Eindhoven University of Technology

MASTER

On the design of a data flow oriented visual programming language for scientific computing

van Es, Roland A.

Award date:
2001

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

TECHNISCHE UNIVERSITEIT EINDHOVEN

Department of Mathematics and Computing Science

MASTER'S THESIS

On the Design of a Data Flow Oriented Visual Programming Language for Scientific Computing

by

R.A. van Es

Supervisor: Prof.Dr. R.M.M. Mattheij
Advisor: Dr. J.M.L. Maubach

June 2001

On the Design of a Data Flow Oriented Visual Programming Language for Scientific Computing Master's Thesis

Roland van Es

June 18, 2001

Contents

Keywords: Multiple Shooting Methods for Boundary Value Problems, Visual Programming, data Flow Oriented Networks, The NumLab Numerical Laboratory

Abstract: The numerical and visualisation algorithms employed in Scientific Computing grow more and more complex. The traditional manner of constructing algorithms from source code becomes prohibitively expensive: Interpretation is too slow, compilation prevents interactive inspection, and overview gets lost. Additionally, there is no formal (computer language) specification of complex mathematical notions such as partial differential equations, ordinary differential equations, boundary value problems, etc., adding to the software tower of Babel.

Visual Programming languages allow the construction of algorithms in a visual and interactive manner and can provide a good overview. They are based on a hybrid of interpretation and compilation, and as such can offer best of both. However, their data flow oriented nature is not necessarily well suited for mathematical modelling.

This thesis presents and discusses several (design) solutions required for an efficient visual programming language for Scientific Computing. Amongst them are: The implementation of loops in a visual programming language; The integration of classical FORTRAN 77 libraries in a visual programming environment; The visual modelling of multiple shooting methods; Turing completeness of NumLab. All solutions have been implemented in the NumLab visual programming environment, initiated by the TUE Scientific Computing Group.

This project was supervised by Prof. Dr. R.M.M. Mattheij and Dr. J.M.L. Maubach. Valuable input has been received from Dr. A. Telea and Ir. S. Houben.

Introduction

This thesis examines and solves a collection of related visual programming design problems. These problems arise when complex Scientific Computing algorithms are implemented in an interactive visual programming environment. Throughout this thesis, we use *NumLab* (Numerical Laboratory) in order to discuss the problems, their solutions and implementation. This environment, see [12], [14], [6] and [15] is a research spin-off of the Scientific Computing Group at the Technical University at Eindhoven, the Netherlands.

The target of the NumLab environment is:

- The rapid visual interactive construction of complex Scientific Computing algorithms;
- Based on the (re)use of professional numerical and visualisation software;
- Complemented with own research-software.

In short, *the aim is to let different numerical software standards co-work in a lucid, efficient and user-friendly manner.* That is, to lower the programming hurdle.

Related to the existing numerical software, two major problems can be identified. First, formal specifications of mathematical entities such as partial differential equations (*PDE*), and related *finite element methods* do not (fully) exist. They are under development and for instance targeted by the OpenMath/MathML standards (see [2]).

Secondly, the existing numerical software is often written by different individual researchers, in different languages, and is not engineered for reuse. Trying to let such software co-work in a user-friendly environment is difficult, both technically and conceptually.

This thesis addresses and presents solutions for some issues, which were not or unsatisfactorily solved in the current NumLab visual programming language. It provides:

1. A clear description of a visual program, its visual and non-visual functions;
2. Loop control modules in section 6;
3. A process for the integration of standard *numerical* software in NumLab, section 7;
4. A proof of the Turing completeness of NumLab, section 8;

The solutions (1) – (3) are *design solutions* because they are not solutions of mathematical problems, and neither solutions of pure software problems. In connection with (4), a proof is presented that with incorporation of a few modules, NumLab is *Turing complete*. Except for section 8, this thesis does not deal with the theory of algorithms but with their implementation in a visual programming language.
The design solution (1), a clear description of a visual programming language (such as NumLab) and its implementation, is this thesis itself. *The* reference work [15] presents visual programming from a top-down objective oriented point of view. Here, we present it from a linear bottum-up point of view: First, we present basic modules (atoms/axioms). Next, we introduce derived modules and libraries thereof. Then we conclude with the introduction of logic (control structures). However, actual computer languages are not theoretical languages. Therefore, it is often not possible to introduce rigorous definitions.

Further, due to the demand to have a clear concept of the involved mathematics, as well as lucid visual solution, several cycles of (re)design and (re)implementation were required. Only results from the last but one (section 7) and the last (section 5) are presented.

In order to present our solutions, several concepts have to be introduced. First, visual programming and its history are discussed in section 1. Next, the related concepts of ports, modules, networks, etc., are examined in section 3. The case of loops is special, and studied in 6. Then, we provide a brief analysis of the concepts related to a standard programming language in 2. Keeping both the concepts of visual and classical programming in mind, we show how a library written in a classical language can be *wrapped* for use in a visual programming language (section 5). As an implementational test, section 7 wraps the FORTRAN 77 library BOUNDPACK for use with NumLab. Finally, section 8 shown that for a specific set of basic modules and a specific manner of deriving modules, NumLab is Turing complete. Section 10 summarises all conclusions.

A valid mathematical proof is admissible regardless of its complexity and length. An implementation must be efficient, lucid and user-friendly.

1 Visual Programming

1.1 Introduction

This section provides historical background information and shows examples of visual programming languages. The precise definitions of the related concepts are provided in sections 3–6. Sections 1.3 and 1.4 describe the visual programming in the context of Computer Science and Scientific Computing.

For complementary information, see [5], [11], [6] and [15].

1.2 Basic Concepts

This thesis studies the *visual implementation* of algorithms. An *implementation* is a (mechanical or otherwise) realisation of an *algorithm*.

Definition 1 *According to Church's thesis ([7], page 223) a function* $\mathbf{F}\colon \mathbf{X} \mapsto \mathbf{Y}$ *or algorithm is a Turing computable function.*

For the definition of a Turing computable function, see section 8.

In a *non-visual implementation*, we assume that an algorithm is implemented using procedures (Pascal), subroutines and functions (FORTRAN 77), or functions (C). In the sequel, procedures and subroutines are also called (implementations of) functions.

In a *visual implementation*, an algorithm is implemented with the use of visual components. These visual components are discussed in detail later, see 3.

Visual programming languages make use of a *canvas* which contains functions called *modules*, in between which data is communicated. Figure 1 from [5] shows a visual program, called *network*. An input datum x_0 floats along arrow direction(s) through the network, functions operate on it, and return results which get transported further. The standard order in which the data is communicated and the functions are executed is called *data flow orientation*.

Existing programming languages can be split into three categories:

- cause driven;
- cause and event driven;
- event driven;

The difference is subtle. Examine possible implementations of `z = f(g(x))`. If first calling `f` (which calls back on `g`, which calls back on the datum) yields `z`, the implementation is called an *event-driven implementation*. Alternatively, if first `x` is handed over to `g`, next `g(x)` is computed, followed by `f(g(x))`, the implementation is called a *cause-driven implementation*. The difference is not in the order of computations, nor in the obtained result, but in their initiation.

Mathematics (manipulation of expressions and functions) turns out to best fit in the event-driven case, but standard visual languages (and also NumLab) are cause-driven. For this model, inspection of data, – visualisation – seems to be easier. As a witness hereof, the professional visualisation libraries *Visualisation ToolKit (VTK)*, *Open Inventor (IV)*, as well as the visual environments *Application Visualisation System (AVS)*, and *NumLab* are all making use of at least cause-driven (data flow) techniques.

NumLab combines cause and event driven methods, and almost certain, so does AVS (source code is not available). This makes it more difficult to understand than a pure cause or event driven language.

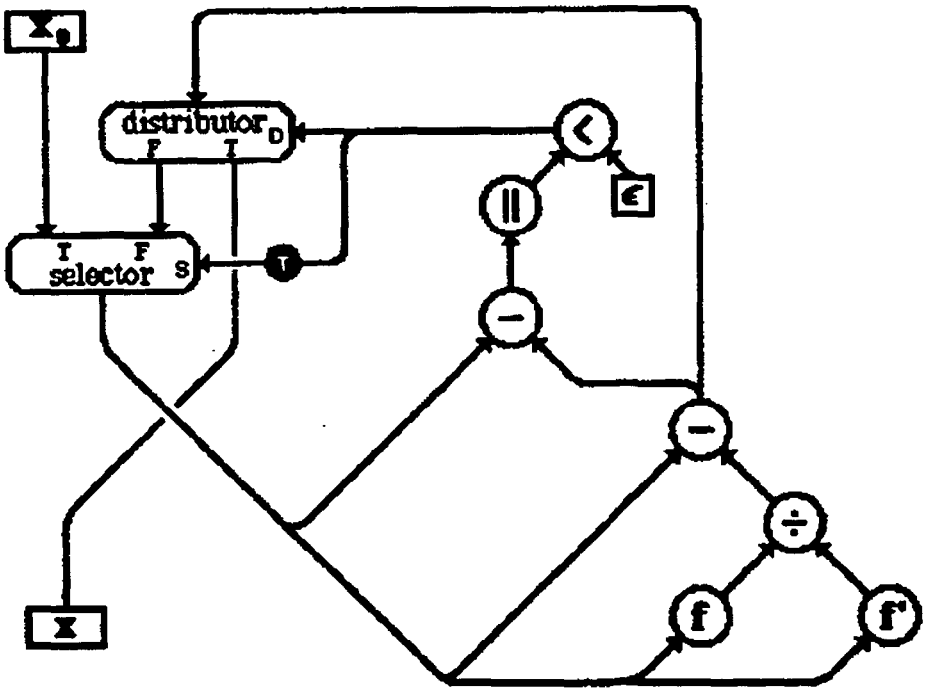


Figure 1: A Newton-Raphson network from [5]

1.3 Computer Science

In the Computer Science research in [5] and [11] the basic building modules of a visual programming language frequently are data types (**D**), related functions (**F**), and control structures (**C**):

D Natural/Float numbers;

F Basic functions: $+$, $\times$, and $-$, $/$.

