp = fopen(argv[3], "w");
    write_elements(fp, size, elements);
    fclose(fp);
    fprintf(stderr, "Freeing gunk memory...\n");
    free_memory(size, elements);
    fprintf(stderr, "Writing new coefficients...\n");
    fp = fopen(argv[4], "w");
    write_coeff(fp, size, coeff);
    fclose(fp);
    fprintf(stderr, "Freeing coefficient memory...\n");
    free_coeff(size, coeff);
    return 0;
}
void free_coeff(int size, farray **coeff)
{
    int i;
    for (i = 0; i < size; i++)
        farray_free(coeff[i]);
    free(coeff);
}
void write_coeff(FILE *fp, int size, farray **coeff)
{
    int i, j, N;
    for (i = 0; i < size; i++) {
        N = fa_size(coeff[i]);
        putint(fp,N);
        for (j = 0; j < N; j++)
            putlf(fp, fa_ref(coeff[i], j));
    }
}
void read_coeff(FILE *fp, int size, farray **coeff)
{
    double val;
    int i, j, N;
    for (i = 0; i < size; i++) {
        getint(fp, &N);
        if (N > 0) {
            coeff[i] = farray_cons(N);
            getlf(fp, &val);
```

```
            fa_set(coeff[i], 0, val);
            for (j = 1; j < N; j++) {
                getlf(fp, &val);
                fa_set(coeff[i], j, val);
            }
        }
    }
}
/**
    * Generate normal equations for the system of equations generated by
    * gen_coeff. This has the distinct advantage that the resulting matrix is
    * symmetric positive-definite. This function assumes that neighbor lists have
    * already been sorted.
    **/
void normal_eqs(int size, node *elements, farray **coeff)
{
    cell **new_neighbors, *cp, *cq, *new, *prev, head;
    double sum;
    farray **new_coeff;
    int count, i, j;
    double inner_product(int, int, node *, farray **);
    /* First step: For each node, compute the union of its neighbor-set and its
    * neighbors' neighbor-sets. */
    fprintf(stderr, "Collecting two-hop neighbors...\n");
    new_neighbors = (cell **)calloc(sizeof(cell *), size);
    new_coeff = (farray **)calloc(sizeof(farray *), size);
    head.index = -1;
    for (i = 0; i < size; i++) {
        /* Copy neighbors first, assuming the list is already sorted. */
        head.next = NULL;
        prev = &head;
        for (cp = elements[i].neighbors; cp != NULL; cp = cp->next) {
            new = (cell *)malloc(sizeof(cell));
            new->index = cp->index;
            new->next = NULL;
            prev->next = new;
            prev = new;
    }
    /* Then copy neighbors' neighbors, taking care to sort the list. */
    for (cp = elements[i].neighbors; cp != NULL; cp = cp->next)
            for (cq = elements[cp->index].neighbors; cq != NULL; cq = cq->next)
                if (cq->index != i) {
                for (prev = &head, new = head.next; new != NULL; new = new->next)
                    if (new->index > cq->index)
                break;
                        else
```

```
                    prev = new;
            if (prev->index != cq->index) {
                new = (cell *)malloc(sizeof(cell));
                new->next = prev->next;
                prev->next = new;
                new->index = cq->index;
            }
        }
    new_neighbors[i] = head.next;
}
/* Next, for each node, compute its new coefficients for each member of its
    * newly-computed neighbors' neighbor sets. */
fprintf(stderr, "Computing new coefficients...\n");
for (i = 0; i < size; i++)
    if (interior_node(elements + i) && new_neighbors[i] != NULL) {
        /* Allocate some space for the new coefficients. */
        for (count = 1, cp = new_neighbors[i];
            cp != NULL;
            cp = cp->next, count++);
        new_coeff[i] = farray_cons(count);
        /* Compute the new coefficients. Do we need to pay special attention to
            * boundary nodes? */
        for (j = 1, cp = new_neighbors[i]; cp != NULL; j++, cp = cp->next)
            fa_set(new_coeff[i], j, inner_product(i, cp->index, elements, coeff));
#ifdef TEST_KLUGE
    sum = 0.0;
    for (j = 1, cp = new_neighbors[i]; cp != NULL; j++, cp = cp->next)
            sum += fabs(fa_ref(new_coeff[i], j));
        fa_set(new_coeff[i], 0, sum);
#else
    fa_set(new_coeff[i], 0, inner_product(i, i, elements, coeff));
#endif
    }
/* Finally, write over the old data structures and free up the
    * temporarily-allocated memory... */
    fprintf(stderr, "Freeing temporary memory...\n");
    for (i = 0; i < size; i++) {
        cp = elements[i].neighbors;
        while (cp != NULL) {
            cq = cp->next;
            free(cp);
            cp = cq;
```

```
        }
        elements[i].neighbors = new_neighbors[i];
        /* Copy coefficients. */
        farray_free(coeff[i]);
        coeff[i] = new_coeff[i];
    }
    /* Free temporary memory. */
    free(new_neighbors);
    free(new_coeff);
}
/**
    * Sort neighbor lists by index.
    **/
void sort_neighbors(int size, node *elements, farray **coeff)
{
    cell *cp;
    int i, j;
    void sort_without_coeff(node *);
    void sort_with_coeff(node *, farray *);
    for (i = 0; i < size; i++)
        if (coeff[i] == NULL)
            sort_without_coeff(elements + i);
        else {
#ifdef DEBUG
            fprintf(stderr, "\nNode %d (%d neighbors):\n", i, fa_size(coeff[i]));
            fprintf(stderr, "Before: (%d %lf)", i, fa_ref(coeff[i], 0));
            for (j = 1, cp = elements[i].neighbors; cp != NULL; j++, cp = cp->next)
            fprintf(stderr, " (%d %lf)", cp->index, fa_ref(coeff[i], j));
            fprintf(stderr, "\n");
#endif
            sort_with_coeff(elements + i, coeff[i]);
#ifdef DEBUG
            fprintf(stderr, "After: (%d %lf)", i, fa_ref(coeff[i], 0));
            for (j = 1, cp = elements[i].neighbors; cp != NULL; j++, cp = cp->next)
                fprintf(stderr, " (%d %lf)", cp->index, fa_ref(coeff[i], j));
            fprintf(stderr, "\n");
#endif
        }
}
void sort_without_coeff(node *node)
{
    cell *cp;
    int i, m, n, length, *key, index;
```

```
    for (length = 0, cp = node->neighbors; cp != NULL; length++, cp = cp->next);
    key = (int *)calloc(sizeof(int), length);
    for (i = 0, cp = node->neighbors; cp != NULL; i++, cp = cp->next)
        key[i] = cp->index;
    for (m = 1; m < length; m++) {
        index = key[m];
        for (n = m - 1; key[n] > index && n >= 0; n--)
            key[n + 1] = key[n];
        key[n + 1] = index;
    }
    for (i = 0, cp = node->neighbors; cp != NULL; i++, cp = cp->next)
        cp->index = key[i];
    free(key);
}
void sort_with_coeff(node *node, farray *coeff)
{
    cell *cp;
    double val, *new;
    int i, m, n, length, *key, index;
    length = fa_size(coeff) - 1;
    key = (int *)calloc(sizeof(int), length);
    new = (double *) calloc(sizeof(double), length);
    for (i = 0, cp = node->neighbors; cp != NULL; i++, cp = cp->next) {
        key[i] = cp->index;
        new[i] = fa_ref(coeff, i + 1);
    }
    for (m = 1; m < length; m++) {
        index = key[m];
        val = fa_ref(coeff, m + 1);
        for (n = m- 1; key[n] > index && n >= 0; n--) {
            key[n + 1] = key[n];
            new[n+1] = new[n];
        }
        key[n + 1] = index;
        new[n + 1] = val;
    }
    for (i = 0, cp = node->neighbors; cp != NULL; i++, cp = cp->next) {
        cp->index = key[i];
        fa_set(coeff, i + 1, new[i]);
    }
    free(key);
    free(new);
}
```

```
/**
    * This procedure assumes that the neighbor lists have been sorted in
    * ascending order by index. This implementation does *not* do the right
    * thing with boundary nodes: The inner product between an interior and a
    * boundary node is automatically zero, and no boundary condition comes in.
    *
    * But why should this introduce instability into the system? *Which* matrix
    * needs to be positive-definite?
    **/
double inner_product(int i, int j, node *elements, farray **coeff)
{
    cell *cp = elements[i].neighbors, *cq = elements[j].neighbors, *cr;
    double sum = 0, prod;
    int count, k, l;
    while (1)
        if (cp == NULL && cq == NULL) {
            if (i == j && interior_node(elements + i))
                sum += fa_ref(coeff[i], 0)*fa_ref(coeff[j], 0);
            return sum;
        }
        else if (cq == NULL || (cp != NULL && cp->index < cq->index)) {
            if (cp->index == j && interior_node(elements + j)) {
                prod = fa_ref(coeff[j], 0);
                for (l = 1, cr = elements[j].neighbors; cr != NULL; l++, cr = cr->next)
                if (cr->index == i) {
                    prod *= fa_ref(coeff[j], l);
                    break;
                }
                sum += prod;
            }
            cp = cp->next;
        }
        else if (cp == NULL || (cq != NULL && cp->index > cq->index)) {
            if (cq->index == i && interior_node(elements + i)) {
                prod = fa_ref(coeff[i], 0);
                for (l = 1, cr = elements[i].neighbors; cr != NULL; l++, cr = cr->next)
                if (cr->index == j) {
                    prod *= fa_ref(coeff[i], l);
                    break;
                }
                sum += prod;
            }
            cq = cq->next;
        }
        else {
                k = cp->index;
            if (interior_node(elements + k)) {
                prod = 1.0;
```

```
                for (count = 2, l = 1, cr = elements[k].neighbors;
                    cr != NULL && count > 0;
                    l++, cr = cr->next) {
                    if (cr->index == i) {
                        prod *= fa_ref(coeff[k], 1);
                        count--;
                    }
                    if (cr->index == j) {
                        prod *= fa_ref(coeff[k], l);
                        count--;
            }
        }
        sum += prod;
        }
        cp = cp->next;
        cq = cq->next;
    }
}
void check_symmetry(int size, node *elements, farray **coeff)
{
    cell *cp, *cq;
    double max = 0, diff;
    int i, j, k, count = 0, total = 0;
    for (i = 0; i < size; i++)
        if (interior_node(elements + i)) {
            for (j = 1, cp = elements[i].neighbors; cp != NULL; j++, cp = cp->next)
                if (interior_node(elements + cp->index)) {
                    for (k = 1, cq = elements[cp->index].neighbors;
                        cq != NULL;
                        k++, cq = cq->next)
                if (cq->index == i)
                    break;
            if (cq != NULL)
                        diff = fabs(fa_ref(coeff[i], j) - fa_ref(coeff[cp->index], k));
                    else
                        diff = fabs(fa_ref(coeff[i], j));
            if (diff > max)
                max = diff;
            }
            if (fa_ref(coeff[i], 0) > 0)
                count++;
            total++;
        }
    fprintf(stderr, "Maximum difference from symmetry: %.16lf.\n", max);
    fprintf(stderr, "%d out of %d diagonal entries are positive.\n",
                count, total);
}
```


## C.3.37 nstat.c

