The NumLab Numerical Laboratory

EINDHOVEN UNIVERSITY OF TECHNOLOGY
Department of Mathematics and Computing Science

RANA 99-46
December 1999

The NumLab Numerical Laboratory

by

J. Maubach and A. Telea

Reports on Applied and Numerical Analysis
Department of Mathematics and Computing Science
Eindhoven University of Technology
P.O. Box 513
5600 MB Eindhoven, The Netherlands
ISSN: 0926-4507

The NUMLAB Numerical Laboratory

J. MAUBACH[1] AND A. TELEA[2]

Department of Mathematics and Computer Science,
Eindhoven University of Technology, The Netherlands

Abstract A large range of software environments addresses numerical simulation, interactive visualisation and computational steering. Most such environments are designed to cover a limited application domain, such as finite element or finite difference packages, symbolic or linear algebra computations or image processing. Their software structure rarely provides a simple and extensible mathematical model for the underlying mathematics. Thus, assembling numerical simulations from computational and visualisation blocks, as well as building such blocks is a difficult task for the researcher in numerical simulation.
This paper presents the NUMLAB environment, a single numerical laboratory for computational and visualisation applications. By closely reflecting the modelled mathematical concepts into the software architecture, NUMLAB offers a basic, yet generic framework for a large class of computational applications, such as partial and ordinary differential equations, non-linear systems, matrix computations and image and signal processing. Building applications which combine interactive visualisation and computations is provided in a simple and interactive visual manner. We present several features of the NUMLAB approach, illustrated with some applications.

1 Introduction

The NUMLAB (Numerical Laboratory) environment has been constructed after a thorough search through a wide range of software environments for numerical computation, interaction, and data visualisation. NUMLAB's goals include seemless integration of computation and visualisation, convenient application construction, communication with other software environments, and a high level of extensibility for research purposes. In order to assess the merits of the NUMLAB environment, we first consider software environments in general.

From a structural point of view, software environments can be classified into three categories (see for instance [33]): Libraries, turnkey systems, and application frameworks.
Libraries such as LAPACK [60], NAGLIB [61], or IMSL [62] for numerics, or OpenGL [64], Open Inventor [20], or VTK [23] for the visualisation, provide services in the form of data structures and functions. Libraries are usually easy to extend with new data types and functions. However, using libraries to build a complete computational or visualisation application requires involved programming.
Turnkey systems such as Matlab [58], Mathematica [59], or most of the existing dedicated

[1] maubachwin.tue.nl
[2] alextwin.tue.nl

numerical simulators on the market, are simpler to use than libraries to build a complete numerical application. However, extending the functionality of such systems is usually limited to a given application domain, as in the case of the dedicated simulators, or to a fixed set of supported data types, as in the case of the Matlab programming environment.
Application (computational) frameworks, such as the Diffpack and SciLab systems for solving differential equations [21, 57] or the Oorange system for experimental mathematics [24] combine the advantages of the libraries and turnkey systems. On one hand, frameworks have an open structure, similarly to libraries, so they can be extended with new components, such as solvers, matrix storage schemes, or mesh generators. On the other hand, some (notably visualisation) frameworks offer an easy manner to construct a complete application that combines visualisation, numerics, and user interaction, e.g. by means of visual programming tools such as Matlab's Simulink [58], the dataflow network editing tool of AVS or Oorange [22, 24].

With the above in mind, consider how the NUMLAB environment integrates all of these categories. On the level of libraries, NUMLAB's C++ routines can be called from and linked to, from Fortran, Pascal, C, C++ (its C++ routines can both be compiled and interpreted [65]). Next, similar to a turnkey system, it offers full integration of visualisation and numerical computation, and implements communication with other environments (for instance with Simulink [58] and MathLink [59]). On the application framework level, NUMLAB provides interactive application construction with its visual programming dataflow system VISSION.

In order to better address NUMLAB's merits on all levels, we need a more detailed examination of computational frameworks. Though efficient, most implemented computational frameworks are limited in several respects. First, limitations exist from the perspectives of the end user, application designer, and component developer [39, 33, 40, 11]. Furthermore, there are limitations due to the computational framework (library design), and extension restrictions.

First, few computational frameworks facilitate convenient interaction between visualisation (data exploration) and computations (numerical exploration), both essential to the end user. Secondly, from the application designer perspective, the visual programming facility, often provided in visualisation frameworks such as [22, 48], usually is not available for numerical frameworks.
Finally, from the numerical component developer perspective, understanding and extending a framework's architecture is still (usually) a very complex task, albeit noticeably simplified in object-oriented environments such as [21, 23].

Next to limitation with respect to the three types of users, many computational frameworks are constrained in a more structural manner: Similar mathematical concepts are not factored out into similar software components. As a consequence, most existing numerical software is heterogeneous, thus hard to deploy and understand. For instance, in order to speed up the iterative solution of a system of linear equations, a preconditioner is often used. Though iterative solvers and preconditioners fit into the same mathematical concept – that of an approximation $\mathbf{x}$ which is mapped into a subsequent approximation $\mathbf{z} \approx \mathbf{F}(\mathbf{x})$ – most computational software implements them incompatibly, so preconditioners can not be used as iterative solvers and vice versa [21].

Another example emerges from finite element libraries. Such libraries frequently restrict reference element geometry and bases to a (sub)set of possiblilities found in the literature. Because this set is hardcoded, extensions to different geometries and bases for research purposes is difficult, or even impossible.

The design of NUMLAB addresses the problems above. NUMLAB is a numerical framework which provides C++ software components (objects) for the development of a large range of interdisciplinary applications (PDEs, ODEs, non-linear systems, signal processing, and all combinations). Further, it provides interactive application design/use with its visual programming dataflow system VISSION [11, 38], cooperates (Simulink and MathLink), and can be used outside the interactive environment: Both with standard compilation and interpretation. Its computational libraries factor out fundamental notions with respect to numerical computations (such as evaluation of operators $\mathbf{z} = \mathbf{F}(\mathbf{x})$ and their derivatives), which keeps the amount of basic components small. All components of these libraries are aware of dataflow, even in the absence of the vission dataflow system, and can for instance call back to see whether provided data is valid.

The remainder of this paper addresses some fundamental NUMLAB design aspects, and is structured as follows. In section 2, the mathematics that we desire to model in software is reduced to a set of simple but generic concepts. Section 3 shows how these concepts are mapped to software entities. Section 4 illustrates the above for the concrete case of solving the Navier-Stokes partial differential equation. Section 5 presents how concrete simulations combining computations and visualisation are constructed and used in NUMLAB. Finally, section 6 concludes the paper presenting further directions.

2 The mathematical framework

In order to reduce the complexity of the entire software solution, we show how NUMLAB reduces different mathematical concepts to fewer more basic mathematical notions. As a rule, NUMLAB's components are either operators $\mathbf{F}$, or their vector space arguments $\mathbf{x}, \mathbf{y}$. The most frequent NUMLAB operations are the related operator evaluation $\mathbf{F}(\mathbf{x})$ as well as vector space operations such as $\mathbf{x} + \mathbf{y}$:

1. *Transient boundary value problems (BVPs)*: Static Finite Elements/Volumes/Differences discretizations are *operators* $\mathbf{F}$ such that $\mathbf{F}(\mathbf{x}) = \mathbf{0}$, defines the BVP's solution $\mathbf{x}$. For transient problems, such operators can be reformulated with the use of time-step and time-integration operators into ordinary differential equations (ODEs) operators. Alternatively, BVP operators can be directly reduced to differential algebraic equation (DAEs) operators, or straightforward to a sequence of non-linear operators;

2. *Systems of ODEs*: Such systems are operators and their evaluation is reduced to repeated evaluations of non-linear operators;

3. *Systems of non-linear equations*: Such systems are operators, and reduced to a sequence of linear systems, possibly with the use of derivative evaluations;

4. *Systems of linear equations*: Such sysems are also operators $\mathbf{F}(\mathbf{x}) = A\mathbf{x} - \mathbf{b}$, and solved with a sequence of operator evaluations combined with vector space operations.

The reduction from one type of operator into another is commented on in the subsections of section 2, in the reverse order of the itemisation above. Thus, section 2.1, treats systems of linear and non-linear equations. Section 2.2 treats the reduction of systems of ODEs to non-linear systems. Finally, section 2.3 shows how to either reduce a discretized boundary value problem to a non-linear system, or to a system of ODEs.

2.1 Linear and non-linear systems

This subsection introduces NUMLAB's operator approach, and demonstrates how operator evaluations reduce to repeated vector space operations and operator evaluations. This is demonstrated by means of non-linear systems.

Within NUMLAB, linear and non-linear systems are specializations (special cases) of generic operators. First, consider linear systems. These are of the form:

$$\mathbf{F}(\mathbf{x}) = A\mathbf{x} - \mathbf{b}. \tag{1}$$

The solution of the system satisfies $\mathbf{F}(\mathbf{x}) = \mathbf{0}$, D is the diagonal of the matrix A, and the triangular matrices LU are the (approximate) Gaussian elimination factors. Solving the above equation can be done in several ways, e.g. by direct or iterative solution methods. The latter can use a preconditioner to accelerate the computations. The following example shows that iterative solution method, preconditioner, operator and other related items need not be distinguished within the NUMLAB environment.

NUMLAB (iterative) solution methods are also operators. For the sake of demonstration, consider an accelerated Richardson's method. This method corresponds to the operator

$$\mathbf{R}(\mathbf{x}) := \mathbf{R}_{\mathbf{F},\mathbf{T}}(\mathbf{x}), \tag{2}$$

which solves $\mathbf{F}(\mathbf{x}) = \mathbf{0}$ by repeated evaluation:

$$\begin{aligned} \mathbf{x}^{(k+1)} &= \mathbf{x}^{(k)} - \mathbf{T}(\mathbf{F}(\mathbf{x}^{(k)})) \\ &:= \mathbf{S}(\mathbf{x}^{(k)}). \end{aligned} \tag{3}$$

To be precise: The evaluation

$$\mathbf{z} = \mathbf{R}(\mathbf{x}^{(0)}) \tag{4}$$

yields

$$\mathbf{z} = \mathbf{x}^{(n)}, \tag{5}$$

where $\mathbf{x}^{(n)}$ was obtained after n steps of the successive substitution in 3. As can be observed, NUMLAB allows operators to make use of auxilliary arguments, for Richardson's method $\mathbf{F}$ and $\mathbf{T}$. As a matter of fact, iterative solvers such as Richardon's have an extra auxilliary argument, an iteration control block, where the user can specify the stop criteria.