C If – Then – Else; While; Repeat, etc.

Even with these simple basic modules, a visual iterative Newton-Raphson algorithm must be constructed with care, as is shown in figure 1 from [5].

It is also clear that, unless a specific purpose is kept in mind, a small and concise classical script could be preferred:

```
x = x0; y = 0; while(abs(x - y) > 10^-12) y = x; x = x - f(x)/df(x); end;
```

Keeping this in mind, it is clear that a visual implementation of more modern iterative algorithms such as BiCG-Stab and GMRES, etc. is a non-trivial undertaking.

More recent data flow implementations introduce additional *visual notation* in order to express recursive and nested constructions. An example are the *map* and *fold* operator notations used for the computation of a determinant in figure 2. This extra visual notation, which can lead to a loss of overview, turned out to be not required in the NumLab environment. Moreover, the (new and most old) visual NumLab modules can be used also in a script, as the one shown above.

In the NumLab environment, the Newton-Raphson network in figure 3 looks similar to the one in figure 1, showing that NumLab does not add to the minimal complexity present in figure 1.

1.4 Scientific Computing

In order for a visual programming language to be suitable for serious Scientific Computing computations, it must at least capture:

D Vector functions, and basic functions thereon:

F1 Numerical methods for (non-)linear systems;

F2 Numerical methods for ODE's and PDE's;

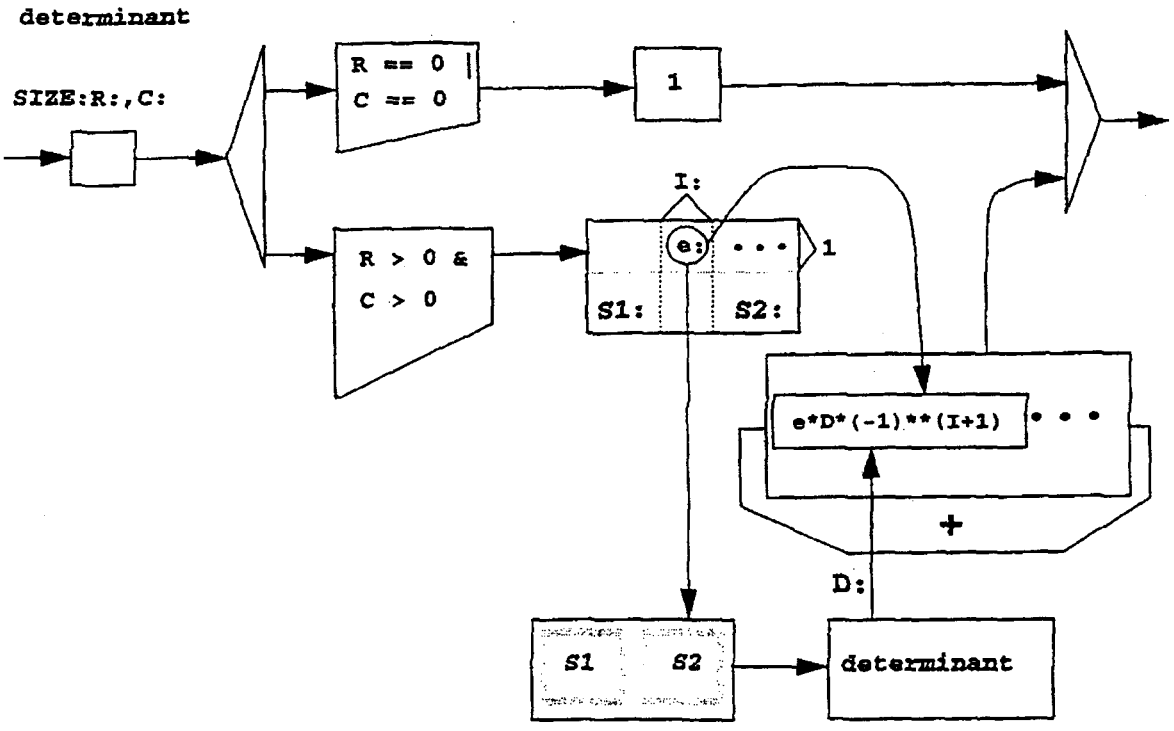


Figure 2: A determinant network from [11]

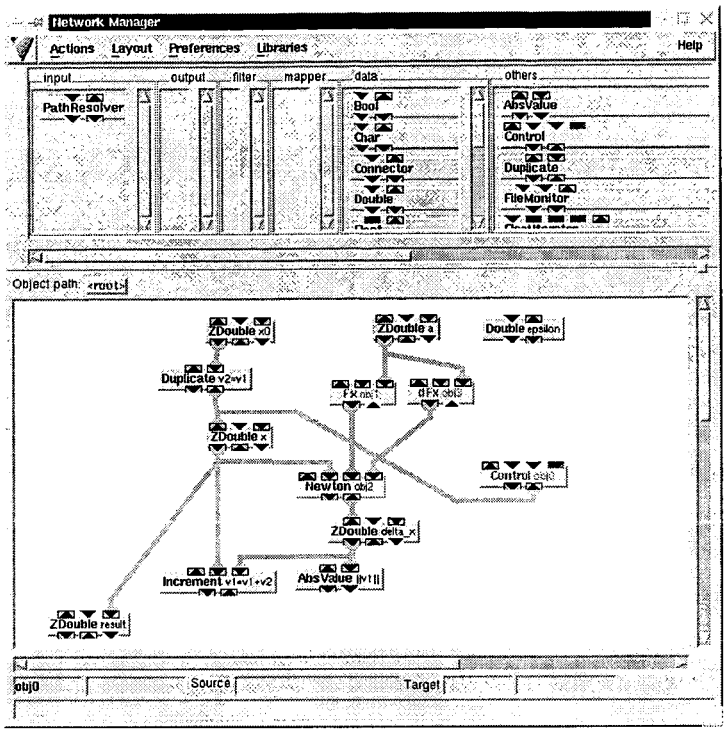


Figure 3: A NumLab Newton-Raphson network

F3 Professional visualisation, geometric modelling, etc;

C Some control structures.

When comparing the requirements with the basic modules examined in Computer Science, a clear observation can be made:

- An additional (derived) set of complex building modules is required.

The numerical workbench NumLab provides such a set of complex (not-basic, derived) language modules for **D** – **F3**. This is for instance demonstrated in figure 4. Here, a complex transient finite element program for the laser drilling of holes in turbine blades is modelled with a network containing few modules. Through this thesis, the visual programming language NumLab will be used for implementations which are to demonstrate the validity of our design solutions.