```
/*************
* nstat.c *
**************
Collects and prints neighborhood information on a gunk file.
*/
#include <stdio.h>
#include <math.h>
#include "gunk.h"
#include "stat.h"
main(int argc, char *argv[])
{
    int size;
    node *elements;
    FILE *fp;
    if (argc != 2) {
        fprintf(stderr, "Usage: %s input-file\n", argv[0]);
        exit(1);
    }
    fp = fopen(argv[1], "r");
    if (fp == NULL) {
        fprintf(stderr, "Error: Cannot open file \"%s\"\n", argv[1]);
        exit(2);
    }
    fscanf(fp, "%d", &size);
    elements = (node *)calloc(sizeof(node), size);
    read_elements(fp, size, elements);
    fclose(fp);
    nstat(stdout, size, elements);
    free_memory(size, elements);
    return 0;
}
```


## C.3.38 old2new.c

```
/***************
* old2new.c *
****************
```

Converts the old (ASCII) format of gunk files to the binary format.
*/
\#include <stdio.h>
\#include "gunk.h"

```
main(int argc, char *argv[])
{
    int size;
    node *elements;
    FILE *fp;
    if (argc != 3) {
        fprintf(stderr, "Usage: %s input-file output-file\n", argv[0]);
        exit(1);
    }
    fp = fopen(argv[1], "r");
    if (fp == NULL) {
        fprintf(stderr, "Error: Cannot open file \"%s\"\n", argv[1]);
        exit(1);
    }
    fscanf(fp, "%d", &size);
    fprintf(stderr, "Reading input file (%d nodes)...\n", size);
    elements = (node *)calloc(sizeof(node), size);
    old_read_elements(fp, size, elements);
    fclose(fp);
    fprintf(stderr, "Writing output file (%d nodes)...\n", size);
    fp = fopen(argv[2], "w");
    write_elements(fp, size, elements);
    fclose(fp);
    fprintf(stderr, "Freeing memory...\n");
    free_memory(size, elements);
    return 0;
}
```


## C.3.39 order.c

```
/************
* order.c *
*************
This is the order in which monomials are used in a polynomial
interpolation method for finite differencing.
*/
#include <stdio.h>
void walk(int, int *, int *);
main(int argc, char *argv[])
{
    int i, n, x, y;
    if (argc != 2) {
        fprintf(stderr, "Usage: %s n\n", argv[0]);
```

```
        exit(1);
    }
    sscanf(argv[1], "%d", &n);
    for (i = 0; i < n; i++) {
        walk(i, &x, &y);
        printf("%d: (%d, %d)\n", i, x, y);
    }
}
void walk(int n, int *x, int *y)
{
    int i = 0, j = 0, k;
    for (; n > 0; n--)
        if (i == j) {
            i++;
            j = 0;
        }
        else if (i > j) {
            k = i;
            i = j;
            j = k;
        }
        else {
            k = i + 1;
            i = j;
            j = k;
        }
    *x = i;
    *y = j;
}
```


## C.3.40 peek.c

```
/*************
* peek.c *
*************
```

Take a peek inside gunk.
*/
\#include <stdio.h>
\#include <math.h>
\#include "gunk.h"
void peek(int, node *, int);
void sort_neighbors(int, node *);
main(int argc, char *argv[])
\{
FILE *fp;
int size, $i$, node_count, $n$;
node *elements;

```
    if (argc < 2) {
        fprintf(stderr, "Usage: %s file-name [node1 node2 ...]\n", argv[0]);
        exit(1);
    }
    fp = fopen(argv[1], "r");
    if (fp == NULL) {
        fprintf(stderr, "Error: Cannot open file \"%s\"\n", argv[1]);
        exit(1);
    }
    fscanf(fp, "%d", &size);
    elements = (node *)calloc(sizeof(node), size);
    read_elements(fp, size, elements);
    fclose(fp);
    sort_neighbors(size, elements);
    node_count = argc - 2;
    for (i = 0; i < node_count; i++) {
        sscanf(argv[i + 2], "%d", &n);
        peek(size, elements, n);
    }
    free_memory(size, elements);
    return 0;
}
void peek(int size, node *elements, int node)
{
    cell *cp;
    int count;
    printf("\nNode %d:\n", node);
    if (node < 0 || node >= size) {
        printf("Out of range!\n\n");
        return;
    }
    printf("Position = (%lf, %lf)\n", elements[node].x, elements[node].y);
    printf("Value = %lf\n", elements[node].z);
    printf("Neighbors = (");
    cp = elements[node].neighbors;
    if (cp == NULL)
        printf(")");
    else {
        printf("%d", cp->index);
        for (count = 1, cp = cp->next; cp != NULL; count++, cp = cp->next)
            printf(" %d", cp->index);
        printf(")");
    }
```

```
    printf("\nCounted %d neighbors.\n", count);
    printf("Node type: ");
    if (interior_node(elements + node))
        printf("Interior.\n");
    else if (boundary_node(elements + node))
        printf("Boundary.\n");
    else
        printf("Unknown.\n");
    printf("\n");
}
void sort_neighbors(int size, node *elements)
{
    cell *cp;
    double *keys, val;
    int i, m, n, length, *cells, index;
    for (i = 0; i < size; i++)
        if (interior_node(elements + i)) {
            length = 0;
        for (cp = elements[i].neighbors; cp != NULL; cp = cp->next)
                length++;
        cells = (int *)calloc(sizeof(int), length);
        keys = (double *)calloc(sizeof(double), length);
        length = 0;
        for (cp = elements[i].neighbors; cp != NULL; cp = cp->next) {
            cells[length] = cp->index;
                keys[length] = -(cp->index);
                length++;
        }
        for (m = 1; m < length; m++) {
            val = keys[m];
                index = cells[m];
                for (n = m-1; keys[n] < val d& n >= 0; n--) {
                keys[n + 1] = keys[n];
                cells[n + 1] = cells[n];
                }
                keys[n + 1] = val;
                cells[n + 1] = index;
        }
            length = 0;
        for (cp = elements[i].neighbors; cp != NULL; cp = cp->next)
                cp->index = cells[length++];
        free(keys);
        free(cells);
    }
}
```


## C.3.41 peek_coeff.c

```
/******************
* peek_coeff.c *
```

*******************

Take a peek inside gunk.
*/
\#include <stdio.h>
\#include <math.h>
\#include "gunk.h"
\#include "farray.h"
void peek(int, node *, farray **, int);
void sort_neighbors(int, node *, farray **);
void read_coeff(FILE *, int, farray **) ;
main (int argc, char *argv[])
\{
farray **coeff;
FILE *fp;
int size, i, node_count, $n$;
node *elements;
if (argc < 3) \{
fprintf(stderr, "Usage: \%s gunk-file coeff-file [node1 node2 ...] $\operatorname{argv}[0]$ );
exit(1);
\}
$f p=f o p e n(\operatorname{argv}[1], \quad " r ") ;$
if ( $\mathrm{f} p==$ NULL) \{
fprintf(stderr, "Error: Cannot open file \"\%s\"\n", argv[1]); exit(1);
\}
fscanf(fp, "\% ${ }^{2}$, \&size);
elements $=$ (node $*$ )calloc(sizeof(node), size);
read_elements(fp, size, elements);
fclose(fp);
$f p=f o p e n(\operatorname{argv}[2], \quad " r ") ;$
if ( $f p==$ NULL) $\{$
fprintf(stderr, "Error: Cannot open file $\backslash " \% s \backslash " \ n ", ~ a r g v[2]) ;$ exit(1);
\}
fprintf(stderr, "Reading coefficients...\n");
coeff = (farray **) calloc(sizeof(farray *), size);
read_coeff(fp, size, coeff);
fclose(fp);
sort_neighbors(size, elements, coeff);
node_count $=\operatorname{argc}-3$;

```
    for (i = 0; i < node_count; i++) {
        sscanf(argv[i + 3], "%d", &n);
        peek(size, elements, coeff, n);
    }
    free_memory(size, elements);
    for (i = 0; i < size; i++)
        farray_free(coeff[i]);
    free(coeff);
    return 0;
}
void peek(int size, node *elements, farray **coeff, int node)
{
    cell *cp;
    int count;
    printf("\nNode %d:\n", node);
    if (node < O || node >= size) {
        printf("Out of range!\n\n");
        return;
    }
    printf("Position = (%lf, %lf)\n", elements[node].x, elements[node].y);
    printf("Value = %lf\n", elements[node].z);
    printf("Neighbors & coefficients:\n");
    if (coeff[node] != NULL) {
        printf("(%d\t%lf)\n", node, fa_ref(coeff[node], 0));
        for (count = 1, cp = elements[node].neighbors;
            cp != NULL;
            count++, cp = cp->next)
        printf("(%d\t%lf)\n", cp->index, fa_ref(coeff[node], count));
}
else {
    printf("(%d\tn/a)\n", node);
    for (count = 1, cp = elements[node].neighbors;
            cp != NULL;
            count++, cp = cp->next)
        printf("(%d\tn/a)\n", cp->index);
}
printf("\nCounted %d neighbors.\n", --count);
printf("Node type: ");
if (interior_node(elements + node))
    printf("Interior.\n");
else if (boundary_node(elements + node))
    printf("Boundary.\n");
else
    printf("Unknown.\n");
printf("\n");
```

\}

```
/**
    * Sort neighbor lists by index.
    **/
void sort_neighbors(int size, node *elements, farray **coeff)
{
    cell *cp;
    int i, j;
    void sort_without_coeff(node *);
    void sort_with_coeff(node *, farray *);
    for (i = 0; i < size; i++)
        if (coeff[i] == NULL)
            sort_without_coeff(elements + i);
        else
            sort_with_coeff(elements + i, coeff[i]);
}
void sort_without_coeff(node *node)
{
    cell *cp;
    int i, m, n, length, *key, index;
    for (length = 0, cp = node->neighbors; cp != NULL; length++, cp = cp->next);
    key = (int *)calloc(sizeof(int), length);
    for (i = 0, cp = node->neighbors; cp != NULL; i++, cp = cp->next)
        key[i] = cp->index;
    for (m = 1; m < length; m++) {
        index = key[m];
        for (n = m - 1; key[n] > index && n >= 0; n--)
            key[n + 1] = key[n];
        key[n + 1] = index;
    }
    for (i = 0, cp = node->neighbors; cp != NULL; i++, cp = cp->next)
        cp->index = key[i];
    free(key);
}
void sort_with_coeff(node *node, farray *coeff)
{
    cell *cp;
    double val, *new;
    int i, m, n, length, *key, index;
    length = fa_size(coeff) - 1;
    key = (int *)calloc(sizeof(int), length);
    new = (double *)calloc(sizeof(double), length);
    for (i = 0, cp = node->neighbors; cp != NULL; i++, cp = cp->next) {
        key[i] = cP->index;
```