In line with the operator concept, NUMLAB regards the accelerator $\mathbf{T}$ in Richardson's method as an operator, just as the system $\mathbf{F}$ – if $\mathbf{T}$ is not provided, NUMLAB substitutes the identity operator. Now recall that the evaluation of Richardson's method 4 reduces to repeated evaluations of iteration operator $\mathbf{S}$:

```
z = S(x):
{
  r  = F(x);      // operator evaluation
  d  = T(r);      // operator evaluation
  z  = x - d;     // vector space operation
}
```

Thus the Richardson's operator evaluation in 4 involves *vector space operations* and *operator evaluations*, nothing else. In NUMLAB all non vector space operations are regarded as operator evaluations, whence all operators (which includes iterative solution methods) are build with these basic two items: *vector space operations* and *operator evaluations*. This thoroughly simplifies the mathematical framework, especially for interdisciplinary applications.

In a last emphasis on NUMLAB's operator approach, consider the accelerator $\mathbf{T}$, which is likely to depend on (a derivative of) $\mathbf{F}$:

$$\mathbf{T}(\mathbf{x}) := \mathbf{T}_{\mathbf{F}}(\mathbf{x}), \tag{6}$$

An accelerator in nature, the operator $\mathbf{T}$ presumedly approximates A's inverse: Potential candidates are:

- *Non-accelerate methods:* $\mathbf{T}(\mathbf{x}) = \mathbf{x}$;
- *Diagonal preconditioners:* $\mathbf{T}(\mathbf{x}) = D^{-1}\mathbf{x}$;
- *Incomplete LU preconditioners:* $\mathbf{T}(\mathbf{x}) \approx U^{-1}L^{-1}\mathbf{x}$;
- *Direct methods:* $\mathbf{T}(\mathbf{x}) = U^{-1}L^{-1}\mathbf{x}$,

and other related methods (even Richardson's method itself). Because $\mathbf{R}$ uses $\mathbf{T}$ which at its turn can use $\mathbf{F}$, is follows that in NUMLAB solvers can make use of other solvers, in an multiple level nested manner.

Though NUMLAB regards preconditioning as approximate function evaluation – which simplifies its framework – this does not solve the problem of proper preconditioning. Iterative solution methods might require preconditioners to preserve for instance symmetry (such as the preconditioned gradient method PCG [51]) or at least positive definiteness of the symmetric part (see GMRES [53] and GCGLS[52], and many other minimal residual methods). The application designer should keep these mathematical restrictions in mind, when designing a suitable solver for the problem at hand.

Similar to linear systems, non-linear systems are also formulated to be operators $\mathbf{F}$, with solutions $\mathbf{x} = \mathbf{F}^{-1}(\mathbf{0})$. Possibly related solution methods, are again formulated as an operator. For the sake of demonstration, consider a successive substitution operator

$$\mathbf{G}(\mathbf{x}) := \mathbf{G}_{\mathbf{F}}(\mathbf{x}). \tag{7}$$

This operator solves $\mathbf{F}(\mathbf{x}) = \mathbf{0}$ by repeated evaluation:

$$\begin{aligned} \mathbf{x}^{(k+1)} &= \mathbf{x}^{(k)} - \mathbf{F}(\mathbf{x}^{(k)}) \\ &:= \mathbf{S}(\mathbf{x}^{(k)}). \end{aligned} \tag{8}$$

More precise: The evaluation

$$\mathbf{z} = G_{\mathbf{F}}(\mathbf{x}^{(0)}) \tag{9}$$

yields

$$\mathbf{z} = \mathbf{x}^{(n)}, \tag{10}$$

where $\mathbf{x}^{(n)}$ was obtained after n user controlled steps 8. Non-linear operators which do not provide derivatives can further be solved by the combinatorial fixed point method [50] (a multi-dimensional variant of the bisection method).
Other solution methods, such as (damped, inexact) Newton methods, lead to a sequence of linear systems, and are treated as mentioned above. Again, the application designer should take care that the fixed point function $\mathbf{G}$ is chosen properly. The above choice can work for operators $\mathbf{F}$ which have a Jacobian with a positive definite symmetric part.

In order to close this section on non-linear operators, note that images are also treated as operators

$$\mathbf{F}(\mathbf{x}) = A\mathbf{x}, \tag{11}$$

where A is a matrix (or block-diagonal) matrix of color-intensities. Thus, image visualisation reduces to jacobian visualisation:

Figure 1: An image: a 256 × 256 matrix of greyscales

Section 5 will comment on the software implementation of NUMLAB, and the related viewer in figure 1 above.

2.2 Ordinary differential equations

Discretisations of ordinary differential equations can also be formulated as operators whos evaluation reduces to a sequence of vector space operations and function evaluations. For instance, let $\mathbf{E}$ be an operator, and consider the initial value problem: Find $\mathbf{x} := \mathbf{x}(t)$ for which:

$$\frac{\partial}{\partial t}\mathbf{x} = \mathbf{E}(t, \mathbf{x}), \qquad \mathbf{x}(0) = \mathbf{x}_0\,. \tag{12}$$

The application of the Euler backward method with $t_n = nh$ for all n

$$\mathbf{x}^{(n+1)} - \mathbf{x}^{(n)} = h\mathbf{E}(t_{n+1}, \mathbf{x}^{(n+1)}) \tag{13}$$

which can be rewritten as

$$\mathbf{x}^{(n+1)} - h\mathbf{E}(t_{n+1}, \mathbf{x}^{(n+1)}) - \mathbf{x}^{(n)} = \mathbf{0}\,. \tag{14}$$

Define the function $\mathbf{T} := \mathbf{T}_{t_n, \mathbf{x}^{(n)}, t_{n+1}}$ as follows:

$$\mathbf{T}(\mathbf{x}) = \mathbf{x} - h\mathbf{E}(t_{n+1}, \mathbf{x}) - \mathbf{x}^{(n)}. \tag{15}$$

Then $\mathbf{x}^{(n+1)}$ is a solution of $\mathbf{T}(\mathbf{x}) = \mathbf{0}$. Of course, $\mathbf{T}$ depends on the user-provided values $\mathbf{x}^{(n)}$, t_n and t_{n+1}.

The above operator formulation $\mathbf{x}^{(n+1)} = \mathbf{T}^{-1}(\mathbf{0})$ not only applies to explicit methods such as Runge Kutta type methods [55], but likewise applies to all implicit discretisation methods, such as Euler Backward and Backward Difference Formulas (BDF) by Gear [54].

It is obvious that the evaluation of T at a given $\mathbf{x}$ again only involves *vector space operations* and *operator evaluations*. Solving $\mathbf{T}(\mathbf{x}) = \mathbf{0}$ can thus be done by several methods: Successive substitution, Newton type methods, preconditioned methods, etc.
Naturally, a time-step integrator complements the time-step mechanism. NUMLAB provides standard fixed time-step methods and – required for stiff problems – adaptive time-step integrators of the PEC and PECE type [56]. The Lotka-Volterra predator-prey problem's solution is shown.

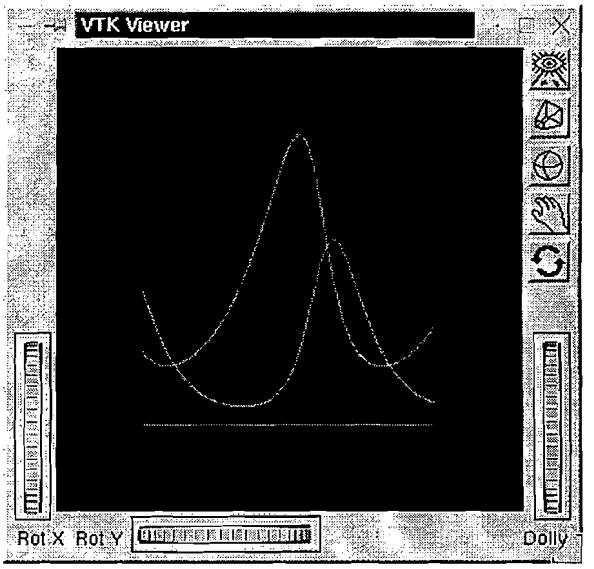


Figure 2: The predator prey solution $(x(t), y(t))$, $t \in [0, 2\pi]$

Naturally, phase-plane plots can also be generated.

2.3 Partial differential equations and initial boundary value problems

In order to show how partial differential equations (PDEs) are reduced to (non-)linear systems of equations, consider the evolution boundary value problem: Let $\Omega \subset \mathbf{R}^n$ be the region of interest. Find a solution $u(\mathbf{x}, t)$ which satisfies

$$\frac{\partial}{\partial t}u = \Delta u + f \qquad (t > 0), \qquad u(0) = u_0. \tag{16}$$

For the sake of presentation, the boundary conditions are all assumed to be of Dirichlet type. Equation (16) directly fits into the framework of section 2.2, for a suitable operator E, defined below.

Standard Galerkin finite element discretisations of the static variant $-\Delta u = f$ assume that the solution $u := u(t)$ is an element of a space $\mathbf{V}_h$, with basis $\{v_i\}_i$, i.e., $u(t, \mathbf{x}) = \sum_i u_i(t) v_i(\mathbf{x})$, for all $\mathbf{x} \in \Omega$. Related to $\mathbf{V}_h$ is the linear vector space $\mathbf{V}_h^0 \subset \mathbf{V}_h$, which contains all elements which are zero at the Dirichlet boundary. The related finite element operator $\mathbf{F}\colon \mathbf{R}^n \mapsto \mathbf{R}^n$ is coefficientwise defined as follows (see section 4.1 for more information):

$$z_i = [\mathbf{F}(\mathbf{x})]_i = \int [\nabla \left(\sum_i x_i v_i \right) \nabla v_i - f v_i] \,, \tag{17}$$

if variable i is not related to a Dirichlet point, and $z_i = 0$ elsewise (is briefly explained below. Due to this choice, the derivative operator $D\mathbf{F}(\mathbf{x})$ acts as the identity on variables related to Dirichlet points). By construction, $\mathbf{F}$ is an operator from $\mathbf{R}^n$ onto $\mathbf{R}^n$, but is related to an operator from $\mathbf{V}_h$ onto $\mathbf{V}_h^0$. The solution vector $\mathbf{x} \in \mathbf{R}^n$ of $\mathbf{F}(\mathbf{x}) = \mathbf{0}$ is the coeffient vector related to the actual solution $\mathbf{x} \in \mathbf{V}_h$.

Due to inhomogeneous boundary conditions, NUMLAB's finite dimensional spaces $\mathbf{V}_h$ are not linear vector spaces because $\mathbf{x}, \mathbf{y} \in \mathbf{V}_h$ does not imply $\mathbf{x} + \mathbf{y} \in \mathbf{V}_h$. However, NUMLAB's spaces *behave* as vector spaces because *operators map* $\mathbf{V}_h$ *into* $\mathbf{V}_h^0$ and *vector space operations are only performed on* $\mathbf{x}, \mathbf{y} \in \mathbf{V}_h^0$. This last claim is now supported by means of an example.