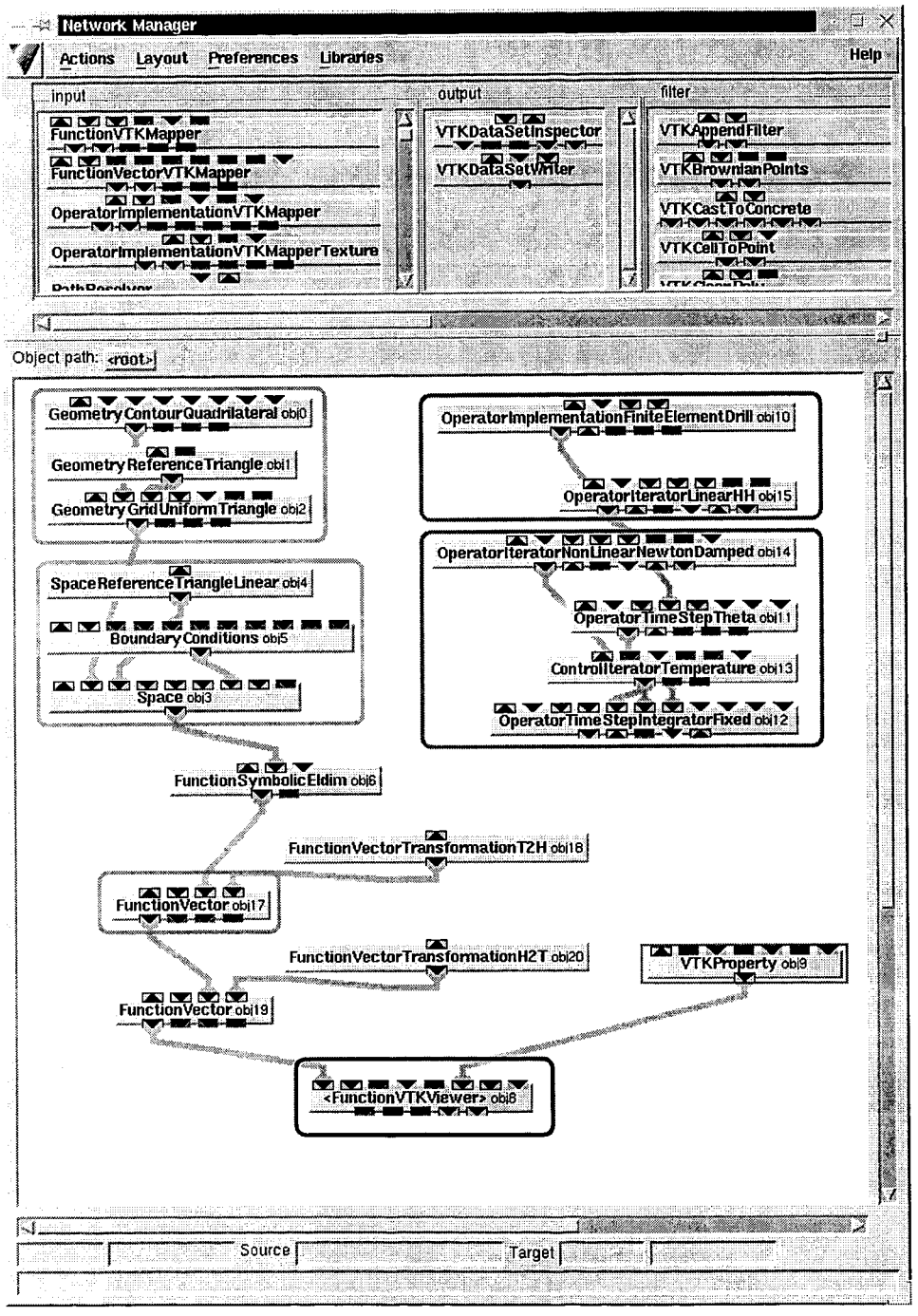


Figure 4: A laser drilling Network: Transient finite elements

2 Functions, Libraries and Sources

2.1 Introduction

Most, if not all, visual programming languages are implemented with the use of classical programming languages. In order to understand their structure, restrictions and merits, one first needs to understand the structure of the classical programming languages.

This section examines the structure of a *classical program*, i.e., the implementation of an algorithm in one of the C-linkable *classical programming languages* FORTRAN 77, Pascal, C or C++. Here, *classical* is meant to stand opposite of *visual*. In fact, this section determines the basic ingredients of such a program. It is important to understand these ingredients for the use of a related library in a visual programming environment (as described in section 5).

In order to restrict the amount of material, we focus on a *FORTRAN 77* implementation. With FORTRAN 77 we mean FORTRAN 66 or FORTRAN 77. A different reason to focus on FORTRAN 77 is the wish to reuse several standard numerical libraries written in this language (LAPACK, SEPRAN), etc.

The languages FORTRAN 90 and FORTRAN 95 resemble more the C++ language. They offer more language constructions than FORTRAN 77 and as such are more difficult to analyse. The largest additions turn out to be: More flexible memory allocation, *inheritance*, and automatic source code generation (*templates*).

2.2 Data and Functions

This section examines the role of data and functions in a classical programming language and introduces the concepts of input and output argument. Later, in section 3.3, modules take over this role in the visual programming language, both for data as well as for functions.

Let a be a real positive number and define $f(x) = x^2 - a$, with derivative $f'(x) = 2x$. Then the Newton-Raphson iterative algorithm algorithm (standard used in floating point units):

$$\begin{aligned} &x_0 \\ &x_{k+1} = x_k - f(x_k)/f'(x_k), \quad k = 0, 1, 2, \ldots \end{aligned} \tag{1}$$

approximates (computes) the square root of a, for x_0 close enough to a. Modern floating point arithmetic (Pentium, Athlon) use this function in combination with Newton's method for the determination of the root $x = \sqrt{a}$.

Example: Throughout this section, we consider a small FORTRAN 77 *source* which implements the Newton-Raphson algorithm:

```
      DOUBLE PRECISION FUNCTION FX(X, A)
C     -------------------------------
      IMPLICIT         none
      DOUBLE PRECISION X, A

C     FX = X^2 - A

      FX = X*X - A

      RETURN
      END


      DOUBLE PRECISION FUNCTION DFX(X, A)
C     --------------------------------
      DOUBLE PRECISION X, A

      DFX = 2*X
      RETURN
      END


      SUBROUTINE NEWTON(F, DF, X, A, DX)
c     ------------------------
      DOUBLE PRECISION DX, X, A
      EXTERNAL         F, DF

c     DX = - FX(X, A) / DFX(X, A)

      DX = - F(X, A)
      DX = DX / DF(X, A)

      RETURN
      END


      SUBROUTINE TESTNEWTON()
c     -----------------------
      DOUBLE PRECISION DX, X, XO, A
      EXTERNAL         FX, DFX
```

```
      WRITE(6, *) 'INPUT A  > 0 FOR COMPUTATION OF SQRT(A):'
      READ(5, *) A
      WRITE(6, *) 'INPUT X0 > 0 AS A STARTING POINT:'
      READ(5, *) X0

      X = X0

  100  WRITE(6, *) 'X:  ', X
       CALL NEWTON(FX, DFX, X, A, DX)
       WRITE(6, *) 'DX: ', DX
       X = X + DX
      IF (ABS(DX) .GT. 1.D-12) GOTO 100

      WRITE (6, *) 'SQRT(', A, ')   = ', X ,'.'

      RETURN
      END
```

For our purposes FORTRAN 77 source is generic enough – all complications related to the use libraries in section 5 can be found here. The difference between source, program and execution is explained in section 2.4.

Definition 2 *An argument* `x` *of a function* `f` *is called* input argument *if* `f` *must read its value in order to produce a result. An argument* `z` *of a function* `f` *is called* output argument *if* `f` *writes a (partial) result in* `z` *while producing its result.*

Definition 3 *The formal component specification of a library or language describes which functions it contains, which types of data are employed, and which function arguments are input or output.*

In this thesis, a formal component specification is also called *data flow analysis.*

2.3 Libraries

Important for the later reuse of FORTRAN 77 libraries in another language's context is the structure of the program. Here are the different items we distinguish (using the FORTRAN 77 code as an example):

- There are data, outside the functions, such as `a`;
- There are functions `Fx` and `dFx`;

- There is a control structure (loop).

These will be the items to be addressed in the context of a visual programming language.

Here are some restrictions which apply when a FORTRAN 77 library is to be used in a C++ context:

- Whereas a FORTRAN 77 datum's (for instance matrices) size can not be altered, a visual basic module representation must allow the size to be altered. Reason: The module is put on the canvas before the program is complete, so its size can not be predetermined.
- The amount of different data type X in FORTRAN 77 is not limited, except for machine restrictions. It could be impossible to provide basic visual modules for all data types.
- The amount of different (builtin) functions $f\colon X \mapsto X$ in FORTRAN 77 is not limited, except for machine restrictions. It could be impossible to provide basic visual modules for all data types.

2.4 Sources and Programs

We use the word *source* for the implementation of the *algorithm*. The *program* related to this source is a derived executable version. In order to execute this version, an *execution manager*, or *manager* is required. The manager also handles all input and output (I/O) or *I/O-management*. An executed program does **not** perform its own I/O.

For classical languages, the manager is a linker/loader/execution unit, and part of the operating system. This manager does not depend on the implementation language, all languages share the same manager. For a visual program, for instance for NumLab, a different manager (`cint`) is used, see section 3.6. As for the classical manager, it manages execution as well as I/O.

3 Modules, Toolboxes and Networks

3.1 Introduction

This section introduces entities such as port, module and toolbox, and presents networks and network managers.

In order to elucidate the connection between all these entities, we proceed in a kind of mathematical manner. First we define basic modules (atoms, or axioms) in section 3.3.

Next, we define derived modules called networks in section 3.5, and finally, we discuss the relation between a source (program) and network in section 3.6.

The basic reference is the work [15], where almost identical material is presented in another manner. Where we address a programming language departing from basic and derived operations, the thesis [15] presents the topic from an oo-design point of view with its on merits. Thesis [15] discusses parts of NumLab (VISSION) which reach far beyond the scope of this thesis.

Below, we offer some help, for those inclined to compare this thesis with the Ph.D. thesis [15]:

First, most of the terminology presented here can be found in [15], with a little more effort. The definition of `components` (according to the index on pages 9, 11) as well as information on modules and blocks (according to the index on page 36), is located elsewhere.

Next, the functions of the network manager (part of the kernel of [15]) are not as indicated on pages 11 or 31, but (see page 87, last line) mostly listed in figure 3.13 (KC interface). However, the list fails to include the load/unload function, which is presented on page 88.

Then, information on the network execution order is presented on pages 97 – 98. A traversal algorithm is provided but no proof regarding its operation is given.

Finally, not all employed terminology is identical. Where [15] uses the term *input port* and *output port*, we use *inflow port* and *outflow port*. This, because we reserve the terms input and output for input and output arguments of functions. We keep however the term *read port* and *write port*.

3.2 Ports

A *port*, also called *data-port* contains a value or a reference to a value. Thus, it is can contain builtin data types such as `int` but also derived data types such as structures (and more common, pointers to modules).

3.2.1 Visual Representation

Ports occur at the top or bottom of modules, introduced in the next subsection. In this thesis (see figure 5) read and write ports are positioned differently from the ports in [15], shown in figure 6. Instead of arrows pointing inwards (write port) or outwards (read port) in [15], we prefer to represent them as small rectangles, either on the outside of a module (read port) or on the inside (write port).