```
        nev[i] = fa_ref(coeff, i + 1);
    }
    for (m = 1; m < length; m++) {
        index = key[m];
        val = fa_ref(coeff,m+1);
        for (n = m-1; key[n] > index && n >= 0; n--) {
            key[n + 1] = key[n];
            new[n + 1] = new[n];
        }
        key[n + 1] = index;
        new[n + 1] = val;
    }
    for (i = 0, cp = node->neighbors; cp != NULL; i++, cp = cp->next) {
        cp->index = key[i];
        fa_set(coeff, i + 1, new[i]);
    }
    free(key);
    free(new);
}
void read_coeff(FILE *fp, int size, farray **coeff)
{
    double val;
    int i, j, N;
    for (i = 0; i < size; i++) {
        getint(fp,&N);
        if (N > 0) {
            coeff[i] = farray_cons(N);
            getlf(fp, &val);
            fa_set(coeff[i], 0, val);
            for (j = 1; j < N; j++) {
            getlf(fp, &val);
            fa_set(coeff[i], j, val);
        }
        }
    }
}
```


## C.3.42 poly.c

```
/************
* poly.c *
*************
This is pretty much like approx, only it dumps the polynomial coefficients instead of the operator coefficients.
*/
```

```
#include <stdio.h>
#include <math.h>
#include "gunk.h"
#include "matrix.h"
#define CACHE_SIZE }10
void init_op(int, node *, farray **);
void gen_coeff(int, node *, farray **);
void check_coeff(int, farray **);
void lsqfit(int, node *, farray *);
void memo_walk(int, int *, int *);
void walk(int, int *, int *);
double monomial(double, double, int, int);
int term_count(int);
main(int argc, char *argv[])
{
    farray **coeff;
    int size, i, j, N;
    FILE *fp;
    node *elements;
    if (argc != 3) {
        fprintf(stderr, "Usage: %s input-file output-file\n",
                    argv[0]);
        exit(1);
    }
    fp = fopen(argv[1], "r");
    if (fp == NULL) {
        fprintf(stderr, "Error: Cannot open file \"%s\"\n", argv[1]);
        exit(2);
    }
    fscanf(fp, "%d", &size);
    elements = (node *)calloc(sizeof(node), size);
    fprintf(stderr, "Reading input file (%d nodes)...\n", size);
    read_elements(fp, size, elements);
    fclose(fp);
    fprintf(stderr, "Initializing differential operator coefficients...\n");
    coeff = (farray **)calloc(sizeof(farray *), size);
    init_op(size, elements, coeff);
    gen_coeff(size, elements, coeff);
    check_coeff(size, coeff);
    fp = fopen(argv[2], "w");
    fprintf(stderr, "Writing output file...\n");
    for (i = 0; i < size; i++) {
        N = fa_size(coeff[i]);
```

```
        fprintf(fp, "%d",N);
        for (j = 0; j < N; j++)
            fprintf(fp, " %.16lg", fa_ref(coeff[i], j));
        fprintf(fp, "\n");
    }
    fclose(fp);
    fprintf(stderr, "Freeing memory...\n");
    free_memory(size, elements);
    for (i = 0; i < size; i++)
        farray_free(coeff[i]);
    free(coeff);
    return 0;
}
void init_op(int size, node *elements, farray **coeff)
{
    cell *cp;
    int i, length;
    for (i = 0; i < size; i++)
        if (interior_node(elements + i)) {
            length = 0;
            for (cp = elements[i].neighbors; cp != NULL; cp = cp->next)
                length++;
            coeff[i] = farray_cons(length + 1);
        }
}
void gen_coeff(int size, node *elements, farray **coeff)
{
    int n;
    for (n = 0; n < size; n++)
        if (interior_node(elements + n))
            lsqfit(n, elements, coeff[n]);
}
void check_coeff(int size, farray **coeff)
{
    double sum;
    int n, i, N, count = 0;
    for (n = 0; n < size; n++) {
        N = fa_size(coeff[n]);
        if (N>0) {
            sum = 0.0;
            for (i = 1; i < N; i++)
            sum += fabs(fa_ref(coeff[n], i));
```

```
            if (sum > fabs(fa_ref(coeff[n], 0)))
                count++;
        }
    }
    if (count > 0)
        fprintf(stderr, "Warning: Matrix not diagonally dominant! (%d/%d)\n",
            count, size);
}
void memo_walk(int n, int *x, int *y)
{
    static int xi[CACHE_SIZE], yi[CACHE_SIZE], filled[CACHE_SIZE];
    static int first_time = 1;
    int k;
    if (n < CACHE_SIZE) {
        if (first_time) {
            for (k = 0; k < CACHE_SIZE; k++)
                filled[k] = 0;
            first_time = 0;
        }
        if (!filled[n]) {
            walk(n, xi + n, yi + n);
            filled[n] = 1;
        }
        *x = xi[n];
        *y = yi[n];
    }
    else
        walk(n, x, y);
}
void walk(int n, int *x, int *y)
{
    int i = 0, j = 0, k;
    for (; n > 0; n--)
        if (i == j) {
            i++;
            j = 0;
        }
        else if (i > j) {
            k = i;
            i = j;
            j = k;
        }
        else {
            k=i+1;
            i = j;
            j = k;
        }
    *x = i;
    *y = j;
}
```

```
double monomial(double x, double y, int i, int j)
{
    double product = 1.0;
    for (; i > 0; i--) product *= x;
    for (; j > 0; j--) product *= y;
    return product;
}
int term_count(int n)
{
    n = (int)ceil(sqrt(8.0*n + 1.0)/2.0-1.5);
    n = ((n + 1)*(n + 2))/2;
    return (n > 6) ? n : 6;
}
void lsqfit(int index, node *elements, farray *coeff)
{
    FILE *fp;
    static int print = 0, first_time = 0;
    cell *cp;
    double x = elements[index].x, y = elements[index].y, sum;
    farray *W;
    int i, j, M, N, p, q, other, bad_count = 0;
    matrix *U, *Ut, *V;
    N = fa_size(coeff);
    M = term_count(N);
    U = matrix_cons(M,N);
    V = matrix_cons(N,N);
    W = farray_cons(N);
    if (first_time) {
        fp = fopen("matrix", "ซ");
        fprintf(fp, "%.16lg", elements[index].z);
        for (cp = elements[index].neighbors; cp != NULL; cp = cp->next)
            fprintf(fp, " %.16lg", elements[cp->index].z);
        fprintf(fp, "\n");
    }
    for (i = 0; i < M; i++) {
        memo_walk(i, &p, &q);
        other = index;
        cp = elements[index].neighbors;
        for (j = 0; j < N; j++) {
            mset(U, i, j, monomial(elements[other].x - x, elements[other].y - y,
                                    p, q));
        if (first_time) fprintf(fp, "%.16lg ", mref(U, i, j));
            other = cp->index;
            cp = cp->next;
        }
```

```
    if (first_time) fprintf(fp, "\n");
}
Ut = transpose(U);
matrix_free(U);
U = Ut;
svdecomp(U, W, V); /* use _Numerical Recipes_ */
if (print) {
    printf("\nSingular values:\n");
    for (i = 0; i < N; i++)
        printf("%.16lg\n", fa_ref(W, i));
    printf("\nU =\n");
    mprint(stdout, U);
    printf("\nV =\n");
    mprint(stdout, V);
}
for (i = 0; i < N; i++)
    if (fabs(fa_ref(W, i)) < TOL)
        fa_set(W, i, 0.0);
    else {
        sum = elements[index].z*mref(U, 0, i);
            for (cp = elements[index].neighbors, j = 1;
                cp != NULL;
                cp = cp->next, j++)
            sum += elements[cp->index].z*mref(U, j, i);
        fa_set(W, i, sum/fa_ref(W, i));
    }
for (i = 0; i < N; i++) {
    sum = 0.0;
    for (j = 0; j < N; j++)
            sum += mref(V, i, j)*fa_ref(W, j);
    fa_set(coeff, i, sum);
}
if (print) {
    print--;
    printf("\nCoefficients:\n");
    printf("(%lf, %lf)\t%.16lg\n", 0.0, 0.0, fa_ref(coeff, 0));
    for (cp = elements[index].neighbors, i = 1; i < N; i++, cp = cp->next)
        printf("(%.16lg, %.161g)\t%.16lg\n",
            elements[cp->index].x - elements[index].x,
            elements[cp->index].y - elements[index].y,
                    fa_ref(coeff, i));
}
if (first_time) {
    first_time = 0;
```

```
        fclose(fp);
        fprintf(stderr, "Done dumping.\n");
    }
    matrix_free(U);
    matrix_free(V);
    farray_free(W);
}
```


## C.3.43 pot.c

```
/***********
* pot.c *
************
```

Prints the value of the potential at the specified point.
*/

```
#include <stdio.h>
#include "gunk.h"
main(int argc, char *argv[])
{
    double x, y;
    if (argc != 3) {
        fprintf(stderr, "Usage: %s x y\n", argv[0]);
        exit(1);
    }
    sscanf(argv[1], "%lf", &x);
    sscanf(argv[2], "%lf", ky);
    printf("%.16lf\n", potential(x, y));
    return 0;
}
```


## C.3.44 random.c

```
/* A translation of /scheme/src/runtime/random.scm. */
```

\#include <stdlib.h>
\#include <time.h>
\#define RANDOM_DECLS
\#include "random.h"
double frandom(void)
\{
return lrandom()/double_B;
\}
unsigned long lrandom(void)
\{
int index = random_state.index, index_s;