In order to solve $\mathbf{F}(\mathbf{x}) = \mathbf{0}$, the solution $\mathbf{x} \in \mathbf{V}$ is split into a user elected part $\mathbf{x}^{(0)}$ which satisfies the (inhomogeneous) Dirichlet conditions, and into the to be computed correction $\mathbf{d} = \mathbf{x} - \mathbf{x}^{(0)}$. Then, equation $\mathbf{F}(\mathbf{x}) = \mathbf{0}$ is solved as follows:

$$\begin{array}{rcl} \mathbf{x}^{(0)} & \in & \mathbf{V} \\ \mathbf{d} & = & (D_{\mathbf{x}}\mathbf{F}(\mathbf{x}^{(0)}))^{-1}(-\mathbf{F}(\mathbf{x}^{(0)})) \\ \mathbf{x}^{(1)} & = & \mathbf{x}^{(0)} + \mathbf{d} \end{array} \tag{18}$$

For linear systems this procedure might seem complicated. However, in this case the middle line of equation (18) translates into

$$\mathbf{d} = A^{-1}(b - A\mathbf{x}^{(0)}) \tag{19}$$

where the right hand side contribution $-A\mathbf{x}^{(0)}$ is usually described as the *elimination of boundary conditions.*
The fact that $D\mathbf{F}(\mathbf{x}) = I$ on the subset of Dirichlet related variables does not cause problems with standard iterative solution algorithms (subspace invariance) and neither create problems with properties such as positive definite, elementwise storage, and so forth.

Finally, we consider the NUMLAB formulation of 16. Under the assumption that the solution

$\mathbf{x} \in \mathbf{V}$, equation (16) is equivalent to the autonomous ODE

$$\frac{\partial}{\partial t}\mathbf{x} = -\mathbf{F}(\mathbf{x}) \qquad (t > 0), \tag{20}$$

where $\mathbf{x}(0)$ is the coefficient vector related to function u_0. At its turn, this equation is equivalent to (12) for $E(t, \mathbf{x}) = -\mathbf{F}(\mathbf{x})$. Therefore, the initial boundary value problem (16) reduces to a sequence of systems of non-linear equations.

2.4 Conclusions

The examples in sections 2.1, 2.2 and 2.3 have shown how mathematical problems with a seemingly different formulation can be reduced to the two basic operations of vector space computations and operator evaluation. Because of this, the NUMLAB software provides the basic notions as well as concrete specialisations of vector spaces $\mathbf{V}$ and operators $\mathbf{F}$ on $\mathbf{V}$.

3 From the Mathematical to the Software Framework

In this section, we show how the notion of operators $\mathbf{F}$ and arguments $\mathbf{v}$ in linear vector spaces $\mathbf{V}$ map to a software framework. As outlined in the previous section, a large class of solution methods for problems of the form $\mathbf{F}(\mathbf{x}) = \mathbf{0}$ can be reduced to a simple mathematical framework based on dimensional linear vector spaces and operators on those spaces. The software framework we propose will closely follow the mathematical model. Consequently, the obtained software product will be simple and generic as well.

First, the vector space $\mathbf{V}$ and the operator $\mathbf{F}$ are presented in more detail. The linear vector space $\mathbf{V}$ is spanned by n functions $v\colon \mathbf{R}^d \mapsto \mathbf{R}$. Thus, an element $\mathbf{x}$ in $\mathbf{V}$ is a function $\mathbf{a} \mapsto \sum x_i v_i(\mathbf{a})$, characterized by its coefficient vector $\mathbf{x} \in \mathbf{R}^n$. (Vector-spaces can be of cross product type).

In many situations, the functions v_i of the basis of $\mathbf{V}$ have local support. For instance, for finite element computations, one uses an approximation space spanned by functions having local support. In order to define this support a triangulation algorithms subdivides the domain $\Omega \in \mathbf{R}^d$ into subregions called elements. In such cases, the functions v_i are defined element-wise.

3.1 The `Grid` module

To be able to define local support for the basis functions v_i, we need to discretize the function's domain Ω. This is modelled in the software framework by the `Grid` module, which covers the function's domain with elements. As a matter of fact, `Grid` takes a `Contour` as input, which describes the boundary $\partial\Omega$ of the region Ω. The default contour is the unit-square.

In our framework, the grid can be of any dimension (e.g. 2D planar or 3D spatial), and can have any desired element shapes, such as triangles and quadrilaterals in 2D or tetrahedra, prisms, or cubes in 3D. All grids implement a common interface which provides services to iterate over the grid elements and vertices, topological queries such as vertex and element neighbours, and location of the element in which a given point falls. The common grid interface can be implemented by `Grid` modules that produce the grid in different manners, such as performing Delaunay triangulation of a given contour or reading an existing triangulation from a file.

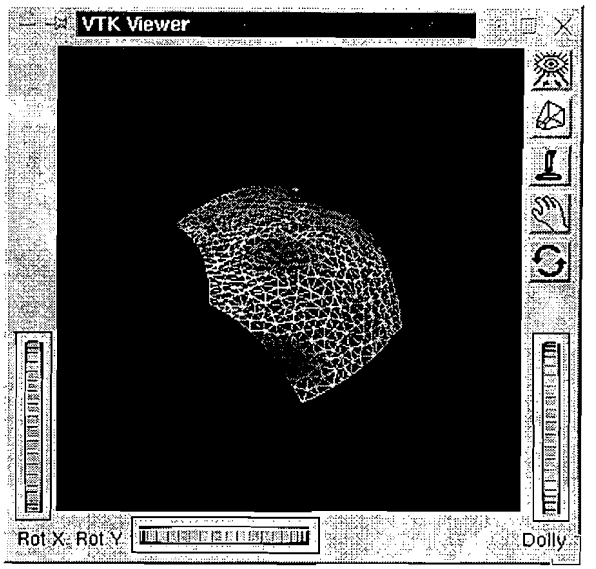


Figure 3: A cubic finite element interpolant on a 2-manifold in $\mathbf{R}^3$

3.2 The `Space` module

The vector space $\mathbf{V}$ is implemented by the software module `Space`. `Space` takes a `Grid` and `BoundaryConditions` as inputs. The grid's discretisation in combination with the boundary conditions are used to build the supports of its basis functions v_i. The default boundary conditions are Dirichlet type conditions for all solution components. None, Robin, Neumann and vectorial boundary conditions are specified per boundary part.
The `Space` module provides services to evaluate its basis functions v_i and their derivatives. A specific `Space` module will thus implement a specific set of basis functions, such as constant, linear, quadratic, or cubic. The interface of the `Space` module follows the mathematical properties of the vector space $\mathbf{V}$ presented so far, i.e. provides services to evaluate functions $\mathbf{x} \in \mathbf{V}$ and their derivatives. To present this, we first outline the implementation of the function mathematical concept.

3.3 The `Function` module

As discussed so far, a function $\mathbf{x}$ is defined in a linear vector space of functions $\mathbf{V}$, as a sum of the basis functions v_i of $\mathbf{V}$ weighted by the coefficients x_i of $\mathbf{x}$. Therefore, $\mathbf{x}$ is simply implemented as a vector of real numbers x_i in the software module `Function`. Together with these coefficients, `Function` stores a reference to its vector space `Space`. This is consistent with the assumption of our mathematical framework that a function can only exist in a given vector space.
The `Function` module provides services to evaluate the function and its derivatives at a given point $a \in \mathbf{R}^d$. To evaluate a given function $\mathbf{x}$ in a given point $\mathbf{a} \in \mathbf{R}^d$, both $\mathbf{x}$ and $\mathbf{a}$ are passed to the `Space` module of that function, which returns the value of $\mathbf{x}(\mathbf{a})$. This is computed following the definition $\mathbf{x}(\mathbf{a}) = \sum_i x_i v_i(\mathbf{a})$ described before. In order to implement this efficiently, `Space` first determines which of the basis functions v_i have supports that contain

the point **a**. Next, the above scalar product between the values of the selected basis functions v_i in the point **a** and the corresponding coefficients x_i is computed. The computation of the derivatives of a given function **x** in a point **a** follows a similar implementation.
Providing evaluation of functions $\mathbf{x} \in \mathbf{V}$ and of their derivatives at given points is, strictly speaking, the minimal interface the `Space` module has to implement. However, it is sometimes convenient to be able to evaluate a function at a point given as an element number and local coordinates within that element. This is especially important for efficiency in the case where one operation is iterated over all elements of a `Grid`, such as in the case of numerical integration. If the `Space` module allows evaluating functions at points specified as elements and local element coordinates, the implementation of the numerical integration is considerably faster than when point-to-element location has to be performed. Consequently, we also provided the `Space` module with a function evaluation interface which accepts an element number and a point defined in the element local coordinates.

3.4 The `Operator` module

As described previously, an operator $\mathbf{F}\colon \mathbf{V} \mapsto \mathbf{W}$ maps an element $\mathbf{x} \in \mathbf{V}$ to an element of a linear vector space $\mathbf{W} \ni \mathbf{z} = \mathbf{F}(\mathbf{x})$. The evaluation $\mathbf{z} = \mathbf{F}(\mathbf{x})$ computes the coefficients z_i of **z** from the coefficients x_i of **x**, as well as from the bases $\{v_i\}$ and $\{w_i\}$ of **V** and **W** respectively. Next to the evaluation of **F**, derivatives such as the Jacobian operator $D\mathbf{F}$ of **F** are evaluated in a similar manner. Such derivatives are important in several applications. For example, they can be used in order to find a solution of $\mathbf{F}(\mathbf{x}) = \mathbf{0}$, with Newton's method.
The software implementation of the operator notion follows straightforwardly the mathematical concepts introduced in Section 2. The implementation is done by the `Operator` module, which offers two services: evaluation of $\mathbf{z} = \mathbf{F}(\mathbf{x})$ and of the Jacobian of **F** in point **y**, $\mathbf{z} = D\mathbf{F}(\mathbf{y})\mathbf{x}$. To evaluate $\mathbf{z} = \mathbf{F}(\mathbf{x})$, the `Operator` module takes two `Function` objects **z** and **x** as input and computes the coefficients z_i using the coefficients x_i and the bases of the `Space` objects **z** and **x** carry with them. It is important that both the 'input' **z** and the 'output' **x** of the `Operator` module are given, since it is in this way that most `Operators` determine the spaces **V**, respectively **W**.
To evaluate $\mathbf{z} = D\mathbf{F}(\mathbf{y})\mathbf{x}$, the `Operator` proceeds similarly. It takes two input `Function` objects **y** and **x**. The point **y** determines where the Jacobian $D\mathbf{F}$ is computed, and the point **x** where it is evaluated. Next, `Operator` takes also an output `Function` object **z** in which the result of the Jacobian evaluation is stored. Finally, `Operator` provides also an output for its Jacobian in a given point **y**, $D\mathbf{F}(\mathbf{y})$. The Jacobian is also implemented as an `Operator` which, given an argument **x**, outputs $D\mathbf{F}(\mathbf{y})\mathbf{x}$. Internally, $D\mathbf{F}(\mathbf{y})$ is implemented as a matrix of coefficients, and the operation $D\mathbf{F}(\mathbf{y})\mathbf{x}$ is a matrix-vector multiplication. However, the implementation details are hidden from the user, who works only with the mathematical notions of `Function` and `Operator`.
Specific `Operator` implementations differ in the way they compute the above two evaluations. For example, a simple `Diffusion` operator $\mathbf{z} = \mathbf{F}(\mathbf{x})$ may produce a function **z** where $z_i = x_{i-1} - 2x_i + x_{i+1}$. A generic `Linear` operator may produce a vector of coefficients $\mathbf{z} = \mathbf{A}\mathbf{x}$ where **A** is a matrix. A `Summator` operator $\mathbf{z} = \mathbf{F}_1(\mathbf{x}) + \mathbf{F}_2(\mathbf{x})$ may take two inputs $\mathbf{F}_1$ and $\mathbf{F}_2$ and produce a vector of coefficients $z_i = [\mathbf{F}_1(\mathbf{x})]_i + [\mathbf{F}_2(\mathbf{x})]_i$. Remark that the modules implementing the `Linear` and `Summator` operators actually have two inputs each. In both cases the function **x** is the first input, while the second is the matrix **A** for the `Linear` operator and the function **y** for the `Summator` operator. These values could be as