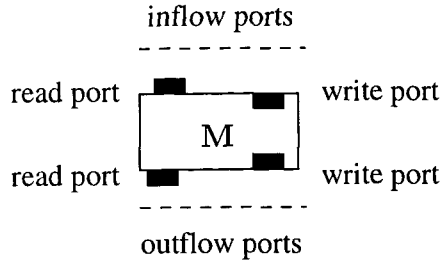


Figure 5: Ports at the bottom (outflow) and top (inflow) of a module

Figure 6: Snapshot of a NumLab module

More information regarding ports (colour, types, default values, etc.) can be found in [15]. Take care, [15] address default values using different terminology. Regarding port types: Ports which contain different data types cannot be connected. (Different data types values are represented by different colours, but reference values are all coloured green).

3.2.2 Visual Functions

Definition 4 *Functions which can be called by the visual programming manager (see section 3.6) are called visual functions.*

Remark: Because in general, the visual environment does not know about mathematics, visual functions are not related to mathematical operations. We call all non-visual functions *auxiliary functions.*

A port is either an *inflow port* or an *outflow port*, as shown in figures 5 and 6. Furthermore, a port is either a *read port* or *write port*, shown in the same figures.

If port `x` is a read port, it must provide the visual function: `get_x()`, and if it is a write port, it must provide `set_x()`. However, in order to be able to inspect data values of ports, all ports are required to provide the visual function `get_x()`.

3.3 Modules

3.3.1 Introduction

This section regarding modules comments on the *basic modules*, modules which are not built from (an) other module(s). Modules built from other ones can be seen as networks, see section 3.5.

3.3.2 Visual Representation

We will visually represent a module from the canvas as a square or a rectangle. Modules that we use on the canvas are oriented, in the sense that we distinguish a top and a bottom of a module.
The top side of a module will be referred to as *inflow* and the bottom side as the *outflow*. All modules have an *inflow* and an *outflow*, *data ports*: *read ports* and/or *write ports*.

A *module* contains ports, as is shown in figures 5 and 6. By default, at least two ports are available, related to a C++ pointer to the module itself. These ports are called the `this-`*ports*, the related standard C++ pointer is called the `this-`*pointer*. In figure 6 these ports are green (the reference colour) and have a special horizontal division bar. In the notation employed in this thesis (see figure 5) there is no corresponding visual entity for this-pointers.

Modules can be connected to one another. The connections are made via their data ports, with specific simultaneous restrictions. The connections must be:

- between a read port and a write port (of identical types);
- between an inflow and an outflow port.

In principle the two ports could belong to the same module.

These restrictions leave a limited choice for possible connections between modules. Here are all four possible cases: (see figure 7) In (a) module **A** exports its this-port to module **B**. We say that **B** has (full access) to **A** because all of **A**'s members are available for **B**. Conversely, **A** has no access to **B**. In (c) on the other hand, the access rights are precisely opposite: **A** has full access to all of **B**'s members because **B** exports itself to **A**. In (b) and (d) we no not export `this-pointers` but single pieces of data. So in (b) module **A** exports only a part of itself to **B** and in (d) only a part of **B** is available to **A**.

3.3.3 Visual Functions

The visual functions of a module are (summarised):

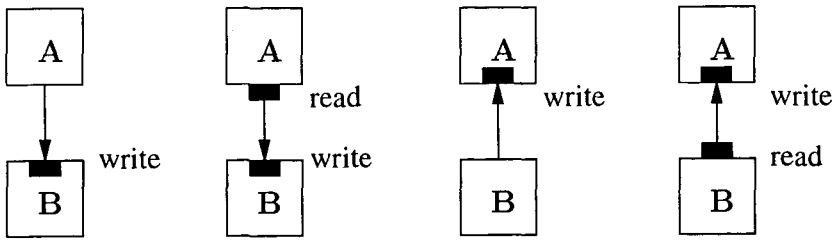


Figure 7: a) **B** has (access to) **A**, b) **A** has **B**, c) **B** has (access to) part of **A**, d) **A** has part of **B**

- `destruct()`
- `construct()`
- `update()`
- `clone()`

In fact, for reasons related to asynchronous execution – outside the scope of this thesis – it is better to use two functions `activate()` and `deactivate` instead of the single `update()`. The functions `destruct()`, `construct()` and `clone()` are not part of the visual programming language NumLab at the moment. Right now, these functions are derived from the standard C++ constructors and destructors, which can have undesired effects.

For further information, amongst the use of modules inside modules, we refer to [15].

3.3.4 Auxiliary Functions

All Scientific Computing numerical visual programs call auxiliary functions to do the numerical computations. The visual functions are never used to this end. A description of these functions would be best at its place in section 4, but is also omitted there due to space limitations. A more complete specification of these auxiliary functions could take another hundred pages.

3.4 Toolboxes

A *toolbox* contains modules. A toolbox is a standard C++ library complemented with additional content information. This information is incorporated via the C++ interpreter `cint`, and is later on required by the manager (the executioner of a network), which happens to be also `cint`.

3.5 Networks

A *network* is a collection of modules, and connections between ports. Thus, a network is a graph, nothing else. The execution of a network (see section 3.6) is a graph traversal problem.

In NumLab, a network is not a module though it can and should be. The algorithm which makes it a (super)module is as follows:

- Export all unconnected inflow ports to become inflow ports of the super module;
- Export all unconnected outflow ports to become outflow ports of the super module;
- Save and set all default (or interactor inputted) values as internal values;

3.6 Managers

As in classical programming languages, a manager executes a program and takes care of all input and output (i.e. user interaction).

3.6.1 Visual Representation

The visual representation of the network manager (in the case of the NumLab environment) is called *VISSION*. VISSION integrates a network manager and a *graphical user interface (GUI)*. The latter manages the canvas (drawing) and requests the integrated manager to execute specific visual functions.

The visual representation (i.e., VISSION) is quite complex. For more detail, we refer to [15].

3.6.2 Visual Functions

The manager's visual functions are:

- loading/unloading toolboxes of modules;
- loading/unloading networks;
- the execution of the visual functions of ports and modules,
- determined with the use of a network traversal algorithm.

Data is passed from one module to the other, executing the `get()` visual function from the exporting module and `set()` visual function from the receiver

A saved *network* can be and should have been C++ source, where source is defined in section 2.

In NumLab, a network is not C++ source perhaps because this would require the network manager (defined below) to scan the C++ source in order to figure out which visual components to be put on the canvas.

3.7 Network Execution

The manner in which to be built networks are executed, determines a large part of the design decisions for building a module library. An algorithm for network execution is given in [15], on pages 97 – 98. This section discusses the merits of the implementation of such an algorithm in VISSION.

The basic rule for the order of execution of a network is: from top to bottom, as shown in figure 8. Thus, modules at the top of the network are executed first. If there are

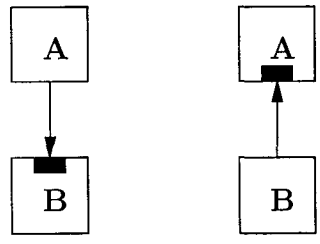


Figure 8: In both cases, **A** is executed before **B**

several top nodes, the order amongst them is not documented.

As commented on before, inflow ports must be connected to outflow ports and vice versa. Whether this is related to the traversal algorithm could not be located in [15].

In more complex networks as in figure 9, [15] does not comment on whether module **B** or **C** is first executed, before **D** is.

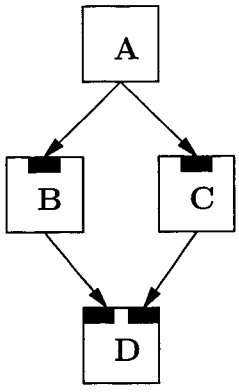


Figure 9: **A** before **B**, **C**; **D** after **B**, **C**; order of **B**, **C** undetermined

4 NumLab

4.1 Introduction

This section 4 introduces the *Numerical Laboratory NumLab* visual programming environment, used in order to implement all designs in sections 5, 6 and 7. First, section 4.2 describes NumLab in general terms. Next, section 4.3 describes module requirements in 4.3.1 and 4.3.2. Section 4.3.3 introduces some actual NumLab basic and derived modules. In conclusion, section 4.3.6 explains how NumLab models mathematical functions.

At the end of the section 4.4 discusses a few differences with DiffPack. For complementary information, see the Ph.D. thesis [15], and the manuals [3], [4] and [13].

4.2 Language and Manager

The NumLab workbench is a set of professional numerical and visualisation libraries, **managed** through a compiler, interpreter or network manager. Figure 10 shows the managers on top of the libraries. Thus, *NumLab* is both a *classical language* and a *visual language*. That is, its modules can be used in a network and be managed by a network manager, or alternatively, be used in a source and be managed by a program executioner. In short, the best of both visual and classical programming styles can be combined.

4.3 Modules

It requires large programming skills to solve complex numerical Scientific Computing tasks using libraries of functions. So, *the* requirement for a good module is that much

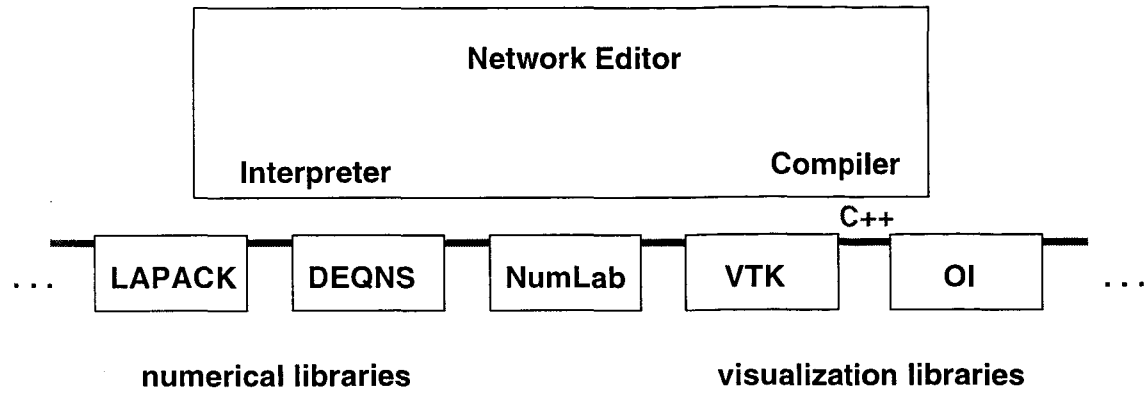


Figure 10: NumLab depicted as a set of managers, and related libraries

less skill is required for the composition of a complex numerical solver.