```
    unsigned long borrow = random_state.borrow;
    unsigned long *vector = random_state.vector;
    unsigned long t, val = vector[index];
    index_s = (index >= S) ? index - S : index + R_S;
    if (vector[index_s] >= val + borrow) {
        vector[index] = vector[index_s] - val - borrow;
        random_state.borrow = 0;
    }
    else {
        vector[index] = vector[index_s] + (B - val - borrow);
        random_state.borrow = 1;
    }
    if (random_state.index == R_1)
        random_state.index = 0;
    else
        random_state.index++;
    return val;
}
void initialize_random(void)
{
    int i, flag = 1;
    unsigned long *vector = random_state.vector;
    srand48(time(NULL) + 123456789);
    while (flag) {
        for (i = 0; i < R; i++)
            vector[i] = abs(mrand48());
        if ((vector[0] != 0) && (vector[0] != B_1))
            flag = 0;
        else
            for (i = 1; i < R; i++)
                if (vector[0] != vector[i]) {
                    flag = 0;
                    break;
                }
    }
    random_state.index = 0;
    random_state.borrow = 0;
    for (i = 0; i < TRANSIENT; i++)
        lrandom();
}
```


## C.3.45 relax.c

/************

* relax.c *
*************

```
A simple SOR program on gunk.
*/
#include <stdio.h>
#include <math.h>
#include "gunk.h"
#include "random.h"
/* const double omega = 1.9390892311242500; */
double relax(int, node *, unsigned long, double);
double random_relax(int, node *, unsigned long, double);
main(int argc, char *argv[])
{
    double omega = 1.0;
    int size;
    unsigned long count;
    FILE *fp;
    node *elements;
    if (argc != 4 && argc != 5) {
        fprintf(stderr,
                        "Usage: %s input-file number-of-iterations output-file [omega]\n",
                    argv[0]);
        exit(1);
    }
    fp = fopen(argv[1], "r");
    sscanf(argv[2], "%u", &count);
    if (fp == NULL) {
        fprintf(stderr, "Error: Cannot open file \"%s\"\n", argv[1]);
        exit(2);
    }
    fscanf(fp, "%d", &size);
    elements = (node *)calloc(sizeof(node), size);
    fprintf(stderr, "Reading input file (%d nodes)...\n", size);
    read_elements(fp, size, elements);
    fclose(fp);
    fprintf(stderr, "Running %u iterations...\n", count);
    if (argc == 5)
        sscanf(argv[4], "%lf", &omega);
    fprintf(stderr, "Maximum residual = %.16lf\n",
            relax(size, elements, count, omega));
    fp = fopen(argv[3], "w");
    fprintf(stderr, "Writing output file...\n");
    write_elements(fp, size, elements);
    fclose(fp);
    fprintf(stderr, "Freeing memory...\n");
```

```
    free_memory(size, elements);
    return 0;
}
double relax(int size, node *elements, unsigned long count, double omega)
{
    cell *cp;
    int n, i, length;
    double sum, residual, max_residual;
    for (n = 0; n < count; n++) {
        max_residual = 0.0;
        for (i = 0; i < size; i++)
            if (interior_node(elements + i)) {
                sum = 0.0;
                length = 0;
                for (cp = elements[i].neighbors; cp != NULL; cp = cp->next) {
                    sum += elements[cp->index].z;
                length++;
                }
            if (length > 0) {
                    residual = sum - length*elements[i].z;
                    elements[i].z += omega*residual/length;
                        if (fabs(residual) > max_residual)
                        max_residual = fabs(residual);
                }
            }
    }
    return max_residual;
}
double random_relax(int size, node *elements, unsigned long count,
                    double omega)
{
    cell *cp;
    int n, i, length, index;
    double sum, residual, max_residual;
    initialize_random();
    for (n = 0; n < count; n++) {
        max_residual = 0.0;
        for (i = 0; i < size; i++) {
            index = lrandom()%size;
                if (interior_node(elements + index)) {
                    sum = 0.0;
            length = 0;
                for (cp = elements[index].neighbors; cp != NULL; cp = cp->next) {
                    sum += elements[cp->index].z;
                    length++;
```

```
                }
                if (length > 0) {
                    residual = sum - length*elements[index].z;
                    elements[index].z += omega*residual/length;
                        if (fabs(residual) > max_residual)
                max_residual = fabs(residual);
            }
        }
        }
    }
    return max_residual;
}
```


## C.3.46 repos.c

```
/************
* repos.c *
*************
```

This program moves each node closer to the closest regular grid point.
Doesn't do much.
*/
\#include <stdio.h>
\#include <math.h>
\#include "gunk.h"
void reposition(int, node $*$, int);
double grid_direction(double, double, int);
double round(double);
main(int argc, char *argv[])
\{
int size, grid_size;
FILE *fp;
node *elements;
if (argc != 3) \{
fprintf(stderr, "Usage: \%s input-file output-file\n",
$\operatorname{argv}[0]$ );
exit(1);
\}
$f p=$ fopen (argv[1], "r");
if ( $f p==$ NULL) $\{$
fprintf(stderr, "Error: Cannot open file \"\%s\"\n", argv[1]);
exit(1);
\}
fscanf(fp, "\%d", \&size);
elements $=$ (node $*$ )calloc (sizeof(node), size);

```
    fprintf(stderr, "Reading input file (%d nodes)...\n", size);
    read_elements(fp, size, elements);
    fclose(fp);
    grid_size = (int)sqrt(size);
    fprintf(stderr, "Repositioning elements (%dx%d)...\n", grid_size, grid_size);
    reposition(size, elements, grid_size);
    fp = fopen(argv[2], "w");
    fprintf(stderr, "Writing output file...\n");
    write_elements(fp, size, elements);
    fclose(fp);
    fprintf(stderr, "Freeing memory...\n");
    free_memory(size, elements);
    return 0;
}
void reposition(int size, node *elements, int grid_size)
{
    cell *cp;
    double x, y, dist, max_dist, radius = 1.0/(grid_size - 1), angle;
    int index;
    for (index = 0; index < size; index++)
        if (interior_node(elements + index)) {
            max_dist = 0.0;
            angle = grid_direction(elements[index].x, elements[index].y, grid_size);
            for (cp = elements[index].neighbors; cp != NULL; cp = cp->next) {
                dist = distance(elements + index, elements + cp->index);
                max_dist = (dist > max_dist) ? dist : max_dist;
            }
            elements[index].x += (radius - max_dist)*cos(angle);
            elements[index].y += (radius - max_dist)*sin(angle);
        }
}
double grid_direction(double x, double y, int grid_size)
{
    double dx, dy, radius = 1.0/(grid_size - 1);
    dx = round(x/radius)*radius - x;
    dy = round(y/radius)*radius - y;
    if (dx == 0.0)
        if (dy > 0.0)
            return pi/2;
        else
            return -pi/2;
    else
        if (dx > 0.0)
            return atan(dy/dx);
        else
            return atan(dy/dx) + pi;
}
```

```
double round(double x)
{
    double flr = floor(x);
    if (x - flr >= 0.5)
        return flr + 1.0;
    else
        return flr;
}
```


## C.3.47 reset_val.c

```
/****************
* reset_val.c *
*****************
```

Reset the nodal values in a gunk system.
*/

```
#include <stdio.h>
#include <math.h>
#include "random.h"
#include "gunk.h"
void reset_val(int, node *, int);
main(int argc, char *argv[])
{
    FILE *fp;
    int size;
    node *elements;
    if (argc > 4 || argc < 3) {
        fprintf(stderr, "Usage: %s input-file output-file [-random]\n", argv[0]);
        exit(1);
    }
    fp = fopen(argv[1], "r");
    if (fp == NULL) {
        fprintf(stderr, "Error: Cannot open file \"%s\"\n", argv[1]);
        exit(1);
    }
    fscanf(fp, "%d", &size);
    fprintf(stderr, "Reading input file (%d nodes)...\n", size);
    elements = (node *)calloc(sizeof(node), size);
    read_elements(fp, size, elements);
    fclose(fp);
    if (argc == 4 && strcmp(argv[3], "-random") == 0) {
        fprintf(stderr, "Randomizing nodal values...\n");
        reset_val(size, elements, 1);
    }
    else {
```

```
        fprintf(stderr, "Resetting nodal values...\n");
        reset_val(size, elements, 0);
    }
    fp = fopen(argv[2], "พ");
    if (fp == NULL) {
        fprintf(stderr, "Error: Cannot open file \"%s\"\n", argv[1]);
        exit(1);
    }
    fprintf(stderr, "Writing output file...\n");
    write_elements(fp, size, elements);
    free_memory(size, elements);
    return 0;
}
void reset_val(int size, node *elements, int flag)
{
    int i;
    if (flag) {
        initialize_random();
        for (i = 0; i < size; i++)
            if (interior_node(elements + i))
                elements[i].z = frandom();
    }
    else
        for (i = 0; i < size; i++)
            if (interior_node(elements + i))
                elements[i].z = 0;
}
```


## C.3.48 rref.c

```
/************
* rref.c *
*************
```

Tests the matrix routines by putting the hard-wired matrix into
reduced row-echelon form.
*/

```
#include <stdio.h>
```

\#include "matrix.h"
main()
\{
matrix $*$ mat $=$ matrix_cons $(3,4)$;
mset(mat, 0, 0, 1.0);
mset(mat, 0, 1, 2.0);

```
    mset(mat, 0, 2, 1.2);
    mset(mat, 1, 0, 3.0);
    mset(mat, 1, 1, 4.0);
    mset(mat, 1, 2, 1.2);
    mset(mat, 2, 0, 1.0);
    mset(mat, 2, 1, 4.0);
    mset(mat, 2, 2, 1.2);
    mset(mat, 0, 3, 1.0);
    mset(mat, 1, 3, 1.0);
mset(mat, 2, 3, 1.0);
printf("Before:\n");
mprint(stdout, mat);
rref(mat);
printf("After:\n");
mprint(stdout, mat);
matrix_free(mat);
return 0;
}
```


## C.3.49 sample.c

```
/***************
* sample.c *
```

**************

Fix four neighbor nodes and move center node to see how far we get from a diagonally dominant matrix.
*/

```
#include <stdio.h>
#include <math.h>
#include "gunk.h"
#include "matrix.h"
#define CACHE_SIZE }10
#define SIZE 5
void print_coeff(FILE *, farray *);
void gen_coeff(double, double, farray *);
void add_cell(node *, int, int);
void lsqfit(int, node *, farray *, int);
void memo_walk(int, int *, int *);
void walk(int, int *, int *);
double monomial(double, double, int, int);
double laplace_monomial(double, double, int, int);
main(int argc, char *argv[])
{
    double delta, sum, offset;
```