well hard-coded in the operator implementation. In both cases however, we see `Operator` as a function of a *single* variable $\mathbf{x}$, as described in the mathematical framework.

3.5 The `Solver` module

The previous section has described a mathematical framework in which a large class of problems is reduced to solving an equation $\mathbf{F}(\mathbf{x}) = \mathbf{0}$. Solving this equation is, in its turn, reduced to performing vector space operations and evaluations of similar operators $\mathbf{G}(\mathbf{x})$. We model the solving of $\mathbf{F}(\mathbf{x}) = \mathbf{0}$ by the module `Solver` in our software framework. Mathematically speaking, `Solver` is similar to an operator $\mathbf{S}\colon \mathbf{V} \mapsto \mathbf{W}$, where $\mathbf{V}$ and $\mathbf{W}$ are function spaces. The interface of `Solver` provides evaluation at functions $\mathbf{x} \in \mathbf{V}$, similarly to the `Operator` module. The implementation of the `Solver` evaluation operation $\mathbf{z} = \mathbf{S}(\mathbf{x})$ should provide an approximation $\mathbf{z}$ to $\mathbf{z} \approx \mathbf{F}^{-1}(\mathbf{x})$. However, `Solver` does not provide evaluation of its Jacobian, as this is complex to compute in the general case, i.e. for any solver implementation.
Practically, `Solver` takes as input an initial guess `Function` object $\mathbf{x}$ and an `Operator` object $\mathbf{F}$ and modifies the $\mathbf{x}$ object such that $\mathbf{F}(\mathbf{x}) = \mathbf{0}$. The operations done by the solver are either vector space operations and evaluations of its input `Operator` object, or evaluations of similar operators $\mathbf{G}(\mathbf{x})$. In the actual implementation, this is modelled by providing the `Solver` module with one or more extra inputs of type `Solver`. In this way, one can for example connect a chain of preconditioners to an iterative solver module.
Implementing several `Solver` s follows straightforwardly from its mathematical description. Iterative solvers such as Richardson, GMRES, (bi)conjugate gradient, with or without preconditioners, are easily implemented in this framework. The framework makes no distinction between a solver and a preconditioner, following the mathematical model introduced in Section 2. The sole difference between a solver and a preconditioner in this framework is semantic, not structural. A solver is supposed to produce an *exact* solution of $\mathbf{F}(\mathbf{x}) = \mathbf{0}$, whereas the preconditioner is *supposed* to return an approximate one. Both are implemented as `Solver` modules, which allows easy cascading of a chain of preconditioners to an iterative solver as well as using preconditioners and solvers interchangeably in applications. The framework makes also no structural distinction between direct and iterative solvers. For example, an `ILUSolver` module is implemented to compute an incomplete LU factorization of its input operator $\mathbf{F}$. The `ILUSolver` module can be used as a preconditioner for a `ConjugateGradient` solver module. In the case the `ILUSolver` is not connected to the `ConjugateGradient` module's input, the latter performs non preconditioned computations. Alternatively, a `LUSolver` module is implemented to provide a complete LU factorization of its input operator F. The `LUSolver` can be used eigher directly to solve the equation $\mathbf{F}(\mathbf{x}) = \mathbf{0}$, or as preconditioner for another `Solver` module.

3.6 An object-oriented approach to the software framework

Sofar, sections have outlined the structure of the proposed numerical software framework. This structure is based upon a few basic modules which parallel the mathematical concepts of `Grid`, `Function`, `Space`, `Operator`, and `Solver`. These modules provide their functionality via interfaces containing a small number of operations, such as the `Operator`'s evaluation operation or the `Grid`'s element-related services previously outlined.

As stated in the beginning of this section, a large range of numerical problems can be modelled with these few generic modules. In order to capture the specifics of a given problem, such as the type of PDE to be solved or the basis functions of an approximation space, the generic modules have to be specialized. The specialized modules provide the interface declared by their class, but can implement it in any desirable fashion. For example, a `ConjugateGradient` module implements the `Solver` interface of evaluating $\mathbf{z} = \mathbf{F}^{-1}\mathbf{x}$ by using the conjugate gradient iterative method.
The above architectural requirements are elegantly and efficiently captured by using an object-oriented approach to software design [7, 26, 10]. Consequently, we have implemented our numerical software framework as an object-oriented library written in the C++ language [13]. This design enabled us to naturally model the concepts of basic and specialized modules as class hierarchies. The software framework implements a few base classes `Grid`, `Function`, `Space`, `Operator`, and `Solver`. These base classes declare the interface to their operations. The interface is next implemented by various specializations of these base classes. An overview of the implemented specializations follows:

- `Grid`: 2D and 3D grid generators for regular and unstructured grids, and grid file readers;
- `Function`: Several specific functions v_i are provided, such as sines, cosines, or piecewise (non-)conforming polynomial functions in several dimensions;
- `Space`: There is a single `Space` class, but several basis functions are implemented, as described further in Section 5;
- `Operator`: Operators for several ODEs,PDEs, and non-linear systems have been implemented, such as Laplace, Stokes, Navier-Stokes, and elasticity problems. Next, several operators for matrix manipulation and image processing have been implemented;
- `Solver`: A range of iterative solvers including biconjugate gradient, GMRES, GCGLS, QMR, and Richardson are implemented. Several preconditioners such as ILU are also provided as `Solver` specializations, following the common treatment of solver and preconditioner modules previously described.

Besides the natural modelling of the mathematics in terms of class hierarchies, the object-oriented design allows users to easily extend the current framework with new software modules. Implementing a new solver, preconditioner, or operator usually involves writing only a few tenths of lines of C++ to extend an existing one. The same approach also facilitates the reuse of existing numerical libraries such as LAPACK [60] or Templates [63] by integrating them in the current object-oriented framework.

4 The transient Navier-Stokes equations

This section describes a finite element NUMLAB operator `F`, suited for the solution of the transient Navier-Stokes equations. This finite element operator is chosen because the involved mathematics requires a finite dimensional vector space `V` of basis functions, and because the transient formulation leads to differential algebraic equations. This class of equations is common in industrial problems.

To begin with, we examine the static problem. For a discretisation, we first show that there is a straightforward and lucid relation between the mathematical formulae and their NUMLAB software implementation. We further show how the NUMLAB implementation of `F` accomplishes the finite element integration, without `V` exposing its basis functions and element geometries to `F`.

The static case is followed with the NUMLAB software formulation of the transient problem. We demonstrate that the static problem operator `F` can be used in combination with all suitable time-integrators `S` – suited for indefinite/stiff problems.

Due to the high degree of orthogonality between `F`, `V` and time-stepper methods, NUMLAB can and does offer a range of finite element types – higher order, as well as non-conforming Crouzieux-Raviart types – on rather arbitrary support geometries: simplices, parallellepipida, prisms, etc. It facilitates and supports user-defined reference bases and geometries, as well as user-supplied geometries and grid generators. Existing applications do not have to be adapted for new bases and geometries, as long as all required mathematical conditions hold.

4.1 The Navier-Stokes equations

The Navier-Stokes equations describe an incompressible fluid $\mathbf{u}$ subject to forces $\mathbf{f}$, in a region Ω. For the sake of brevity, we assume $\Omega \in \mathbf{R}^2$. Then the fluid's velocities are $\mathbf{u} = [u_1, u_2]$, and p denotes the pressure. The classical problem is to find sufficiently smooth $(\mathbf{u}, p)$ such that in Ω:

$$\begin{cases} -\epsilon\Delta\mathbf{u} + \mathbf{u}\nabla\mathbf{u} + \nabla p &= \mathbf{f}, \\ \nabla\cdot\mathbf{u} &= 0. \end{cases} \tag{21}$$

For the sake of demonstration, we assume all boundary conditions of Dirichlet type (parabolic in/outflow profiles and no-slip along walls). Problem 21 is discretized with the use of a finite element method:

The equations are multiplied with a test function $(\mathbf{v}, q)$ in a finite dimensional Hilbert space $\mathbf{V}$ – commented on later – and partial integration results in a variational problem: Find $(\mathbf{u}, p) \in V$ such that for all $(\mathbf{v}, q) \in \mathbf{V}$

$$\begin{cases} \int \epsilon\nabla\mathbf{u} : \nabla\mathbf{v} - p\nabla\mathbf{v} + (\mathbf{u}\nabla\mathbf{u} - \mathbf{f})\mathbf{v} &= 0, \\ \int \nabla\cdot\mathbf{u}\, q &= 0. \end{cases} \tag{22}$$

This problem can be reformulated as: Find the vector function $\mathbf{x} = [u_1, u_2, p] \in \mathbf{V}$ which solves $\mathbf{F}(\mathbf{x}) = 0$.