4.3.1 User Requirements

The list of user-related requirements contains at least:

- Simple, common and orthogonal approach;
- Modules should behave as expected;
- Modules should be identical to functions (or data);
- Networks should behave as expected;
- Networks should be identical to sources;
- Replacing modules at runtime must be possible;
- Inspecting modules (data) at runtime must be possible;
- Average algorithms use about 10 modules.

The order in which these requirements are met by the NumLab visual programming language are not discussed here due to a lack of space.

4.3.2 Technical Requirements

The list of technical (machine/language-related) requirements includes:

- Use and wrap in a common language C++ (data-dump, OpenMath, MathML);

- Reuse modules via containment and extension;
- Reuse programs in different languages (Babel problem);
- Efficiency and computability etc., etc., etc.

Also here, the order in which these requirements are met, is not further discussed.

4.3.3 NumLab's Basics

This section briefly describes some of NumLab's basic modules, of importance for Scientific Computing. These are the modules closest involved for the implementation of loop control and BOUNDPACK modules.

These most important basic modules are:

- Matrix;
- Vector.

Vectors are in fact block vectors, and to be more precise vector valued functions (*vector functions*). However, the vector functions are not based on a symbolic kernel. The integration of symbolic (algebraic) information is beyond the scope of this thesis.

Matrices are in fact block matrices of which each block, independently, can be:

- full;
- sparse;
- diagonal.

More precisely, matrices are operators on vector functions, which provide certain basic operations. One of the operations is that of computing a Jacobian matrix, itself a (linear) operator on vector functions (so also represented with a matrix).

Because vectors and matrices in fact are argument (vector function) and operator (operator on vector function), the actual module names are `FunctionVector` and `OperatorImplementation`. The names `Vector` and `Matrix` are aliases for the uninitiated.

In fact, NumLab derived modules can handle quite complex tasks – albeit slow without speed optimisation:

- *partial differential equations PDE's* (boundary value problems);
- *ordinary differential equations ODE's* (initial value problems);
- (non-linear) equations;
- (non-linear) solvers;
- (non-linear) preconditioners, etc.

4.3.4 Design

The `Operator` and `FunctionVectors` modules from the previous section share an identical *operator design* (so behave identical towards their user), called the *NumLab operator module*, shown in figure 11.

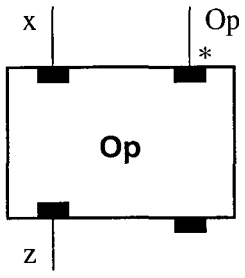


Figure 11: NumLab operator and datum

However, at this moment (NumLab version 2.72),
`FunctionVector` does not have the `x` and `z` ports mentioned above. The reason is a former limitation of the `cint` C++ interpreter.

As is shown in figure 10, NumLab also contains visual libraries. Whereas the basic numerical NumLab modules in section 4.3 were designed from scratch according to the brand-new operator/datum design, the visual libraries are based on reusing C++ sources (*Visualisation ToolKit (VTK)* and *Open Inventor (IV)*). In fact, the visualisation libraries (by Telea) have been the first sources so be integrated as modules.

Though some of the NumLab modules integrate source (the eigenvalue module integrates for instance two FORTRAN 77 LAPACK functions) most modules do not because no clear process existed for the integration of FORTRAN 77 sources. This process has been developed as a part of and during the writing of this thesis.

Furthermore, for instance, PDE operator modules can be connected to ODE modules in

order to solve transient problems. In this sense, all modules are orthogonal with respect to each other.

4.3.5 Data Modules

Figure 12 shows two possible representations for a FORTRAN 77matrix (with dimesnions m and n) to be read and exported to function **F** use in the C++ language (NumLab uses the second representation). The network in (a) has a matrix reader and exports its data explicitly to a matrix, an integer m and an integer n. Next, these three modules are connected to **F**. Alternatively, we can create one module that reads a matrix and its dimensions and keeps it within its own compound.

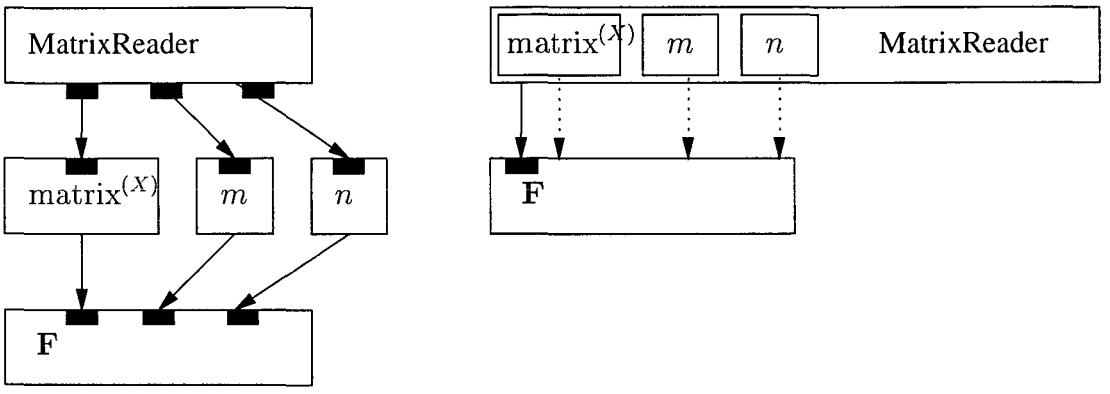


Figure 12: a): No *data-hiding* b): *data-hiding*

The second representation seems the desirable one: Instead of five modules and 6 links, just two modules and one link are required. Here, **F** retrieves m and n using its pointer to the `MatrixReader` module. It should be clear that though *data-hiding* reduces the amount of connections, modules can no longer be reused in another context.

4.3.6 Function Modules

Because the most important basic module in NumLab represents an operator (regarding non-visual functions) it is important to address the relation ship between *function input and output arguments* on the one hand and *inflow/outflow and read/write ports* on the other. Presented below, this relationship is perhaps not the one anticipated.

The mathematical relation

$$\mathbf{y} = \mathbf{F}(\mathbf{x}) \tag{2}$$

maps to the NumLab visual network: The design is explained below. In NumLab, a module **F** which implements $z = f(x)$ must have at least two ports:

- A port for the *input variable* x, called *write port*, or `x`;
- A port for the *output variable* z, called *write port* `z`;

The module must implement:

- function `get_x()`, `set_x()`, so it can (dis)connect `x`;
- function `get_z()`, `set_z()`, so it can (dis)connect `z`;

Note that the input argument and output argument are **both** *write ports*, as is shown in figure 13.

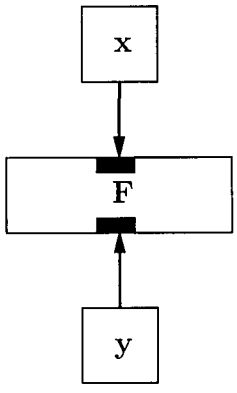


Figure 13: The NumLab visual representation of $\mathbf{y} = \mathbf{F}(\mathbf{x})$

Using the design for a function **F**, the concatenation

$$\begin{aligned} \mathbf{y} &= \mathbf{F}(\mathbf{x}) \\ \mathbf{z} &= \mathbf{G}(\mathbf{y}) = \mathbf{G}(\mathbf{F}(\mathbf{x})) \end{aligned} \tag{3}$$

is imitated with the visual program as shown in figure 14:

Of course, it is important the reduce the amount of modules on the canvas keeping independencies as they are. Figure 15 shows a design alternative to figure 14 using less modules and introducing no new dependencies. Though the design sofar seems straightforward, section 5 shows its limitations. A more detailed explanation falls beyond the scope of this thesis.

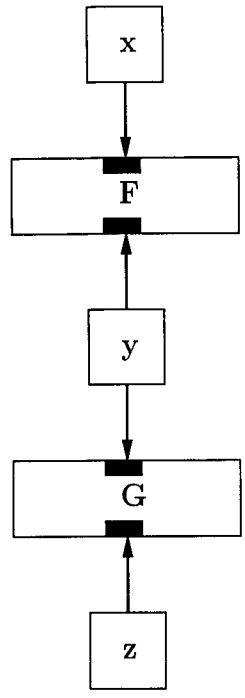


Figure 14: The NumLab visual representation of $\mathbf{z} = \mathbf{G}(\mathbf{F}(\mathbf{x}))$

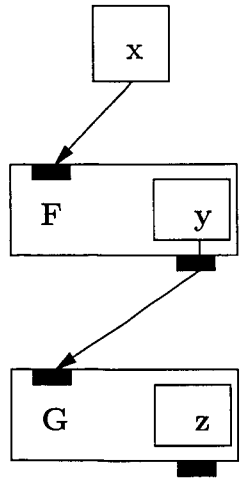


Figure 15: A sparser network

4.4 Other Visual Approaches

There are several other systems and more claims to visual programming, each with its own merits. For mathematicians, a free available package is the differential equation package *DiffPack*.