```
    farray *coeff = farray_cons(5);
    int i, j, k, n;
    if (argc != 2) {
    fprintf(stderr, "Usage: %s n\n", argv[0]);
    exit(1);
}
sscanf(argv[1], "%d", &n);
    delta = 10.0/n;
    offset = -5.0;
    printf("%d\n", n + 1);
    for (i = 0; i <= n; i++) {
        for (j = 0; j <= n; j++) {
            gen_coeff(i*delta + offset, j*delta + offset, coeff);
            sum = fabs(fa_ref(coeff, 0));
            for (k = 1; k < SIZE; k++)
            sum -= fabs(fa_ref(coeff, k));
            printf("%.16lg ", sum);
        }
        printf("\n");
    }
    farray_free(coeff);
    return 0;
}
void gen_coeff(double x, double y, farray *coeff)
{
    int i;
    node *elements = (node *)calloc(sizeof(node), SIZE);
    for (i = 0; i < SIZE; i++) {
        elements[i].neighbors = NULL;
        elements[i].z = 0.0;
    }
    for (i = 1; i < SIZE; i++) {
        add_cell(elements, 0, i);
        elements[i].x = sin(i*pi/2.0);
        elements[i].y = - cos(i*pi/2.0);
        set_boundary_node(elements + i);
    }
    elements[0].x = x;
    elements[0].y = y;
    set_interior_node(elements);
    lsqfit(0, elements, coeff, 6);
    free_memory(SIZE, elements);
}
void print_coeff(FILE *fp, farray *coeff)
{
```

```
    fprintf(fp, "\t\t%.8lf\n\n", fa_ref(coeff, 3));
    fprintf(fp, "%.8lf\t%.8lf\t%.8lf\n\n",
            fa_ref(coeff, 4), fa_ref(coeff, 0), fa_ref(coeff, 2));
    fprintf(fp, "\t\t%.8lf\n", fa_ref(coeff, 1));
}
void add_cell(node *elements, int i, int j)
{
    cell *cp = (cell *)malloc(sizeof(cell));
    cp->next = elements[i].neighbors;
    elements[i].neighbors = cp;
    cp->index = j;
}
void memo_walk(int n, int *x, int *y)
{
    static int xi[CACHE_SIZE], yi[CACHE_SIZE], filled[CACHE_SIZE];
    static int first_time = 1;
    int k;
    if (n < CACHE_SIZE) {
        if (first_time) {
            for (k = 0; k < CACHE_SIZE; k++)
                filled[k] = 0;
            first_time = 0;
        }
        if (!filled[n]) {
            walk(n, xi + n, yi + n);
            filled[n] = 1;
        }
        *x = xi[n];
        *y = yi[n];
    }
    else
        walk(n, x, y);
}
void walk(int n, int *x, int *y)
{
    int i = 0, j = 0, k;
    for (; n > 0; n--)
        if (i == j) {
            i++;
            j = 0;
        }
        else if (i > j) {
            k = i;
            i = j;
            j = k;
        }
        else {
            k = i + 1;
            i = j;
            j = k;
```

```
        }
    *x = i;
    *y = j;
}
double monomial(double }x\mathrm{ , double }y\mathrm{ , int i, int j)
{
    double product = 1.0;
    for (; i > 0; i--) product *= x;
    for (; j > 0; j--) product *= y;
    return product;
}
double laplace_monomial(double x, double y, int i, int j)
{
    double xprod = i*(i - 1), yprod = j*(j - 1);
    int k;
    for (k = 2; k < i; k++) xprod *= x;
    for (k = 0; k < j; k++) xprod *= y;
    for (k = 2; k < j; k++) yprod *= y;
    for (k = 0; k < i; k++) yprod *= x;
    return xprod + yprod;
}
int term_count(int n)
{
    n = (int)ceil(sqrt(8.0*n + 1.0)/2.0 - 1.5);
    return ((n + 1)*(n + 2))/2;
}
void lsqfit(int index, node *elements, farray *coeff, int M)
{
    cell *cp;
    double x = elements[index].x, y = elements[index].y, sum;
    farray *W;
    int i, j, N, p, q, other;
    matrix *U, *V;
    N = fa_size(coeff);
    U = matrix_cons(M,N);
    V = matrix_cons(N,N);
    W = farray_cons(N);
    for (i = 0; i < M; i++) {
        memo_walk(i, &p, &q);
        other = index;
        cp = elements[index].neighbors;
        for (j = 0; j < N; j++) {
            mset(U, i, j, monomial(elements[other].x - x, elements[other].y - y,
                                    p, q));
            other = cp->index;
            cp = cp->next;
        }
```

```
    }
    svdecomp(U, W, V); /* use _Numerical Recipes_ */
    for (i = 0; i < N; i++)
        if (fabs(fa_ref(W, i)) < TOL)
            fa_set(W, i, 0.0);
        else {
            sum = 0.0;
            for (j = 0; j < M; j++) {
                memo_walk(j, &p, &q);
                sum += laplace_monomial (x, y, p, q)*mref(U, j, i);
            }
            fa_set(W, i, sum/fa_ref(W, i));
        }
    for (i = 0; i < N; i++) {
        sum = 0.0;
        for (j = 0; j < N; j++)
        sum += mref(V, i, j)*fa_ref(W, j);
    fa_set(coeff, i, sum);
    }
    matrix_free(U);
    matrix_free(V);
    farray_free(W);
}
```


## C.3.50 scramble.c

```
/*****************
* scramble.c *
*****************
```

Randomize the node positions without changing the connections. (I don't
remember what good this does.)
*/
\#include <stdio.h>
\#include "gunk.h"
\#include "random.h"
void scramble(int, node *);
main(int argc, char *argv[])
\{
int size;
node *elements;
FILE *fp;
if (argc $!=2$ ) $\{$

```
        fprintf(stderr, "Usage: %s input-file\n", argv[0]);
        exit(1);
    }
    fp = fopen(argv[1], "r");
    if (fp == NULL) {
        fprintf(stderr, "Error: Cannot open file \"%s\"\n", argv[1]);
        exit(2);
    }
    fscanf(fp, "%d", &size);
    elements = (node *)calloc(sizeof(node), size);
    read_elements(fp, size, elements);
    fclose(fp);
    scramble(size, elements);
    fp = fopen(argv[1], "w");
    write_elements(fp, size, elements);
    fclose(fp);
    free_memory(size, elements);
    return 0;
}
void scramble(int size, node *elements)
{
    int i;
    initialize_random();
    for (i = 0; i < size; i++)
        if (interior_node(elements + i)) {
            elements[i].x = frandom();
            elements[i].y = frandom();
        }
}
```


## C.3.51 slice.c

## /************

* slice.c *
*********れ***

Does the same thing as list2grid, only this plots the error.
*/

```
#include <stdio.h>
#include <math.h>
#include "gunk.h"
void slice(FILE *, int, node *, int);
#define CALC_ERROR
```

```
main(int argc, char *argv[])
{
    int grid_size, size;
    node *elements;
    FILE *fp;
    if (argc != 3) {
        fprintf(stderr, "Usage: %s input-file grid-size\n", argv[0]);
        exit(1);
    }
    fp = fopen(argv[1], "r");
    sscanf(argv[2], "%d", &grid_size);
    if (fp == NULL) {
        fprintf(stderr, "Error: Cannot open file \"%s\"\n", argv[1]);
        exit(2);
    }
    fscanf(fp, "%d", &size);
    fprintf(stderr, "Reading file (%d nodes)...\n", size);
    elements = (node *)calloc(sizeof(node), size);
    read_elements(fp, size, elements);
    fclose(fp);
    fprintf(stderr, "Printing output...\n");
    slice(stdout, size, elements, grid_size);
    fprintf(stderr, "Freeing memory...\n");
    free_memory(size, elements);
    return 0;
}
void slice(FILE *fp, int size, node *elements, int grid_size)
{
    double *value, *count, x, y;
    int i, j, n;
    value = (double *)calloc(sizeof(double), SQUARE(grid_size));
    count = (double *)calloc(sizeof(double), SQUARE(grid_size));
    for (n = 0; n < size; n++)
        if (interior_node(elements + n)) {
        x = elements[n].x;
        y = elements[n].y;
        i = (int)(x*grid_size);
        j = (int)(y*grid_size);
        if (i == grid_size) i--;
        if (j == grid_size) j--;
        *(count + i*grid_size + j) += 1.0;
#ifdef CALC_ERROR
            *(value + i*grid_size + j) += fabs(elements[n].z - potential(x, y));
#else
        *(value + i*grid_size + j) += elements[n].z;
#endif
    }
```

```
    fprintf(fp, "%d\n", grid_size);
    for (j = 0; j < grid_size; j++) {
        for (i = 0; i < grid_size; i++) {
            n = *(count + i*grid_size + j);
            if (n > 0.0)
                fprintf(fp, "%.16lg ", *(value + i*grid_size + j)/n);
            else
            fprintf(fp, "%.16lg ", 0.0);
        }
    fprintf(fp, "\n");
}
    free(value);
    free(count);
}
```


## C.3.52 spect.c

## /************

* spect.c *
*************

Attempts to estimate the spectral radius using a random vector. Doesn't quite work.
*/

```
#include <stdio.h>
#include <math.h>
#include "matrix.h"
#include "random.h"
int get_size(FILE *);
void get_matrix(FILE *, matrix *);
double spectral(matrix *);
main(int argc, char *argv[])
{
    double r;
    int size;
    matrix *mat;
    FILE *fp;
    if (argc != 2) {
        fprintf(stderr, "Usage: %s input-file\n", argv[0]);
        exit(1);
    }
    fp = fopen(argv[1], "r");
    if (fp == NULL) {
        fprintf(stderr, "Error: Unable to open file \"%s\"\n", argv[1]);
```

```
        exit(2);
    }
    size = get_size(fp);
    rewind(fp);
    if (size > 0) {
        initialize_random();
        mat = matrix_cons(size, size);
        get_matrix(fp, mat);
        r = spectral(mat);
        printf("The spectral radius is at least %.16lg.\n", r);
    }
    else
        fprintf(stderr, "Error: Invalid input file\n");
    fclose(fp);
    matrix_free(mat);
    return 0;
}
double spectral(matrix *mat)
{
    double sum, mag_u = 0.0, mag_v = 0.0, val;
    int i, j, size = mat->nrows;
    matrix *u = matrix_cons(size, 1);
    for (i = 0; i < size; i++) {
        val = 2*frandom() - 1;
        mset(u, i, 0, val);
        mag_u += val*val;
    }
    for (i = 0; i < size; i++) {
        sum = 0.0;
        for (j = 0; j < size; j++)
            sum += mref(mat, i, j)*mref(u, j, 0);
        mag_v += sum*sum;
    }
    matrix_free(u);
    return sqrt(mag_v/mag_u);
}
int get_size(FILE *fp)
{
    double x;
    int N;
    for (N = 0; fscanf(fp, "%lf", &x) != EOF; N++);
    x = sqrt(N);
    if (x == floor(x))
        return x;
    else
        return 0;
}
```

```
void get_matrix(FILE *fp, matrix *mat)
{
    double x;
    int m = mat->nrows, n = mat->ncols, i, j;
    for (i = 0; i < m; i++)
        for (j = 0; j < n; j++) {
            fscanf(fp, "%1f", &x);
            mset(mat, i, j, x);
        }
}
```


## C.3.53 sqstat.c