In order to understand the NUMLAB implementation `F` of the Navier Stokes operator $\mathbf{F}$, we must examine the component-wise definition of $\mathbf{F} := [F_1, F_2, F_3]$. Let $\mathbf{x} = [x_1, x_2, x_3] \in V$,

and $\mathbf{z} = [z_1, z_2, z_3] := \mathbf{F}(\mathbf{x})$. Due to the definition of $\{V_c\}_c$, these vector functions are of the form $x_c = \sum x_{ci} v_{ci}$, described with coefficients $[x_c]_i = x_{ci}$. The Navier-Stokes operator, discretized in space, now is:

$$\begin{aligned} z_{1i} &= [\mathbf{F}_1(x_1, x_2, x_3)]_i \\ &= \int \epsilon \nabla x_1 \nabla v_{1i} - x_3 \partial_x v_{1i} + (x_1 \partial_x x_1 + x_2 \partial_y x_1 - f_1) v_{1i} \\ z_{2i} &= [\mathbf{F}_2(x_1, x_2, x_3)]_i \\ &= \int \epsilon \nabla x_2 \nabla v_{2i} - x_3 \partial_y v_{2i} + (x_1 \partial_x x_2 + x_2 \partial_y x_2 - f_2) v_{2i} \\ z_{3i} &= [\mathbf{F}_3(x_1, x_2, x_3)]_i \\ &= \int (\partial_x x_1 + \partial_y x_2) v_{3i} \,. \end{aligned} \tag{23}$$

It is evident – as stated earlier – that $\mathbf{F}$ uses the coefficients of $\mathbf{x}$ as well as the bases functions in order to compute the coefficients of the result $\mathbf{z}$.

The NUMLAB operator `F` implements a discrete version of (23). For the sake of notation, we assume Ω is covered with a triangular grid, and that $\mathbf{V}$ is an orthogonal product of spaces $\mathbf{V} = V_1 \times V_2 \times V_3$. For the Navier-Stokes applications, the spaces $\{V_c\}_c$ have to satisfy the L.B.B. condition, so for instance we use quadratic conforming finite elements for the velocities (V_1 and V_2), and piecewise linear conforming finite for the pressure (V_3).

The finite element operator is evaluated support-wise, which requires the definition of basis function v_{ci} restricted to support e. Using the notation from the previous section, let solution $\mathbf{x} = [x_1, x_2, x_3] = [u_1, u_2, p]$. Then $x_c = \sum x_{ci} v_{ci}$, where v_{ci} are the functions which span finite dimensional vector space $\mathbf{V}_c$. The restriction of v_{ci} to support e is $v_{c,r}$ (global function i, restricted to support e, is function r) is coded in software with `i(c, r)`. The integrals in (22) are computed supportwise, with the use numerical integration, involving integration points $\mathbf{x}_k$. The value $v_{c,r}(\mathbf{x}_k)$ and gradient vector $\nabla v_{c,r}(\mathbf{x}_k)$ are represented with `v(c)(k)(r)`, respectively `dv(c)(k)(r)`. In a similar manner, `x(c)(k)`, respectively `dx(c)(k)` denote $x_c(\mathbf{x}_k)$ and $\nabla x_c(\mathbf{x}_k)$. Likewise, `dv(c)(k)(r)(dY)` denotes $\partial_y v_{c,r}(\mathbf{x}_k)$. Define `U1 = 0, U2 = 1, P = 2`. The NUMLAB evaluation of `z = F(x)` and `z = DF(x)*y` for support e (typeset to fit this layout) is:

```
Operator z = F(x):

 z(U1)(i(U1)(r)) += qw(k)*
   (eps*dx(U1)(k)*dv(U1)(k)(r) -
    x(P)(k)*dv(U1)(k)(r)(dX) +
   (x(U1)(k)*dx(U1)(k)(dX) +
    x(U2)(k)*dx(U1)(k)(dY) -
    f1(qp(k)) * v(U1)(k)(r)));
 z(U2)(i(U2)(r)) += qw(k)*
   (eps*dx(U2)(k)*dv(U2)(k)(r) -
    x(P)(k)*dv(U2)(k)(r)(dY) +
   (x(U1)(k)*dx(U2)(k)(dX) +
    x(U2)(k)*dx(U2)(k)(dY) -
```

```
    f2(qp(k)) * v(U2)(k)(r)));
z(P)(i(P)(r)) += qw(k)*
   ((dx(U1)(k)(dX) +
     dx(U2)(k)(dY)) * v(P)(k)(r));
```

The Jacobian's description is:

```
Jacobian z = DF(x)*y:

DF(U1)(U1)(i(U1)(r))(i(U1)(s)) += qw(k)*
    (dv(U1)(k)(s)*dv(U1)(k)(r) +
     v(U1)(k)(s)*dx(U1)(k)(dX) +
     x(U1)(k)*dv(U1)(k)(s)(dX));
       ...

DF(P)(U2)(i(P)(r))(i(U2)(s))    += qw(k)*
    (dv(U2)(k)(s)(dY)*v(P)(k)(r));

z = DF * y;
```

Both evaluation operations have an almost identical loop structure:

```
V = x->getSpace();
for (Integer e = 0; e < V->NElements(); e++)
 V->fetch(i, v, dv, x, dx, .....);
 for (Integer c = 0; c < i.size(); c++)
  for (Integer r = 0; r < i(c).size(); r++)
   for (Integer k = 0; k < x(c).size(); k++)
```

The Jacobian has an extra inner loop over trial functions `s`. With regard to this implementation, several observations come to mind:

- First, `F` does not have spaces `V` and `W` as input (i.e., as auxilliary variables). The spaces are obtained from the input/output variables. This technique simplifies computational networks.
- Secondly, because `F` performs numerical integration, it solely requires *the value* of (partial derivatives of) the basis functions at the quadrature points. The basis functions themselves are not required, so `F` operates orthogonal to `V` and `W`.
- Finally, the NUMLAB operator models the Navier-Stokes equations in (22) in a convenient fashion. The software implementation is one-to-one with the mathematics, and can in fact be automated.

Finally, recall that the derivative operator acts as the identity operator on Dirichlet point related variables, which requires `fetch` to deliver the related information.

4.2 The time discretisation

The transient Navier-Stokes equations can be formulated as so-called differential algebraical equations (DAEs):

$$\begin{cases} \frac{\partial}{\partial t}\mathbf{u} &= \epsilon\Delta\mathbf{u} - \mathbf{u}\nabla\mathbf{u} - \nabla p + \mathbf{f}, \\ \mathbf{u}(0) &= \mathbf{u}_0, \\ \nabla\cdot\mathbf{u} &= 0. \end{cases} \tag{24}$$

NUMLAB will solve such systems with the use of specialized packages such as `dassl`, or ortherwise explicitly with the use of the static NUMLAB operator $\mathbf{F}$ in (23), in combination with a specific time-discretisation.

For the sake of illustration of the last option, first consider solving the time-dependent variables $\mathbf{u}$ in (24) with the use of a rather basic time-step method: the θ method. In practice, for stiff problems – high Reynolds number – one would rather use Gear's method. The θ-method for a general non-linear system of ODEs:

$$\frac{\partial}{\partial t}\mathbf{u} = \mathbf{E}(t, \mathbf{u}) \tag{25}$$

leads to the recursion:

$$\begin{array}{rcl} u_0 &=& \mathbf{u}(0), \\ \frac{\mathbf{u}_{n+1}-\mathbf{u}_n}{h} &=& \theta\mathbf{E}(t_n, \mathbf{u}_n) + (1-\theta)\mathbf{E}(t_{n+1}, \mathbf{u}_{n+1}) \end{array} \tag{26}$$

with solution $\mathbf{u} = \lim_{n\mapsto\infty}\mathbf{u}^{(n)}$, for $t \mapsto \infty$. This all fits into the NUMLAB `Operator` style, if we define the time-step operator $\mathbf{T} := \mathbf{T}_{t_n,\mathbf{u}^{(n)},t_{n+1}}$ as follows:

$$\mathbf{T}(\mathbf{u}) := \mathbf{u} - \mathbf{u}^{(n)} - h\theta\mathbf{E}(t_n, \mathbf{u}^{(n)}) - h(1-\theta)\mathbf{E}(t_{n+1}, \mathbf{u}). \tag{27}$$

In this manner, the Jacobian of $\mathbf{T}$ is positive definite for small h, if the Jacobian of $\mathbf{E}$ is, and the approximation $\mathbf{u}^{(n+1)}$ of $\mathbf{u}(t_{n+1})$ is a root of

$$\mathbf{T}(\mathbf{u}) = \mathbf{0}. \tag{28}$$

Finally, the operator $\mathbf{T}$ is used to describe the solution of the explicit DAE: Note that (24) is equivalent to:

$$\begin{cases} \frac{\partial}{\partial t}u_1 &= -F_1(\mathbf{u}, p) \\ \frac{\partial}{\partial t}u_2 &= -F_2(\mathbf{u}, p) \\ \mathbf{u}(0) &= \mathbf{u}_0, \\ 0 &= F_3(\mathbf{u}, p). \end{cases} \tag{29}$$

If we define operator $\mathbf{E}_p(\mathbf{u}) = [-F_1(\mathbf{u}, p), -F_2(\mathbf{u}, p)]$, then (29) reduces to

$$\begin{cases} \frac{\partial}{\partial t}\mathbf{u} &= E_p(\mathbf{u}) \\ \mathbf{u}(0) &= \mathbf{u}_0, \\ 0 &= F_3(\mathbf{u}, p). \end{cases} \tag{30}$$

Every approximate solution $\mathbf{x}^{(n+1)} = [\mathbf{u}^{(n+1)}, p^{(n+1)}]$ must both solve

$$\begin{cases} \mathbf{0} &= \mathbf{T}(\mathbf{u}, p) \\ 0 &= F_3(\mathbf{u}, p). \end{cases} \tag{31}$$

where according to the theta-step method (27),

$$\mathbf{T}(\mathbf{u}, p) := \mathbf{u} - \mathbf{u}^{(n)} - h\theta\mathbf{E}_p(t_n, \mathbf{u}^{(n)}) - h(1-\theta)\mathbf{E}_p(t_{n+1}, \mathbf{u}) \,. \tag{32}$$

Summarizing (24) – (32), we have shown that each approximate solution $\mathbf{x}^{(n)}$ of $\mathbf{x}(t_n)$ must solve the non-linear system of equations $\mathbf{G}(\mathbf{x}) = \mathbf{0}$ defined by

$$\mathbf{G}_1(\mathbf{x}) = T_1(\mathbf{x}), \qquad \mathbf{G}_2(\mathbf{x}) = T_2(\mathbf{x}), \qquad \mathbf{G}_3(\mathbf{x}) = F_3(\mathbf{x}). \tag{33}$$

This new non-linear operator $\mathbf{G}$ can be supplied to a NUMLAB non-linear solver in order to compute a solution $\mathbf{x}$ to $\mathbf{G}(\mathbf{x}) = \mathbf{0}$.