The DiffPack environment, and others, do not have the flexible common operator basis employed by most NumLab modules. Instead, there is much more differentiation right at the basis, which means a loss of orthogonality:

- Solvers can employ one other solver, which can **not** employ other solvers!
- The fundamental module is the linear system module, all others are either extensions, or have nothing in common.

For more information on this topic and for comparisons of NumLab to different visual programming platforms, see [6].

5 From Library to Toolbox

5.1 Languages

This section deals with the *(re)use of FORTRAN 77 numerical software libraries* inside a (NumLab) visual programming environment.

The first problem to be addressed is the Babel-problem. Two different languages are involved, in our (new) design which uses three levels:

- The FORTRAN 77 level, providing the functions (called source level);
- An intermediary level, with all source level functions and related data in C++ (called class level);
- The module level, adding the visual functions to all items in the class level C++ (called module level);

A typical FORTRAN 77 function `fx` at source level is said to be wrapped to class `NFx` at class level. This class `NFx` is wrapped to module `MFx` at module level. Both transformation require specific designs, see section 5.1.3 for more information.

The second problem invloves the relation between FORTRAN 77 source function and the design of the corresponding module. It turns out to be difficult to answer the following

simple question: Is data situated (inside) function modules, or is it outside on the canvas? Section 5.2 demonstrates the various solutions their advantages and related problems.

Summarised, in section 5.1.1, we inspect the basic building blocks of computer languages, from a mathematical point of view. Next, we discuss the transformation of one language to another in section 5.1.2. Finally, we present out process for using FORTRAN 77 source in the visual programming language NumLab.

5.1.1 Definitions

Reusing a library written in one computer language, in another language, requires a kind of translation, also called *wrapping*. In order to understand the possible translation problems, it is necessary to have a precise knowledge of what is meant with *computer language*. This section's definition of computer language agrees with the notion of language as seen in for instance OpenMath (see for instance the literature list in [2]).

Let S be the set of all strings (finite sequences of characters), as defined in [7].

Definition 5 *A (computer) language consists of the following components:*

- *a set of elements* $x \in X$;
- *functions* $f^X : X \mapsto X$;
- *readers and writers* $R_X : S_X \subset S \mapsto X$ *respectively* $W_X : X \mapsto S_X \subset S$;
- *a set of control structures* C_X *(*`while`, `if ... then ... else ... endif`*)*

Strictly spoken, the readers and writers are not functions, unless $X = S$. For more information on reading and writing to strings, we refer to [15]. Please, look for the term *serialisation*.

Real programs deal with multiple data types $x \in X$, $y \in Y$ and $z \in Z$, with related functions, etc. Assume $S_X \subset S_Y \subset Z$. Then the function $\mathbf{F} : (X, Y) \mapsto Z$ is realised with a combination as shown in figure 16: This is how *recent new visual programming toolboxes in NumLab (BOUNDPACK, LAPACK and SEPRAN) communicate.*

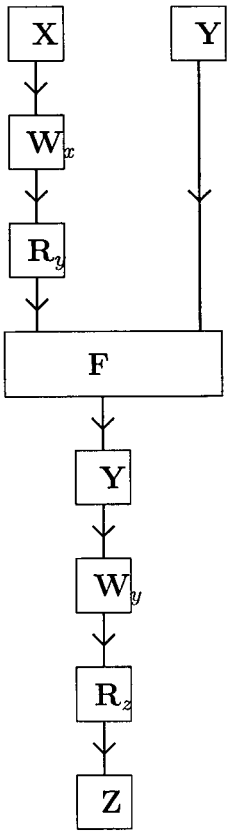


Figure 16: $Z = \mathbf{F}(X, Y)$

5.1.2 Transformation to Language

Look at the following diagram which describes the reuse of software written in X inside programming language Z: For simplicity this diagram assumes all data types and functions represented in one language can be represented in another – often not the case.

The goal is to have an implementation in the language Z, (re)using the one in X. We distinguish reusing data, functions and control structures:

1. Reusing $x \in X$ with the use of conversion E and its inverse;
2. - Reusing f^X: $f^Z(z) = E(f^X(E^{-1}(z)))$;
   - No reuse f^X: $f^Z(z)$ computed at the spot (for instance $x \mapsto \sin(x)$)
3. - No reuse of control structures

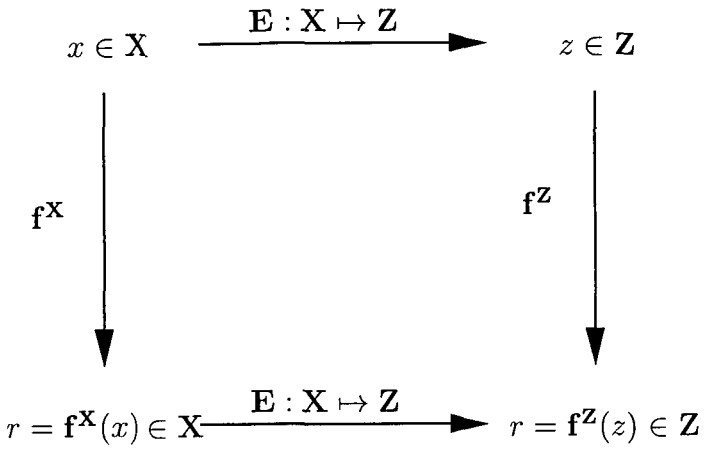


Figure 17: Mathematical diagram

Next, we would like a pure functional visual implementation $r = f(x)$ (to stick close to the mathematics). However:

- Languages such as FORTRAN 77 do not permit this: Example: The function f which for $\mathbf{x} = [1, 2, 3, 4]$ returns $\mathbf{r} = [1, 2]$ can not be coded in FORTRAN 77 without side effects.
- Functional languages are often less efficient.

Thus, $r = f^X(x)$ is denoted with `void` $f^X(r, x)$, or `void` $f^X(r_1, r_2, ..., r_k, x_1, x_2, ..., x_n)$ This leads to the adapted diagram in figure 18.

5.1.3 Transformation to Toolbox of modules

This section describes the design for the generation of visual modules from FORTRAN 77 source.

This table shows the FORTRAN 77 source, and its equivalent on the next level (class), as well on the final level (module).

Language	F77	C/C++	Canvas
datum	—	datum	module
function	function	function	module
control	—	—	module

It should be interpreted as follows:

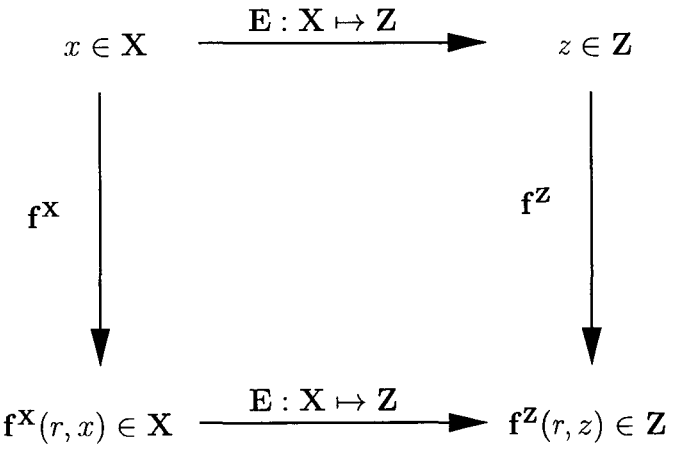


Figure 18: Diagram using implementational notation

For a given library in FORTRAN 77, we only reuse the functions on C++ level. They are wrapped to classes in C++, with a call back to the FORTRAN 77 source. FORTRAN 77 data types are not reused or wrapped. We build new classes to imitate the original data types and their usage on C++ level. Finally, on canvas level, both data types and functions from C++ appear as modules (What else?) with call backs to their respective predecessors.

The table shows that we do not reuse control structures from FORTRAN 77. The reason is that it is in fact pointless: a conditional statement as `IF .. THEN .. ELSE` has the same semantics as the C++equivalent `if ... then ... else`. The structure already exists, so there is no need to build it ourselves. The canvas does not dispose over control structures, and here we do need to create a control structure module.

We demonstrate the transformation by means of an example taken from the small LAPACK function toolbox (which contains three modules), based on a developed FORTRAN 77 data toolbox.

First, consider the *wrapping* for a double precision FORTRAN 77 vector. According to the table above, no FORTRAN 77 source is used. At the C/C++ level, we construct the double precision vector class `NDVector`. It is described in the file `NDVector.h`:

```
class NDVector : public NDBlock
{
public:
 NDVector(const int _n  = 0);
 double &operator()(const int _i);
```

```
 NDVector(const NDVector &);
 NDVector &operator=(const NDVector &);

 int read(const char *, const int format = ReadWriteFormatMatlab);
 int write(const char *, const int format = ReadWriteFormatMatlab) const;
};
```

Note that `NDVector` does not have its own data field `double` to store its values. This is in fact not the case: `NDVector` has been derived from the base class `NDBlock` (also used for matrices and tensors) that does dispose over this data field.

The provided operations for this class are entry-level access, creation and duplication.