```
/**************
* sqstat.c *
***************
```

Collects and prints error information, etc., about a gunk file.
*/
\#include <stdio.h>
\#include <math.h>
\#include "gunk.h"
\#include "stat.h"
main (int argc, char *argv[])
\{
int size;
node *elements;
FILE *fp;
if (argc != 2) \{
fprintf(stderr, "Usage: \%s input-file\n", argv[0]);
exit(1);
$\}$
fp = fopen(argv[1], "r");
if ( $f p==N U L L$ ) $\{$
fprintf(stderr, "Error: Cannot open file \"\%s\"\n", argv[1]);
exit(2);
\}
fscanf(fp, "\%d", \&size);
elements = (node *) calloc(sizeof(node), size);
read_elements(fp, size, elements);
fclose(fp);
sqstat(stdout, size, elements);
free_memory(size, elements);
return 0 ;
\}

## C.3.54 stat.c

## /*********** <br> * stat.c * <br> ************

Routines used to collect statistics about gunk files.
*/

```
#include <stdio.h>
#include <math.h>
#include "gunk.h"
void sqstat(FILE *fp, int size, node *elements)
{
    cell *cp;
    double abs_sum = 0.0;
    node abs_max, abs_min;
    double rel_sum = 0.0;
    node rel_max, rel_min;
    double pot, err, x, y;
    int n, count = 0, j;
    abs_max.z = -1.0;
    abs_min.z = 1e16;
    rel_max.z = 0.0;
    rel_min.z = 1e16;
    for (n = 0; n < size; n++) {
        if (interior_node(elements + n)) {
        count++;
        x = elements[n].x;
        y = elements[n].y;
        pot = potential(x, y);
        err = fabs(elements[n].z - pot);
        if (err > abs_max.z) {
            abs_max.x = x;
            abs_max.y = y;
            abs_max.z = err;
        }
            if (err < abs_min.z) {
                abs_min.x = x;
                abs_min.y = y;
                abs_min.z = err;
            }
            abs_sum += err;
            if (fabs(pot) > 0.0) {
                err /= fabs(pot);
                rel_sum += err;
                if (err > rel_max.z) {
```

```
            rel_max.x = x;
            rel_max.y = y;
            rel_max.z = err;
            }
            if (err < rel_min.z) {
                rel_min.x = x;
                rel_min.y = y;
                rel_min.z = err;
            }
        }
        }
    }
    fprintf(fp, "Absolute error:\n");
    fprintf(fp, " Maximum = %.16lf (x = %.16lf, y = %.16lf)\n",
        abs_max.z, abs_max.x, abs_max.y);
    fprintf(fp, " Minimum = %.16lf (x = %.16lf, y = %.16lf)\n",
            abs_min.z, abs_min.x, abs_min.y);
    if (count > 0)
        fprintf(fp, " Average = %.16lf\n", abs_sum/count);
    fprintf(fp, "Relative error:\n");
    fprintf(fp, " Maximum = %.16lf (x = %.16lf, y = %.16lf)\n",
        rel_max.z, rel_max.x, rel_max.y);
    fprintf(fp, " Minimum = %.16lf (x = %.16lf, y = %.16lf)\n",
        rel_min.z, rel_min.x, rel_min.y);
    if (count > 0)
        fprintf(fp, " Average = %.16lf\n", rel_sum/count);
    fprintf(fp, "\n");
}
void nstat(FILE *fp, int size, node *elements)
{
    cell *cp;
    double dist_max = 0.0, dist_min = 1.0, dist, dist_sum;
    int n, count = 0, n_count,n_max = 0, n_sum = 0, n_min = size + 1;
    for (n = 0; n < size; n++) {
        if (interior_node(elements + n)) {
            count++;
            n_count = 0;
            for (cp = elements[n].neighbors; cp != NULL; cp = cp->next)
            if (interior_node(elements + cp->index)) {
                        n_count++;
                        dist = distance(elements + cp->index, elements + n);
                        dist_sum += dist;
                        if (dist > dist_max)
                        dist_max = dist;
                    if (dist < dist_min)
                        dist_min = dist;
            }
            if (n_count > n_max)
                n_max = n_count;
```

```
        if (n_count < n_min)
            n_min = n_count;
        n_sum += n_count;
    }
}
    fprintf(fp, "File contains %d interior nodes (out of %d):\n",
        count, size);
    fprintf(fp," Maximum number of neighbors is %d.\n", n_max);
    fprintf(fp, " Minimum number of neighbors is %d.\n", n_min);
    fprintf(fp, " Average number of neighbors is %.16lf\n",
        (double)n_sum/count);
    fprintf(fp, "Internodal distance:\n");
    fprintf(fp, " Maximum = %.16lf\n", dist_max);
    fprintf(fp, " Minimum = %.16lf\n", dist_min);
    fprintf(fp, " Average = %.16lf\n", dist_sum/n_sum);
    fprintf(fp, "\n");
}
```


## C.3.55 walk.c

```
/************
* walk.c *
*************
```

An attempt to use random walks to solve Laplace's equation.
(See Adams/Guillemin, _Measure theory and probability_, for a discussion of
this application of random walks to the discrete Dirichlet problem.)
*/

```
#include <stdio.h>
#include "gunk.h"
#include "random.h"
#include "stat.h"
void dirichlet(int, node *, int);
main(int argc, char *argv[])
{
    int size, count;
    FILE *fp;
    node *elements;
    if (argc != 4) {
        fprintf(stderr, "Usage: %s input-file number-of-walks/node output-file\n",
                        argv[0]);
        exit(1);
    }
    fp = fopen(argv[1], "r");
    sscanf(argv[2], "%d", &count);
```