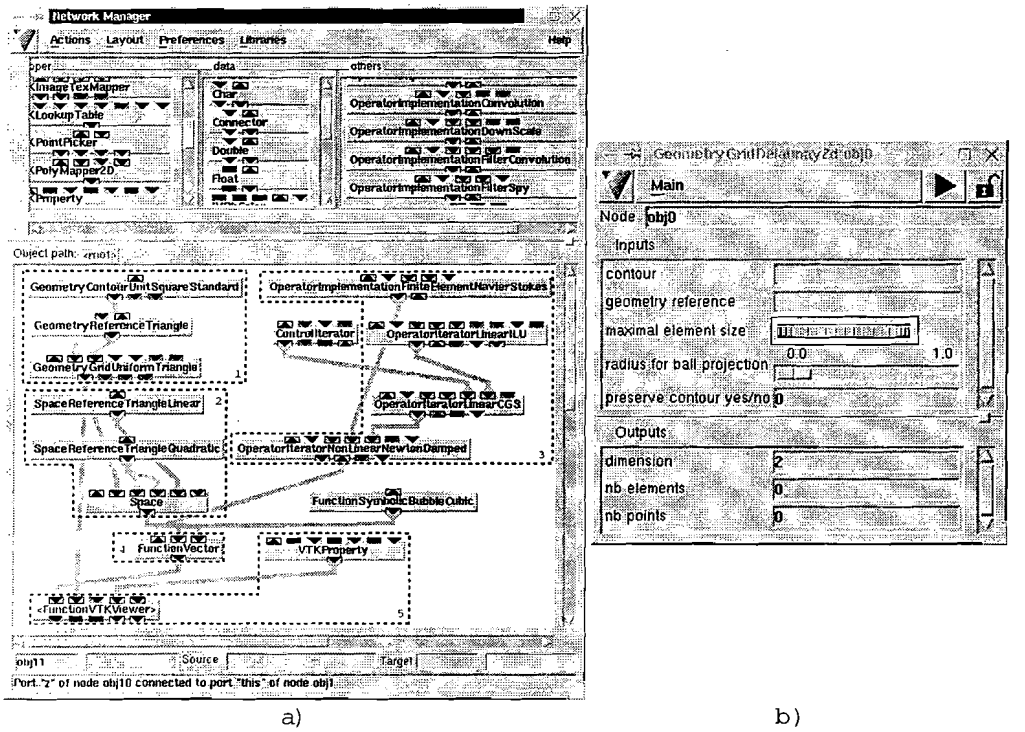
Finally, some remarks and observations. First, the value which operator $\mathbf{G}$ attains at $\mathbf{x}$, is composed of the values which $\mathbf{F}$ attains at related points. Therefore, the Jacobian $D\mathbf{G}(\mathbf{x})$ can be formulated in terms of $D\mathbf{f}$ at related points. The NUMLAB implementation of time-steps exploits this: The Jacobian $D\mathbf{G}(\mathbf{x})$ is a sequence of call-backs to the Jacobians $D\mathbf{f}$. Secondly, a NUMLAB discretized system of partial differential equations leads to an operator $\mathbf{F}(\mathbf{u})$ which can be *the* operator provided to an ODE step method. In this case, the PDE and ODE discretisations are used strictly orthogonal in the software implementation.

5 Application design and use

The previous sections have presented the structure of the NUMLAB computational framework. It has been shown how new algorithms and numerical models can easily be embedded in the NUMLAB framework, due to its design based on few generic mathematical concepts. This section treats the topics of numerical application construction and use with the NUMLAB system.

As stated in section ??, a numerical framework should provide an easy way to construct numerical experiments by assembling predefined components such as grids, problem definitions, solvers, and preconditioners. Next, one should be able to interactively change all parameters of the constructed application and monitor the produced results in a numerical or visual form. Shortly, we need to address the three roles of component development, application design, and iteractive use for the scientific computing domain.

We have approached the above by integrating the NUMLAB component library in the VISSION system. As NUMLAB is written as a C++ component library, its integration into VISSION was easy. Moreover, the structure of NUMLAB as a set of components that communicate by data streams in order to perform the desired computation matches well VISSION's dataflow application model. The integration of the NUMLAB library in VISSION implied writing the short metaclass descriptions for its few tenths of numerical components. As no modification of the NUMLAB code was necessary, its integration in VISSION took only a few hours of work. Once all the NUMLAB components were integrated into VISSION, constructing numerical applications and providing interactive computational steering and visualisation was easily achieved by using VISSION's visual network construction and end user interaction facilities described in chapter ??. We shall illustrate the above with the Navier-Stokes problem discussed in the previous section.



a) b)

Figure 1: a) Navier-Stokes simulation built with NUMLAB components. b) User interface for the grid generator module

5.1 Navier-Stokes simulation construction

As outlined previously, numerical applications built with the NUMLAB components are actually VISSION dataflow networks. Figure 1 a shows such a network built for the Navier-Stokes problem discussed in the previous section. The modules in the Navier-Stokes computational network in Fig. 1 are arranged in five groups. The functionality of these groups is explained in the following.

5.1.1 Computational Domain

The first group contains modules which define the geometry of the computational domain. This basically contains modules that accomplish three functions:

1. definition of the computational domain's contour.
2. definition of the reference geometric element.
3. mesh generation

In our example, the computational domain is a rectangular region whose boundary is defined by the `GeometryContourUnitSquareStandard` module. This module allows the specification of the rectangle's sizes, as well as a distribution of mesh points on the contour. Next, the `GeometryGridUniformTriangle` module produces a meshing of the rectangle into triangles. The reference triangle geometry is given by the `GeometryReferenceTriangle`. The mesh produced by the `GeometryGridUniformTriangle` module conforms both to the reference

element supplied as input and to the boundary points output by the `GeometryContourUnitSquareStandard` module. Different combinations of contour definitions, mesh generators, and reference elements are easily achieved by using different modules. In this way, 2D and 3D regular and unstructured meshes of various element types such as triangles, quadrilaterals, hexahedra, or tetrahedra can be produced. The produced mesh can be directly visualised or further used to define a computational problem.

5.1.2 Function Space

The second group contains modules that define the function space V over the computational domain. The modules in this group perform two functions:

1. definiton of a set of basis functions v_i that span V.
2. definition of V from the basis functions and the discretised computational domain.

The first task is done by the `SpaceReferenceTriangleLinear` and `SpaceReferenceTriangleQuadratic` modules, which define linear, respectively quadratic basis functions on the geometric triangles. The functions are next input into the `Space` module, which has already been discussed in the previous sections. The support of the basis functions is defined by the computational domain's discretisation which is also input into `Space`. In our case, `Space` uses the quadratic basis function module twice and the linear basis function module once, as the 2D Navier-Stokes problem has two velocity components to be approximated quadratically and one linearly approximated pressure component.
An important advantage of the design of NUMLAB is the orthogonal combination of basis functions and geometric grids. Several other (e.g. higher order) basis function modules are provided as well, defined on different geometric elements. By combining them as inputs to the `Space` module, one can easily define a large range of approximation spaces for various computational problems. In the case of a diffusion PDE solved on a grid of quadrilaterals, for example, one would use a single `SpaceReferenceQuadLinear` basis function input to the `Space` module.

5.1.3 Operators and solvers

The third group contains modules that define the function $\mathbf{F}$ for which the equation $\mathbf{F}(\mathbf{x}) = \mathbf{0}$ is to be solved, as well as the solution method to be used. This group contains thus specialisations of the `Operator` and `Solver` modules described in the previous sections.
In our example, the discrete formulation of (23) discussed in the previous section is implemented by the `OperatorImplementationFiniteElementNavierStokes` module. The static Navier-Stokes problem is then solved by a Newton solver implemented by the `OperatorIteratorNonLinearNewtonDamped` module. The linear system output by the Newton module is then solved by a conjugate gradient solver implemented by the `OperatorIteratorLinearCGS` module. The solution is accelerated by using an incomplete LU preconditioner `OperatorIteratorLinearILU` which is passed as input to the conjugate gradient solver.
Other problems can be readily modelled by choosing other operator implementations. Similarly, to use another solution or preconditioning method, a chain of `Solver` specialisations can be constructed. As solvers have an input of the same `Solver` type, complex solution algorithms can be built on the fly.

5.1.4 Functions

The fourth group contains specialisations of the `Function` module. These model both the solution of a numerical problem as well as its initial conditions or other involved quantities such as material properties. In our example, the `FunctionVector` module holds both the velocity and pressure solution of the Navier-Stokes equation. The solution is updated at every iteration, as this module is connected to the solver module's output. As explained in the previous sections, a function is associated with a space. This is seen in the `Function`'s input connection to the `Space` module.
The solution of the problem is initialised by connecting the `FunctionSymbolicBubble` module to the `FunctionVector`'s input. When the user changes the initial solution value, by changing an input of the `FunctionSymbolicBubble` signal or by replacing it with another function, the network restarts the computations from this new value.

5.1.5 Scientific Visualisation

As presented in section 1, a computational environment should provide a large range of facilities for data visualisation and monitoring. Such facilities should cover the following:

- several *dataset representations*, such as structured, unstructured, curvilinear, rectilinear, and uniform grids, with several types of values defined per node or per cell (scalar, vector, tensor, colour, etc). Support for image datasets should be provided as well. Besides these discrete datasets, the possibility of defining continuous datasets (e.g. implicit functions) should also be taken into account.

- several *dataset processing* tools, such as dataset readers and writers for various data formats, filters producing streamlines, streamribbons, isosurfaces, warp planes, slices, dataset simplifications, feature extraction, and so on. Imaging operations should also be supported, such as image filtering, Fourier transforms, image segmentation, colour processing, etc.

- several *visualisation primitives*, such as 2D and 3D rendering or objects with various shading models, mapping scalars to colors via various colourmaps, direct manipulation of the viewed objects, interactive data probing and object picking, hard copy options, animation creation, and so on.

A second requirement is that the visualisation tools should be open for extension or customisation, as researchers often need to extend, adapt, optimise, or experiment otherwise with various visualisation algorithms and data structures.
Writing such a library is clearly a task out of the scope of a single person. Moreover, such libraries exist, offering various degrees of application domain specificity and numbers of components. In order to provide NUMLAB with the desired visualisation capabilities, we have integrated the Visualization Toolkit (shortly VTK) [23] library into the VISSION environment. VTK is one of the most powerful freely available scientific visualisation libraries, with over 400 components for scalar, vector, and tensor visualisation, imaging, volume rendering, charting, and more. Similarly to NUMLAB, VTK is implemented as a set of C++ classes that specialise a few basic concepts such as datasets, filters, mappers, actors, viewers, and data readers and writers.