On the next module level, the description is:

```
class MDVector : public MModule, public NDVector
{
public:
 MDVector(const int _x = 0);
 MDVector(const MDVector &);
 MDVector &operator=(const MDVector &);

public:
 int update();

public:
 // network: input/output port methods:

};
```

Note that the vector on the module level is (derived from) a vector on the C/C++ level, and that the addition *visual functions* have been added.

Next, consider the *wrapping* of a FORTRAN 77 function. In order to keep the example simple, we did not use LAPACK, but instead used a simple function called `dvectorplus`, which adds to vectors. Naturally, this does not change the principle.:

```
      SUBROUTINE dvectorplus(z, x, y, n)
c     ----------------------------------
      IMPLICIT none
      INTEGER n
      DOUBLE PRECISION z(n), x(n), y(n)

      INTEGER i
```

```
      DO 1 i=1,n
       z(i) = x(i)  + y(i)
    1 CONTINUE

      RETURN
      END
```

Here is the class for this function. From NDVectorddPlus.h:

```
class NDVectorPlus
{
public:
 NDVectorPlus();
 NDVectorPlus(NDVector &z, const NDVector &x, const NDVector &y);

protected:
 // call back on source
 void callback(NDVector *, const NDVector *, const NDVector *);

public:
 ostream      &print(ostream &) const;
};
```

Observe the member function `callback ()`. This is *the* link with the FORTRAN 77 source.

On canvas level, we have (from MDVectorPlus.h):

```
class MDVectorPlus: public MModule, public NDVectorPlus
{
public:
 MDVectorPlus();
 MDVectorPlus(MDVector &_z, const MDVector &_x, const MDVector &_y);

public:
 int update();

public:
 // Input/output port methods:
 const MDVector *getx();
 void            setx(MDVector *);
 const MDVector *gety();
```

```
 void            sety(MDVector *);
 const MDVector *getz();
 void            setz(MDVector *);

private:
 MDVector *x;
 MDVector *y;
 MDVector *z;
};
```

Observe that the class implementation preserves the function notation: It is possible to write a program

```
 NDVector x;
 NDVector y;
 NDVector z;
 NDVectorPlus(z, x, y);
```

without "instantiating" the `NDVectorPlus` module. As in the case of wrapping the data, on the module level, all visual functions are added.

An example taken from the last design toolbox (small LAPACK one, after the BOUNDPACK toolbox had been implemented) has not been inserted in this place. The LAPACK toolbox is small and available upon request.

5.2 Module Granularity

A source can be translated into visual modules using different levels of granularity:

- source to single module (plain wrap);
- source to n modules $\mathbf{M}_1, \ldots \mathbf{M}_n$, one for each function;
- source to n modules $\mathbf{M}_1, \ldots \mathbf{M}_n$, with additional external loops (see section 6);

Figure 19 shows how a FORTRAN 77 source can be wrapped as one module (left) or split into several ones (right). Next, we must decide whether the individual modules should hide their input and output data as in figure 12(b), or make use of external data as in figure 12(a).

First, a typical case of data-hiding has been applied in modules developed for the BOUNDPACK visual library. The disadvantage is that each module must be connected to the next one in order to get all data it requires. Thus reuse possibilities are low.

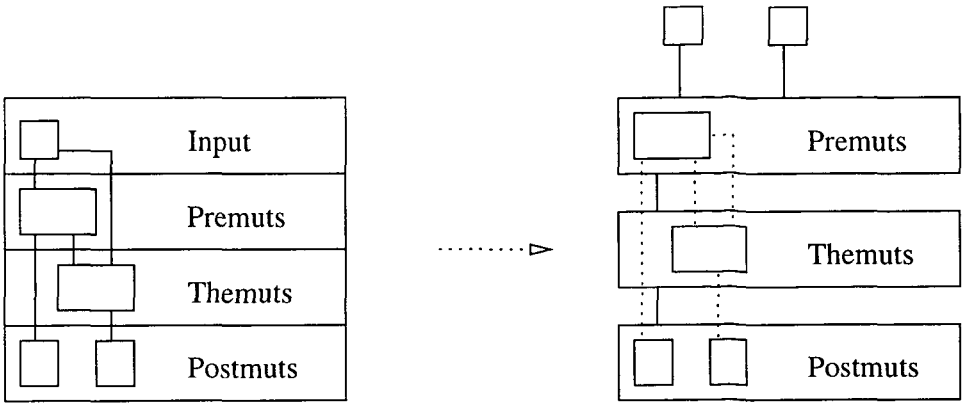


Figure 19: Dividing a test program into four separate parts

Next, in an attempt to produce reusable modules from the FORTRAN 77 source, no data-hiding was applied (figure 12(a)). Figure 20 shows how a network might look in theory. This may lead to too many data modules will reside on the canvas. This can

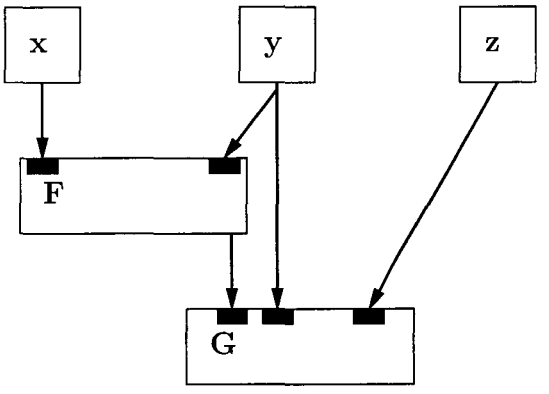


Figure 20: FORTRAN 77 style data flow program, using external data

also be seen in the BOUNDPACK example shown in figure 21.

Though already commented on in section 4.3.5 and 4.3.6, it is clear that a network with so many modules and connections does not add to the overview and simplicity of visual programming.

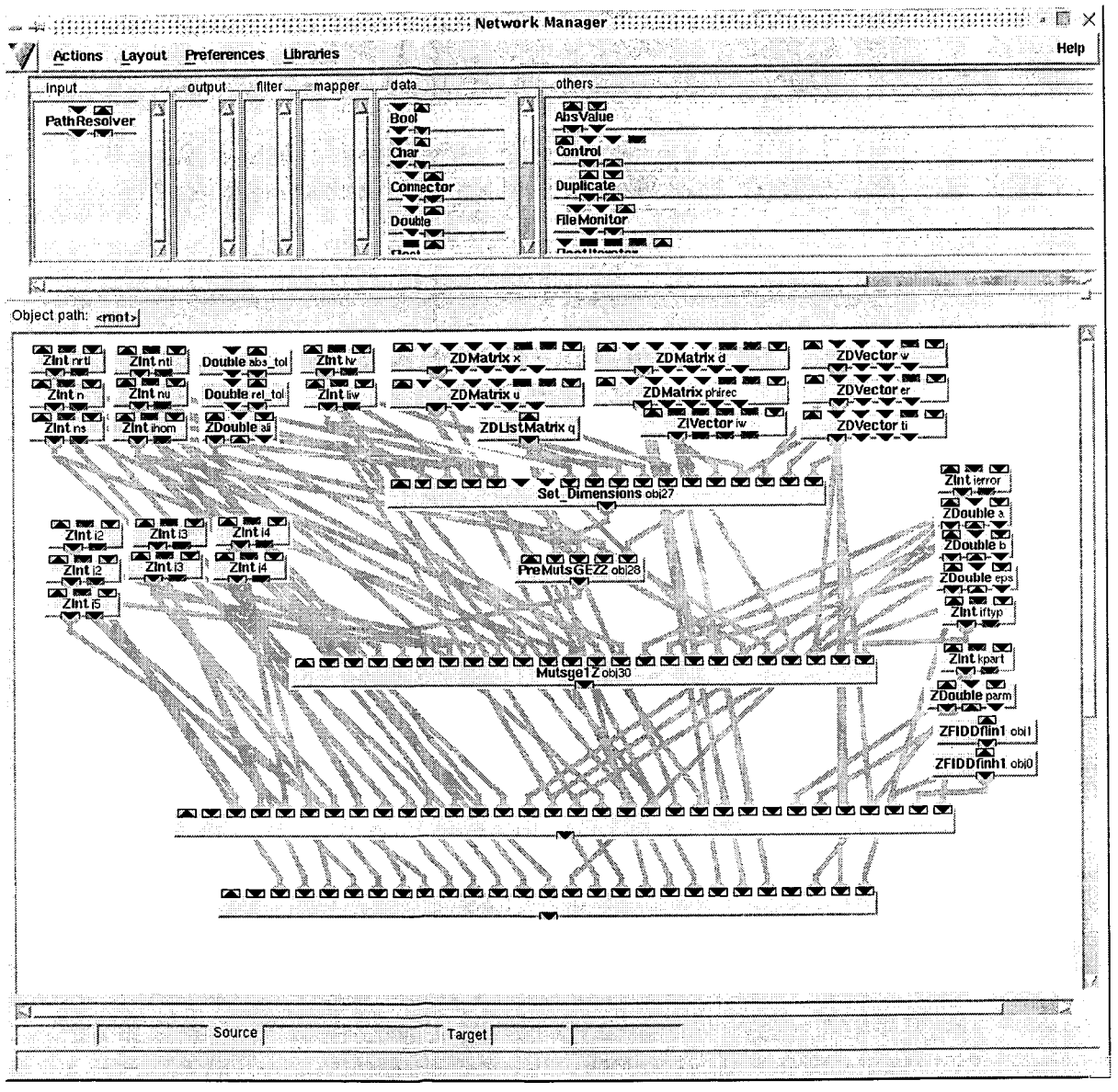


Figure 21: The routine MUTSGE split into different *reusable* components

5.3 Toolbox Communication

When tens of toolboxes are to be used in combination, their independence becomes an important issue. This independence ensures that new toolboxes can be developed without all others. Thus, speed, and reliability increases.