```
    if (fp == NULL) {
    fprintf(stderr, "Error: Cannot open file \"%s\"\n", argv[1]);
    exit(2);
    }
    fscanf(fp, "%d", &size);
    elements = (node *)calloc(sizeof(node), size);
    fprintf(stderr, "Reading input file (%d nodes)...\n", size);
    read_elements(fp, size, elements);
    fclose(fp);
    fprintf(stderr, "Calling Dirichlet...\n", count);
    dirichlet(size, elements, count);
    fp = fopen(argv[3], "v");
    fprintf(stderr, "Writing output file...\n");
    write_elements(fp, size, elements);
    fclose(fp);
    fprintf(stderr, "Computing statistics...\n");
    sqstat(stdout, size, elements);
    fprintf(stderr, "Freeing memory...\n");
    free_memory(size, elements);
    return 0;
}
void dirichlet(int size, node *elements, int count)
{
    cell *cp;
    double sum;
    int n, i, j, node, *neighbor_count;
    initialize_random();
    neighbor_count = (int *)calloc(sizeof(int), size);
    for (n = 0; n < size; n++) {
        for (i = 0, cp = elements[n].neighbors; cp != NULL; cp = cp->next)
            i++;
        neighbor_count[n] = i;
    }
    for (n = 0; n < size; n++)
        if (interior_node(elements + n)) {
        sum = 0.0;
        for (i = 0; i < count; i++) {
            for (node = n; interior_node(elements + node); node = cp->index) {
                cp = elements[node].neighbors;
                for (j = lrandom()%neighbor_count[node]; 0 < j; j--)
                cp = cp->next;
            }
            sum += elements[node].z;
        }
```


## elements[n].z = sum/count;

 \}free(neighbor_count);
\}

## Bibliography

[1] Harold Abelson, Tom Knight, and Gerald Jay Sussman. Amorphous Computing (draft). MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, 1995.
[2] Harold Abelson and Gerlad Jay Sussman with Julie Sussman. Structure and Interpretation of Computer Programs. The MIT Press, Cambridge, Massachusetts, 2 edition, 1996.
[3] Lars Ahlfors. Complex Analysis. McGraw-Hill, New York, 3rd edition, 1979.
[4] V. I. Arnol'd. Mathematical Methods of Classical Mechanics. Springer-Verlag, New York, 2 edition, 1989.
[5] T. Balderes. Finite element method. McGraw-Hill Encyclopedia of Science 8 Technology, 1992.
[6] C. Bradford Barber, David P. Dobkin, and Hannu Huhdanpaa. The quickhull algorithm for convex hulls. Submitted to ACM Transactions on Mathematical Software, January 9, 1995.
[7] G. Chesshire and W. D. Henshaw. Composite overlapping meshes for the solution of partial differential equations. Journal of Computational Physics, 90:1-64, 1990.
[8] James J. Clark, Matthew R. Palmer, and Peter D. Lawrence. A transformation method for the reconstruction of functions from nonuniformly spaced samples. IEEE Transactions on Acoustics, Speech, and Signal Processing, 33(4):1151-1165, October 1985.
[9] William Clinger and Jonathan Rees, editors. Revised ${ }^{4}$ Report on the Algorithmic Language Scheme. MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, 1991.
[10] Richard Courant and David Hilbert. Methods of Mathematical Physics, Volume II: Partial Differential Equations. John Wiley \& Sons, New York, 1989.
[11] C. Armando Duarte and J. Tinsley Oden. $H-p$ clouds-an $h-p$ meshless method. $N u$ merical Methods for Partial Differential Equations, 12(6):673-705, November 1996.
[12] Bjorn Engquist and Andrew Majda. Absorbing boundary conditions for the numerical simulation of waves. Mathematics of Computation, 31(139):629-651, July 1977.
[13] D. Fox and C. Pucci. The Dirichlet problem for the wave equation. Ann. Mat. Pura Appl., 46:155-182, 1958.
[14] Victor W. Guillemin and Alan Pollack. Differential Topology. Prentice-Hall, Englewood Cliffs, New Jersey, 1974.
[15] Hermann A. Haus and James R. Melcher. Electromagnetic Fields and Energy. PrenticeHall, Englewood Cliffs, New Jersey, 1989.
[16] Fritz John. Partial Differential Equations. Springer-Verlag, New York, 4th edition, 1981.
[17] Claes Johnson. Numerical Solutions of Partial Differential Equations by the Finite Element Method. Cambridge University Press, New York, 1987.
[18] James R. Munkres. Elementary Differential Topology. Princeton University Press, Princeton, New Jersey, 1966.
[19] James R. Munkres. Topology: A first course. Prentice-Hall, Englewood Cliffs, New Jersey, 1974.
[20] James R. Munkres. Elements of Algebraic Topology. Addison-Wesley, Reading, Massachusetts, 1984.
[21] James R. Munkres. Analysis on Manifolds. Addison-Wesley, Reading, Massachusetts, 1991.
[22] L. E. Payne. Improperly posed problems in partial differential equations. Regional Conference Series in Applied Mathematics, SIAM, Philadelphia, Pennsylvania, 1975.
[23] N. Anders Petersson. An algorithm for constructing overlapping grids. Submitted to SIAM J. Sci. Comput., March 16, 1997.
[24] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. Numerical Recipes: The Art of Scientific Computing. Cambridge University Press, 1986.
[25] Bernard F. Schutz. A First Course in General Relativity. Cambridge University Press, New York, 1990.
[26] Rafael Sorkin. Time-evolution problem in Regge calculus. Physical Review D, 12(2):385-397, 15 July 1975.
[27] Robert Vichnevetsky. Computer Methods for Partial Differential Equations, Volume 1: Elliptic Equations and the Finite-Element Method. Prentice-Hall, New Jersey, 1981.
[28] Frank W. Warner. Foundations of Differentiable Manifolds and Lie Groups. SpringerVerlag, New York, 1983.


[^0]:    ${ }^{1}$ Often referred to as PDEs for short, just as ordinary differential equations are ODEs.

[^1]:    ${ }^{1}$ The ordinary pendulum, often used to illustrate important physical concepts, is not complicated enough geometrically to bring out the difficulties that manifolds were invented to handle.

[^2]:    ${ }^{2}$ In theory, points in an abstract space need not necessarily be points in a Euclidean space. They can also be classes of matrices or other abstract mathematical structures.
    ${ }^{3}$ In the present setting, the sets $W_{i}$ are required to be open subsets of $V_{i}$ (and hence of $R^{n}$ ). An alternative is to require the subsets $U_{i}$ of the abstract space $M$ to be open, but to define what that means requires some knowledge of general topology (which is not assumed here).
    ${ }^{4}$ It is easy to check that compatibility of charts is transitive, that is, if $C_{1}$ and $C_{2}$ are compatible charts,

[^3]:    ${ }^{6}$ For example, the function $f$ can be a measure of how poorly the chart behaves at $p$, such as how close the procedure $\phi \circ \phi^{-1}$ comes to being the identity map at $p$ and so on. This can be useful in integrating ODEs on manifolds.
    ${ }^{7}$ Note that this is only a restriction on our computational representations of manifolds, not on differentiable manifolds in general. A manifold, in theory, can have an infinite number of charts covering a given point. One should be careful to distinguish between differentiable manifolds, which are theoretical constructs, and their computational representations.

[^4]:    ${ }^{8}$ In theory, these equivalence classes can potentially be uncountably infinite sets. However, the local finiteness requirement in §2.1.2 forces such equivalence classes to be finite, and hence they are representable computationally. These can still be rather large sets, though, if many charts cover a given point.
    ${ }^{9}$ The result that every abstract $n$-manifold can be imbedded as a subspace of some Euclidean space $R^{N}$ is known as the Whitney imbedding theorem. Whitney also showed that there always exists an imbedding such that $N \leq 2 n$. However, the proof of this theorem requires some rather complicated constructions and hence such imbeddings almost never provide much insight into how one could visualize manifolds.

[^5]:    ${ }^{10}$ The problem is that the differentiation of functions in our Scheme system depends not only on the values of the function over its domain, but also on the procedures that compute the function. In particular, the procedure to be differentiated must be a compsosition of elementary functions, such as sin, cos, and exp. Difficulties arise, then, in situations where a smooth map is "differentiated twice."

    More precisely, let $M$ and $N$ be differentiable manifolds, and let $f$ be a smooth map from $M$ to $N$. We can define the function $T f$ from $T M$ to $T N$ by:

    $$
    \begin{equation*}
    T f(p, v)=\left(p, d f_{p}(v)\right) \tag{2.3}
    \end{equation*}
    $$

    where we have used the short-hand $(p, v)$ to denote a tangent vector $v$ in $T_{p} M$ and its anchor $p$.
    Since tangent vectors are computationally represented by local tangent vectors, the procedure that computes $T f(p, v)$ needs to first find a chart of $N$ containing $f(p)$. When computing $T f$, the function must choose a chart in the range of $f$ before it could differentiate the transition function $\phi^{\prime} \circ f \circ \phi^{-1}$ (where $\phi$ is a coordinate map on $M$ and $\phi^{\prime}$ a coordinate map on $N$ ). Thus, the procedure computing $T f$ is no longer a composition of primitive procedures because of this need to choose a chart in $N$, and the system encounters errors when attempting to compute $T(T f)$ directly. One must therefore take care in forming transition functions using smooth maps.

[^6]:    ${ }^{11}$ It is easy to verify that if $p$ lies on the boundary according to one chart, then it must lie on the boundary according to all the charts.
    ${ }^{12}$ Manifolds with boundaries introduce some problems into the theory. For example, the class of differentiable manifolds with boundary is not closed under the product manifold construction: Consider the unit interval $I=[0,1]$. It is a differentiable manifold with boundary, and yet the product manifold $I \times I$ is not a differentiable manifold with boundary-Transition maps will fail to be smooth at the corners of the square.

[^7]:    ${ }^{13}$ This is not the final version of code used, but expresses the main ideas.

[^8]:    ${ }^{14}$ Variational (or Lagrangian) mechanics differs from Newtonian mechanics in the following way: Instead of describing how systems change from moment to moment, as did Newton, one looks at the space of all possible paths through the configuration space that begin at some initial point $x_{1}$ at some time $t_{1}$ and ends up in some place $x_{2}$ at some time $t_{2}$. To every such possible path $\gamma$, one assigns to it a number (called the action) $S(\gamma)$. Then the path actually taken by a particle is the one that is a stationary point (in a sense that can be made mathematically precise) of the action $S$. This is known as the principle of least action because for many cases, $S$ is actually minimized by the real path $\gamma . S(\gamma)$ is generally computed as the integral of some function $L$, called the Lagrangian, along paths; Hamilton's principle of least action then states that the "correct" Lagrangian for many situations is the difference between kinetic and potential energies.
    Since the principle of least action is formulated in terms of integrals of real-valued functions over time intervals, it is coordinate-independent. Furthermore, one can derive the equations of motion in terms of the Lagrangian:

[^9]:    ${ }^{16}$ The Hamiltonian formulation describes mechanics using position and momenta, instead of position and velocity. The space of states here is the cotangent bundle of the configuration space, not its tangent bundle. And, finally, the dynamics is described by the Hamiltonian, which is a function that in many cases agrees with the energy function. As with Lagrangian mechanics, Hamiltonian mechanics also lets us change coordinates easily; the analogous equations of motion for a given Hamiltonian $H$ are:

    $$
    \begin{align*}
    & \dot{q}=\partial_{p} H, \\
    & \dot{p}=-\partial_{q} H, \tag{2.10}
    \end{align*}
    $$

    where $q$ denotes position, $p$ denotes momentum, and $\partial_{p}$ and $\partial q$ denote the corresponding differential operators. These are Hamilton's equations. Notice that they are antisymmetric, and do not require a matrix inversion to isolate the highest-order derivatives.

[^10]:    ${ }^{17}$ This space is commonly denoted as $\mathrm{SO}_{3}$, the special orthogonal group. It is an example of a Lie group, which are manifolds that also happen to be groups, and where the group operations are smooth as maps on manifolds.

[^11]:    ${ }^{1}$ It is even easier to do in one dimension, but such cases are too simple.

[^12]:    ${ }^{2}$ It makes sense to speak of the length of $v$ because this is a directional derivative in a given chart. The length of $v$ is its magnitude according to the dot product with respect to the chart's coordinate maps.

[^13]:    ${ }^{3}$ In the case of spectral decomposition methods, the PDE discretization involves the Fourier transform.

[^14]:    ${ }^{4}$ In particular, the triangulation of surfaces and solids in $R^{3}$ has been extensively studied because of their extensive engineering applications.
    ${ }^{5}$ One might well imagine triangulating each chart first, and then somehow combining these local meshes to form a global mesh. This is, in fact, the strategy employed in proving that every manifold has a triangulation. However, there are technical difficulties with a direct implementation of this idea, as discussed in §3.5.
    ${ }^{6}$ Please do not confuse the local discretization phase with local discretization methods: The former is part of the latter. Since global discretization is not the focus of this report, this terminology should not be too confusing.
    ${ }^{7}$ This terminology comes from imagining the use of these algorithms on massively-parallel computers, where each processor, or node, represents a sample point. For example, Abelson, et. al., describe a novel new approach to computing that may be able to exploit the locality inherent in finite difference and finite element approximations to perform computations in parallel [1].

[^15]:    ${ }^{8}$ Readers unfamiliar with relaxation and other iterative methods for solving large sparse linear systems of equations are referred to Chapter 6 of Vichnevetsky [27]. Appendix A also contains a brief introduction to the subject.
    ${ }^{9}$ Except that, perhaps, one could choose local coordinate systems to "regularize" the sample point ge-

[^16]:    ${ }^{10}$ Special thanks to Thanos Siapas and Gerald Jay Sussman for telling me about this idea.