The last group of modules provides visualisation facilities to the computational network. The main module in here is the `FunctionVTKViewer` meta-group which takes as input the current solution of the Navier-Stokes equation and the grid upon which it is defined. In our example, the `FunctionVTKViewer` module inputs the velocity and pressure solution components into various visualisation methods, such as stream lines and hedgehogs for the vectorial, respectively color plots and isolines for the scalar component. Several other visualisation methods can be easily attached to the Navier-Stokes simulation, by editing the contents of the `FunctionVTKViewer` meta-group. Keeping the visualisation back-end pipeline inside a single meta-group allows a natural separation of the computational network from the post-processing operations. This also helps to reduce the overall network visual complexity.

5.2 Navier-Stokes simulation steering and monitoring

Once the Navier-Stokes computational network is constructed, one can start an interactive simulation by changing the parameters of the various modules involved, such as mesh refinement, solver tolerance, or initial solution value. All the numerical parameters, as well as the parameters of the visualisation back-end are accessible via the module interactors automatically created by VISSION (Fig. 1 b).
Moreover, the evolution of the intermediate solutions produced by the Newton solver can be interactively visualised. This is achieved by constructing a loop which connects the output of the `OperatorIteratorNonLinearNewtonDamped` module to its input. The module will then change the `FunctionVector`, and thus the visualisation pipeline downstream of it, at every iteration. This allows one to interactively monitor the improvement of the solution at a given time step, and eventually change other parameters to experiment new solvers or preconditioners.
Figure 2 shows a snapshot from an interactive Navier-Stokes simulation. The simulation domain, shown meshed in Fig. 2 a, consists of a 2D rectangular vessel with an inflow and an outflow. The inflow and outflow have both parabolic essential boundary conditions on the fluid velocity. The sharp obstacle placed in the middle of the container can be interactively manipulated by the end user by dragging its tip with the mouse anywhere inside the vessel. Once the obstacle's shape is changed, the NUMLAB network re-meshes the new domain, recomputes the stationary solution for the Navier-Stokes simulation defined on this new domain, and displays the velocity solution (Fig. 2 b). Various other parameters, such as fluid viscosity, mesh refinement, and solver accuracy, can also be interactively controlled. The computational steering of the above problem proceeds at near-interactive rates. Consequently, such NUMLAB setups can be used for quick, interactive testing of the robustness and accuracy of various solvers, preconditioners, and mesh generators. For example, one can test the speed and robustness of an iterative solver for different combinations of obstacle size and shape, mesh coarseness, and fluid viscosity for the above problem.
NUMLAB can also be used for solving large computational problems. In the following example, glass pressing in the industry is considered. The process of moulding a hot glass blob pressed by a parison is simulated. The glass is modelled as a viscous fluid, subjected to the Navier-Stokes equations. The pressing simulation is a time-dependent process, where the size and shape of the computational domain is changed at every step, after which the stationary Navier-Stokes equations are solved on the new domain. The flow equations can be solved on a two-dimensional cross-section in the glass, since the real 3D domain is axisymmetric.
The simulation is analogous in many respects to the one previosuly presented. However,

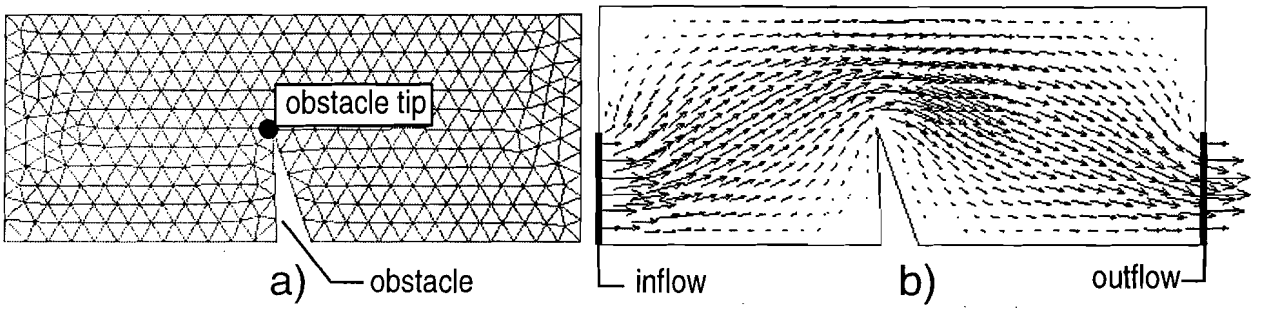


Figure 2: Interactive Navier-Stokes simulation: mesh (a) and velocity solution (b)

a mesh in the glass pressing simulation involves tenths of thousands of finite elements, whereas the previous example used only a few hundreds. Consequently, the latter simulation can not be steered interactively. However, all computational parameters of the involved NUMLAB network can be interactively controlled at the beginning of the process, or between computation steps. Figure 3 shows several results of the glass pressing simulation. The first row depicts several snapshots of the 3D geometry of the moulded glass, reconstructed and realistically rendered in NUMLAB from the 2D computational domain. The second row in Fig. 3 shows fluid pressure snapshots taken during the 2D numerical simulation. The output of the sc NumLab visualisation pipeline can be connected to an MPEG movie creation module. In this way, one can produce movies of the time-dependent simulation which can be visualised outside the VISSION environment as well.
The above has presented two computational applications built with the NUMLAB library in the VISSION system. However different in terms of interactivity, computational complexity, and visualisation needs, these applications illustrate well the smooth integration of numerics, user interaction, and on-line visualisation that is achieved by embedding the NUMLAB library in the VISSION environment.

6 Conclusions

For numerical simulations, the numerical laboratory NUMLAB offers high level interactive application design and use with VISSION, low level extensions – in all C-link compatible languages – and communication (Matlab, Mathematica). Thus, it avoids the critical software environment limitations mentioned in the introduction: The end-user is offered seemlessly integrated numerical exploration and visualisation, the application designer can use the interactive programming environment VISSION and component developers can link against canned software in most languages, and even call back on Matlab and Mathematica, both compiled and interpreted. Furthermore, due to the factoring out of a few fundamental mathematical notions – all iterative solution methods, all preconditioners, all time steppers etc., are instance of approximate evaluations $\mathbf{x}^{(k+1)} = F(\mathbf{x}^{(k)})$ – only a few fundamental concepts exist: Vectors `x`, spaces `V`, and operators `F` on such spaces.

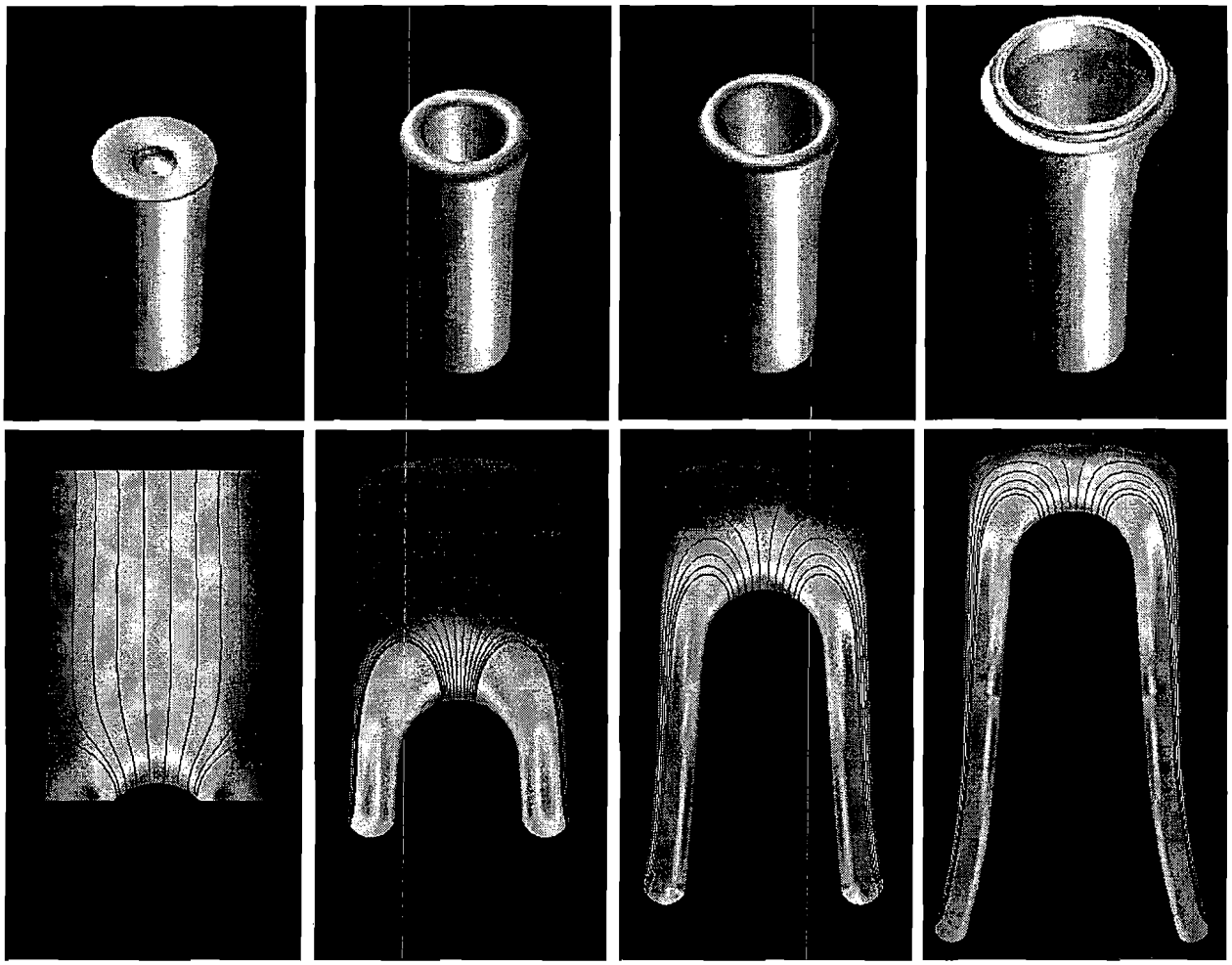


Figure 3: 3D visualisation of glass pressing (top row). Pressure magnitude in 2D cross-section (bottom row)

References

[1] A. GOLDBERG AND D. ROBSON, *Smalltalk-80: The Language and its Implementation*, Addosin-Wesley, 1983.

[2] J.D. MULDER, *Computational Steering with Parametrizable Geometric Objects*, PhD thesis, 1998, Universiteit van Amsterdam, Wiskunde en Informatica, The Netherlands

[3] W. REISIG, *Petri Nets: An Introduction*, Springer-Verlag, 1985

[4] J. D. MULDER, J. J. VAN WIJK, *3D computational steering with parametrized geometric objects*, Proceedings of the 1995 Visualization Conference, eds. G. M. Nielson and D. Silver, p.304-311, 1995.