[^17]:    ${ }^{11}$ The condition number of a matrix measures, in some sense, how close the matrix is to the identity matrix. The larger it is, the harder it is to obtain numerically accurate solutions. For a more thorough discussion of condition numbers, as well as a discussion of this particular problem, see [24].

[^18]:    ${ }^{12}$ For more information about this and other related problems, see Haus and Melcher [15].
    ${ }^{13}$ This is essentially the first term in the Fourier series expansion for the solution of the slot problem with boundary values:

[^19]:    While this boundary condition is much simpler than the one above, its corresponding solution requires the computation of an infinite series that converges rather slowly; the relevant Fourier series is that of the unit-step funtion, where Gibbs' effect shows up.

    Note that this boundary condition is also discontinuous, which makes accurate numerical solutions somewhat harder to obtain (especially near the corners). This is one of the many reasons why one may wish to have the ability to use multiple coordinate systems when solving PDEs, thus concentrating computational

[^20]:    effort near discontinuities in the boundary data.
    ${ }^{14}$ Actually, the algorithm used is successive overrelaxation (SOR), with a relaxation factor of 1.9. This helps accelerate the convergence rate; for more information, see Appenix A or Vichnevetsky [27].
    ${ }^{15}$ Actually, using uniformly distributed random numbers to place nodes uniformly in a rectangular region tends to create clusters of nodes because the law of large numbers does not give us a very tight bound on the variance of the distribution from the mean, so it is necessary to enforce a minimum distance between nodes to ensure a "uniform" distribution.

[^21]:    ${ }^{16}$ The computations in this section are done using MATLAB ${ }^{T M}$.
    ${ }^{17}$ This could be done because the system only has 300 interior nodes, and hence 300 unknowns. With 10,000 unknowns, there is no way to invert the matrix directly! Of course, from the view of error analysis, one should be suspicious of directly inverting even a $300 \times 300$ matrix...

[^22]:    ${ }^{18}$ For those who have not had exposure to point set topology, compactness in this context is equivalent to saying that the image of $\bar{U}_{i}$ under $\phi$ is a closed and bounded subset of $V$. It is a topological property independent of the chart.
    ${ }^{19}$ In most treatments of partitions of unity, axiom 2 is stated using open covers, not atlases. However, for our purposes, partitions of unity are most useful when the open cover is an atlas.

[^23]:    ${ }^{20}$ As discussed in §B.2, the other approach is to associate determinants to functions, so that instead of integrating real-valued functions, one integrates functions called differential forms, whose values are "determinant-like" functions.

[^24]:    ${ }^{21}$ The differential $d \phi^{-1}$ is well-defined because $\phi^{-1}$ is a smooth map from the open subset $V$, which is a manifold itself, into the manifold $M$.

[^25]:    ${ }^{22} \mathrm{~A}$ good introduction to general topology and such concepts as compactness, connectedness, and continuity for general topological spaces is Munkres [19].

[^26]:    ${ }^{23}$ This procedure is a bit of a misnomer, since boundary charts, as defined, are really charts of the manifold $M$, not charts of the boundary manifold $\partial M$.

[^27]:    ${ }^{24}$ That is, $E$ is part of the intersection of the supports of the basis functions $\phi_{i}$ and $\phi_{j}$.
    ${ }^{25}$ It should be clear what $L_{l e f t}$ and $L_{\text {right }}$ mean for functions on Euclidean spaces. In the context of manifolds, think of the operators $L_{\text {left }}$ and $L_{\text {right }}$ as $m$-tuples of partial differential operators as defined earlier in §3.1, which would map real-valued functions $f: M \rightarrow R$ on $M$ to $m$-vector-valued functions $L f: M \rightarrow R^{m}$.

[^28]:    ${ }^{26}$ In the case of Laplace's equation, the symmetric positive semi-definite form $-\int\langle$,$\rangle on the space of$ differentiable functions is called the Dirichlet form.

[^29]:    ${ }^{27}$ A little matter of terminology: Many procedures in this code manipulate data structures called "complexes" (as in chart:get-complex). The term refers to simplicial complexes, which are spaces that can be formed as the union of points, lines, triangles, tetrahedra, and their higher-dimensional generalizations called simplices. Not only are simplicial complexes useful for finite element computation, they are also very important for studying the structure of topological spaces and form one of the starting points for algebraic topology. For more details, see Munkres [20]. For our purposes, however, it is just a convenient way to package data structures that describe triangulations on charts.

[^30]:    ${ }^{28}$ This will be discussed in more detail in the next section.

[^31]:    ${ }^{29}$ The procedure fem-discretize is stored away and called later for the local finite element assembly procedure. It provides a simple interface to the program of the previous section. It can always be replaced by a different FEM routine, of course.

[^32]:    ${ }^{30}$ This is what the procedure sparse-normal-equations does. While there exist much better methods for producing least-squares solutions to overdetermined systems, such as singular value decomposition (also known as SVD; see [24]), they do not apply easily to large systems of equations. In order to use iterative solution methods, the normal equations are the easiest way to facilitate the use of iterative solution methods like relaxation on overdetermined systems.

[^33]:    ${ }^{31}$ That is, its condition number has been increased.

[^34]:    ${ }^{32}$ Actually, images of charts on manifolds are generally open sets, so they cannot intersect along a boundary in the way described here. However, their closures can behave this way.

[^35]:    ${ }^{33}$ These equations are obtained by identifying the variables $u_{i}$ and $u_{i}^{\prime}$, and then taking the sum and the difference of the two resulting equations contributed by the two charts.
    ${ }^{34}$ This can come about if the elements had opposite orientations, so that the integrals pick up an extra minus sign.

[^36]:    ${ }^{35}$ Nodes are shared in the sense that if $n$ belongs to a chart $C_{1}$, and its location on the manifold also places it in the chart $C_{2}$, then a node at exactly the same location exists in $C_{2}$, and hence the two nodes can be identified later on.

[^37]:    ${ }^{36}$ In three dimensions, triangles become tetrahedrons, and in even higher dimensions they are called simplices. A space that is formed by "pasting" together simplices is known as a simplicial complex.

[^38]:    ${ }^{1}$ A notable exception occurs in numerical general relativity, where the use of Regge calculus suggests some interesting ideas for the work at hand. Einstein's field equations are very much beyond the scope of this report, though, and will not be discussed here. For more information on Regge calculus, see Sorkin [26]. For a good introduction to general relativity, see Schutz [25].

[^39]:    ${ }^{2}$ Spacetime is simply the set of all spatial positions of our space along with time indices. Points in spacetime are often called events, and Figure $4-1$ would be an example of a spacetime diagram.

[^40]:    ${ }^{3}$ Technically, this is known as a mixed initial-boundary value problem because it contains both initial data in time (the top two equations) and boundary data in space (the bottom two).
    ${ }^{4}$ Specifically, this problem is ill-posed in that there is no generally applicable existence and uniqueness theorem for such problems. On the other hand, for special cases of the wave equation over rectangular regions, there are existence and uniqueness results for the boundary value problem. See Fox and Pucci [13] and Payne [22].

[^41]:    ${ }^{5}$ There exist equations, such as the diffusion equation $\left(D_{t}-k \Delta\right) u=0$, where neither initial value nor boundary value problems are well-posed.

[^42]:    ${ }^{6} \mathrm{~A}$ "bounded spacetime region" is a subset of the spacetime domain that is bounded in spacetime, not just bounded in space.
    ${ }^{7}$ Normal derivatives of order less than $m$ can be specified arbitrarily. For a more coherent and less vague exposition of this material, see John [16].
    ${ }^{8}$ The general nonlinear partial differential equation can be transformed into a quasilinear equation by differentiating with respect to its highest-order derivative. A quasilinear equation is one that is linear in the highest-order derivatives, but the coefficients may depend on the unknown solution and its lower derivatives. Since the order of the equation is increased by this transformation, additional constraints can and must be derived from the original data and appended to the new data. However, this allows us to define characteristic surfaces for all equations.

    This also shows why nonlinear equations are complicated: The characteristics of linear equations depend only on the coefficients themselves, and thus are almost always well-defined. However, for nonlinear (quasilinear) equations, since the coefficients themselves can depend on the unknown solution and its derivatives, the characteristic manifolds (and hence the directions of information propagation) depend on the particular solution, thus complicating the problem tremendously.

[^43]:    ${ }^{9}$ Equation (4.19) may seem a bit unwieldy in our notation, but consider how one would write this in traditional notation: One is tempted to simply write

    $$
    \frac{\partial L}{\partial u}=\frac{\partial}{\partial x}\left(\frac{\partial L}{\partial u_{x}}\right)+\frac{\partial}{\partial y}\left(\frac{\partial L}{\partial u_{y}}\right)
    $$

    But both $L$ and $u$ have $x$ and $y$ as arguments, and the notation $\frac{\partial}{\partial x}$ does not distinguish between them. So this equation is wrong! The correct way to write this in traditional notation requires writing out all the arguments, which is an even bigger mess than Equation (4.19).

[^44]:    ${ }^{10}$ That is, if such a stationary point $u$ exists at all.
    ${ }^{11}$ Apologies are due to Professors Guillemin and Sternberg for borrowing the title of their book.

[^45]:    ${ }^{12}$ This section supposes some familiarity with relativistic concepts.
    ${ }^{13}$ Symmetric nondegenerate tensor fields, such as Lorentz metrics, are known as pseudo-Riemannian metrics. Because they have orthogonal eigenvectors, the basic argument that defined integration on Riemannian manifolds also works on any pseudo-Riemannian manifold: The key result is the fact that with respect to a Lorentz metric, we can define orthonormal bases, which are orthogonal basis vectors with magnitude $\pm 1$. Then the matrix representatio of bases are also orthogonal matrices, and their determinants are $\pm 1$. Taking absolute values defines local integrals consistently.
    ${ }^{14}$ Compactness is required for computing the action, but not for computing Equation (4.19) in local coordinates.

[^46]:    ${ }^{15}$ Noting the jumps in relative error in alternating entries of Table 4.3.1 and its similarity to Table 4.3.2, the problem does seem to be related to the parity of the mesh used.

[^47]:    ${ }^{16}$ It should be clear the we can choose $\phi$ so that it is continuous almost everywhere, except at the corners on the boundary of $X_{2}$. Similar comments apply to $\phi^{-1}$.

[^48]:    ${ }^{17}$ For such problems, it is necessary to consider absorbing boundary conditions, which help make space "look" infinite using a finite number of spatial sample points. For more information, see Engquist and Majda [12].

[^49]:    ${ }^{1}$ The spectral radius of a matrix $A$ is the maximum among the absolute vaules of the eigenvalues of $A$. Those familiar with some point set topology should notice that this criterion is equivalent to saying that the function defined by $f(x)=B x+b$ is a contraction mapping.

[^50]:    ${ }^{2}$ The idea of well-posedness is due to Jacques Hadamard, the great French mathematician, who also discovered some of the earliest examples of ill-posed problems. As a result, well-posed problems are sometimes, called well-posed in the sense of Hadamard in mathematical literature.

[^51]:    ${ }^{3}$ In the case of Laplace's equation, the coefficients of the discretized equations form a positive-definite

[^52]:    matrix, and may be inverted using many methods, such as relaxation, LU factorization, or conjugate gradient methods.
    ${ }^{4}$ This report does not attempt to precisely define this notion, but the union of the elements should at least be topologically equivalent to the original domain $\Omega$.

[^53]:    ${ }^{5}$ Unlike finite difference methods, which only work with values of solutions at sample poitns, finite elements explicitly interpolates between sample points in discretizing PDEs.
    ${ }^{6}$ These are actually stationary points of the approximate action. For Laplace's equation, this is indeed the minimum. For other equations where variational principles apply, stationary points need not minimize the action.
    ${ }^{7}$ This step brings up a subtle point: There are two conditions that the approximate solution must satisfy, and together they produce a unique solution. One is that the approximate solution minimizes the action, and the other is that the solution has the required boundary values. This can be thought of as a constrained minimization problem. There are two approaches to these sorts of problems: The first (the one used here) is

[^54]:    to enforce the constraint first, and then minimize the action. The second involves minimizing the action first, and then enforcing the constraint. A careful analysis will show that the second approach actually produces a overdetermined system of equation; in order to arrive at the same equations one must justify the elimination of the "extra" equations involving the inner product of the residual and basis functions corresponding to boundary nodes.

[^55]:    ${ }^{8}$ For solving large systems of linear equations, iterative methods are generally preferred over direct methods (such as LU decomposition) because of speed and the accumulation of round-off errors.
    ${ }^{9}$ The matrix is denoted by boldface in this section because the symbol $A$ also refers to one of the regions in Figure (A-2).