[5] I. SOMMERVILLE, P. SAWYER, *Requirements Engineering: a Good Practice Guide*, John Wiley and Sons, 1997.

[6] E. PEETERS, *Design of an Object-Oriented, Interactive Animation System*, Ph.D. thesis, Eindhoven University of Tehcnology, Mathematics and Computing Science, 1995.

[7] J. RUMBAUGH, M. BLAHA, W. PREMERLANI, F. EDDY, W. LORENSEN. *Object-Oriented Modelling and Design*, Prentice-Hall, 1991

[8] *The Java 3D Application Programming Interface*, http://java.sun.com/products/java-media/3D/

[9] T. BUDD, *An Introduction to Object-Oriented Programming*, Addison-Wesley, 1997.

[10] G. BOOCH, *Object-Oriented Analysis and Design*, Benjamin/Cummings, Redwood City, CA, second edition, 1994.

[11] A. C. TELEA AND J. J. VAN WIJK, VISSION: *An Object Oriented Dataflow System for Simulation and Visualization*, Proceedings of IEEE VisSym '99, Springer, 1999.

[12] R. E. JOHNSON, *Frameworks = (Components + Patterns)*, Communications of the ACM, Vol. 40, No.10, Oct. 1997, pp. 39-42.

[13] B. STROUSTRUP, *The C++ Programming Manual*, Addison-Wesley,1993.

[14] S. DEMEYER ET AL., *Design Guidelines for 'Tailorable' Frameworks*, Communications of the ACM, Vol. 40, No.10, Oct. 1997, pp. 60-65.

[15] R. BRUN, S. RADEMAKERS *ROOT - An Object Oriented Data Analysis Framework*, Proceedings AIHENP'96 Workshop, Lausanne, Sep. 1996, Nucl. Inst. & Meth. in See also http://root.cern.ch/.

[16] R. P. GABRIEL, *Patterns of Software - Tales from the Software Community*, Oxford University Press, New York, 1996.

[17] D. L. PARNAS, *On Criteria to be Used in Decomposing Systems into Modules*, Communications of the ACM, Vol. 15, No.12, Dec. 1972, pp. 1053-1058.

[18] H. G. PAGENDARM, *HIGHEND, A Visualization System for 3D Data with Special Support for Postprocessing of Fluid Dynamics Data*, in *Visualization in Scientific Computing*, edited by M. Grave, Y. LeLous, W. T. Hewitt, pp. 87-98, Springer, 1994.

[19] J. O. COPLIEN, *Advanced C++ Programming Styles and Idioms*, Addison-Wesley, 1992

[20] J. WERNECKE, *The Inventor Mentor: Programming Object-Oriented 3D Graphics with Open Inventor*, Addison-Wesley, 1993.

[21] A. M. BRUASET, H. P. LANGTANGEN, *A Comprehensive Set of Tools for Solving Partial Differential Equations: Diffpack*, Numerical Methods and Software Tools in Industrial Mathematics, (M. DAEHLEN AND A.-TVEITO, eds.), 1996.

[22] C. UPSON, T. FAULHABER, D. KAMINS, D. LAIDLAW, D. SCHLEGEL, J. VROOM, R. GURWITZ, AND A. VAN DAM, *The Application Visualization System: A Computational Environment for Scientific Visualization.*, IEEE Computer Graphics and Applications, July 1989, 30–42.

[23] W. SCHROEDER, K. MARTIN, B. LORENSEN, *The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics*, Prentice Hall, 1995

[24] C. GUNN, A. ORTMANN, U. PINKALL, K. POLTHIER, U. SCHWARZ, *Oorange: A Virtual Laboratory for Experimental Mathematics*, Sonderforschungsbereich 288, Technical University Berlin. URL http://www-sfb288.math.tu-berlin.de/oorange/OorangeDoc.html

[25] A.C. TELEA, C.W.A.M. VAN OVERVELD, *An Object-Oriented Interactive System for Scientific Simulations: Design and Applications*, in /textitMathematical Visualization, H.-C. Hege and K. Polthier (eds.), Springer Verlag 1998

[26] B. MEYER, *Object-oriented software construction*, Prentice Hall, 1997

[27] A. C. TELEA *Design of an Object-Oriented Computational Steering System*, in *Proceedings of the 8th ECOOP Workshop for PhD Students in Object-Oriented Systems*, ECOOP Brussels 1998, to be published

[28] J. J. VAN WIJK AND R. VAN LIERE, *An environment for computational steering*, in G. M. Nielson, H. Mueller and H. Hagen, eds, *Scientific Visualization: Overviews, Methodologies and Techniques*, computer Society Press, 1997

[29] S. RATHMAYER AND M. LENKE, *A tool for on-line visualization and interactive steering of parallel hpc applications*, in *Proceedings of the 11th International Parallel Processing Symposium*, IPPS 97, 1997

[30] A.C. TELEA AND C. W. A. M. VAN OVERVELD, *The Close Objects Buffer: A Sharp Shadow Detection Technique for Radiosity Methods*, the *Journal of Graphics Tools*, Volume 2, No 2, 1997

[31] M. J. NOOT, A. C. TELEA, J. K. M. JANSEN, R. M. M. MATTHEIJ, *Real Time Numerical Simulation and Visualization of Electrochemical Drilling*, in *Computing and Visualization in Science*, No 1, 1998

[32] D. JABLONOWSKI, J. D. BRUNER, B. BLISS, AND R. B. HABER, *VASE: The visualization and application steering environment*, in *Proceedings of Supercomputing '93*, pages 560-569, 1993

[33] W. RIBARSKY, B. BROWN, T. MYERSON, R. FELDMANN, S. SMITH, AND L. TREINISH, *Object-oriented, dataflow visualization systems - a paradigm shift?*, in *Scientific Visualization: Advances and Challenges*, Academic Press (1994), pp. 251-263.

[34] E. GAMMA, R. HELM, R. JOHNSON, J. VLISSIDES, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995

[35] S. G. PARKER, D. M. WEINSTEIN, C. R. JOHNSON, *The SCIRun computational steering software system*, in E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools for Scientific Computing*, pages 1-40, Birkhaeuser Verlag AG, Switzerland, 1997

[36] B. SCHNEIDERMANN, *Designing the User Interface: Strategies for Effective Human-Computer Interaction.* Addison-Wesley, second edition, 1992

[37] M. A. HALSE, *IRIS Explorer User's Guide.* Silicon Graphics Inc., Mountain View, California, 1993.

[38] A. TELEA, *Combining Object Orientation and Dataflow Modeling in the VISSION Simulation System*, Proceedings of TOOLS'99 Europe, IEEE Computer Society, 1999.

[39] P. ASTHEIMER *Sonification tools to supplement dataflow visualization*, Scientific Visualization: Advances and Challenges, Academic Press, 1994, pp. 251-263.

[40] A. M. DUCLOS, M. GRAVE, *Reference models and formal specification for scientific visualization*, Scientific Visualization: Advances and Challenges, Academic Press, 1994, pp. 251-263.

[41] T. FRÜHAUF, M. GÖBEL, K. KARLSSON, *Design of a Flexible Monolithic Visualization System, Scientific Visualization: Advances and Challenges*, Academic Press, 1994, pp. 265-286.

[42] M. F. COHEN, J. R. WALLACE, *Radiosity and Realistic Image Synthesis*, Academic Press, San Diego CA, 1993.

[43] M. F. COHEN, J. R. WALLACE, S. E. CHEN, D. P. GREENBERG, *A Progressive Refinement Approach to Fast Radiosity Image Generation*, Computer Graphics (SIGGRAPH '95 Proceedings), vol 22, no 4.

[44] G.S. ALMASI AND A. GOTTLIEB, *Highly parallel computing*, Benjamin/Cummings Publishing Company Inc., 1994.

[45] D.H. HILS, *Visual languages and computing survey: data flow visual programming languages*, Journal of Visual Languages and Computing, 3:69-101, 1992.

[46] J. SHARP, *Data flow computing*, Ellis Horwood Ltd., Chichester, 1985.

[47] J.C. BROWNE, *Parallel architectures for computer systems*, Physics Today, 37(5):28-35, 1984.

[48] G. ABRAM, L. TREINISH, *An Extended Data-Flow Architecture for Data Analysis and Visualization*, Proc. IEEE Visualization 1995, ACM Press, pp. 263-270.

[49] ELECTRONICS STAFF, *Computer technology shifts emphasis to software: a special report*, Electronics, 8:142-150, May 1980.

[50] M. J. TODD, *The Computation of Fixed Points and Applications*, Lecture Notes in Economics and Mathematical Systems 124, Springer Verlag, Berlin 1976

[51] O. AXELSSON AND V.A. BARKER, *Finite Element Solution of Boundary Value Problems*, Academic Press, Orlando, Florida, 1984

[52] O. AXELSSON, *A generalized conjugate gradient, least square method*, Numerische Mathematik, 51(1987), 209-227

[53] Y. SAAD AND M.H. SCHULTZ, *GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems*, SIAM Journal on Scientific and Statistical Computing, 7(1986), 856-869

[54] C. W. GEAR, *Numerical initial value problems in ordinary differential equations*, Prentice-Hall, 1971

[55] J.C. BUTCHER, *The numerical analysis of ordinary differential equations : Runge-Kutta and general linear methods*, Wiley, 1987

[56] R. M. M. MATTHEIJ AND J. MOLENAAR, *Ordinary differential equations in theory and practice*, Wiley, 1996

[57] INRIA-ROCQUENCOURT, *Scilab Documentation for release 2.4.1*, `http://www-rocq.inria.fr/scilab/doc.html`, 2000

[58] MATLAB, *Matlab Reference Guide*, The Math Works Inc., 1992.

[59] S. WOLFRAM, *The Mathematica Book 4-th edition*, Cambridge University Press, 1999.

[60] E. ANDERSON, Z. BAI, C. BISCHOF ET AL., *LAPACK user's guide*, SIAM Philadelphia, 1995.

[61] NAG, *FORTRAN Library, Introductory Guide, Mark 14*, Numerical Analysis Group Limited and Inc., 1990.

[62] IMSL, *FORTRAN Subroutines for Mathematical Applications, User's Manual*, IMSL, 1987.

[63] R. BARRET, M. BERRY ET AL. *Templates for the Solution of Linear Systems, 2nd Edition*, SIAM, `http://netlib2.cs.utk.edu/linalg/html_templates/Templates.html`

[64] N. JACKIE, T. DAVIS AND M. WOO, *OpenGL Programming Guide*, Addison-Wesley, 1993.

[65] M. GOTO, *The CINT C/C++ Interpreter*, `http://root.cern.ch/root/Cint.html`, 2000