Assuming that two toolboxes have been developed independently, than how would they communicate? There are several possible solutions, including:

- filesensor;
- exchange data via standards at file level;
- exchange data via shared data types.

In NumLab, the standard matrix and vector data structures are those used by *MATLAB*. The standard data formats for vector functions and operators are those used by VTK.

At first glance, it could seem strange that independently developed toolboxes would have data types in common. However, all toolboxes can make use of the builtin types such as `int`, or `string`, which means that a writer from one toolbox can be coupled to and reader from another via string arguments.

The BOUNDPACK toolbox implementation (as part of this thesis) and a derived small LAPACK toolbox communicate via a combination of shared data types and files.

In the current NumLab kernel toolbox, the basis modules `vector` and `matrix` communicate with the visualisation toolbox via a direct link between the two underlying libraries, which makes debugging awkward. This should be altered for future releases.

6 Modules for Loop Control

6.1 Introduction

The addition of *control structures* seems to complement the set of mathematical data and functions with mathematical *logic*. Through all applications, also in *OpenMath* and *MathML*, the implementation of control structures seems the most difficult to implement.

6.2 Data Flow Management, when loops occur

The order of execution of pieces of code

```
subroutine1(...)
...
subroutineN(...)
```

in standard programs, is sequential. When started, subroutine1 is first executed and upon finishing, subroutine2 is executed and so forth until subroutineN finishes and the program terminates.
On the canvas – data flow style programming – the connections (data flow graph of the program) determine the order of execution. There is no sense of "sequential", in the classical programming sense.
A typical program with a loop looks like

```
pre;
while (cond(calculated_value, input_value) do something;
post;
```

We need to divide the code into separate parts, if we want to make the loop explicit on the canvas, i.e. construct a loop module and a condition module that controls the number of loops.
The program is best cut into three pieces as follows:

```
pre module;
inside loop module; /* contains all cond_input_value related stuff */
cond module;
post module;
```

with an added control block `cond`. The corresponding network is schematically shown in figure 22:

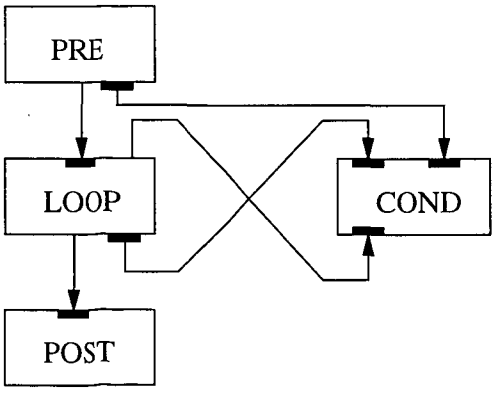


Figure 22: Schematic data flow through a loop

Here, whether the data flow comes from `pre`, or from `cond`, `loop` executes the same code. When `cond` stops, the loop back simply stops and the network terminates. Note that the data flow loops from the bottom of the `loop` module to the `cond` module, enters it through the top and leaves again through the bottom. This is precisely where our intuition works against us.
There is a connection between `cond` and `pre` as well. It serves to supply `cond` with a iteration limit, equal to the amount of memory allocated to store the solution in. We avoid "out of bounds" problems by doing so.
The NumLab network is shown in figure 23.

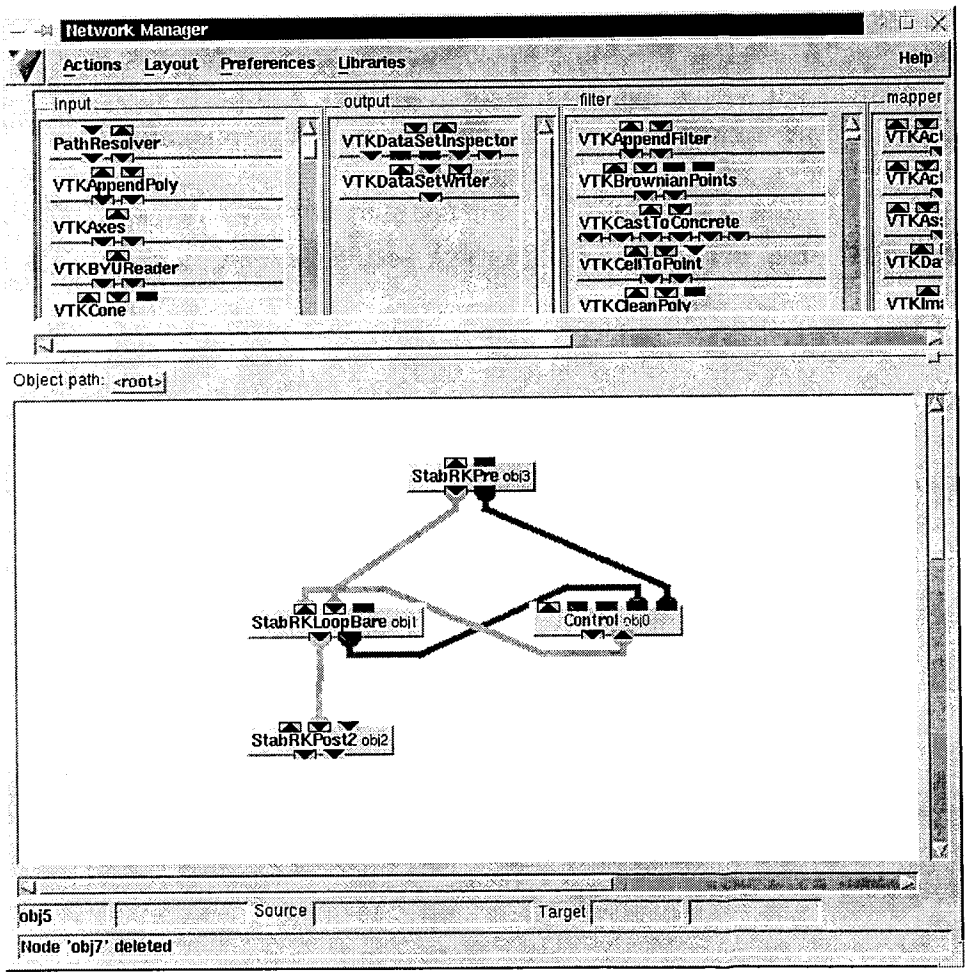


Figure 23: NumLab network with an explicit loop

In this particular case, order of execution is determined by the connections. A typical

update for each of the modules described above could be:

```
Update M:
{
 Module M depends on I_i

 // If I_i has failed, return NetworkHalt;
 // If I_i has never been updated before, do so now.
 // Upon NetworkHalt, return NetworkHalt;

 // Here, we now that I_i is an a Ok state.
}
```

It turns out that in a situation like the network above, it is important to know where the data flow comes from. In other words, we need to know for every module what module was executed before.
Unfortunately, general networks may be more complicated with modules that depend on more than a single module. Consider the network from figure 24.

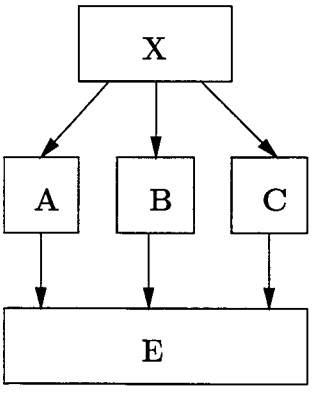


Figure 24: Data flow network in a fragment of a bigger network

Module E now depends on module A, B and C who all depend on module X. Asking the question "Who was last updated?" has no clear answer. After module X has finished, the network manager decides when and in what order to execute module A, B and C. It is impossible to tell who was last updated – either one is just as likely as the others, so it is not clear where the data flow before E comes from.
Another situation occurs in the network in figure 25.
In this network, the question "Who was last updated?" seems more relevant as A and C do not depend on the same module X as in figure 24. But this is only an apparent difference, because the network might be part of bigger network as shown in figure 26.

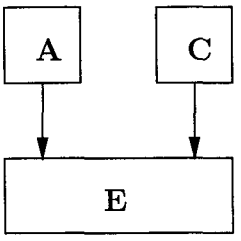


Figure 25: Data flow chart through a simple network

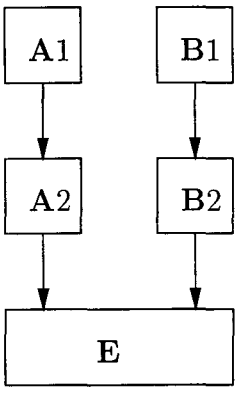


Figure 26: Who was last updated?

In this situation, again, we don't know. The order of execution is likely to be $A1$, $B1$, $A2$, $B2$, so the "Who was last updated" seems again irrelevant.
The problem discussed is in fact a *code switching problem*. The loop module is somehow supposed to decide whether it should execute its code for an iteration step, or the code for initialisation.
In figure 27 the left network is initialised once (module I), does not terminate, and loops back 10 times via module C. The right one is initialised 10 times, and terminates each time. Assume for both networks, we have arrived at module B, we want to know what to do. How does B know what to do? Do we have to initialise now, or has that been done before already and can we skip directly to the body of module B? In fact, what we want to know here is what has happened before B was called. In other words, who triggered module B? Was it I, then we need to initialise, or was it C and can we simply perform a next step in the loop? This simple question is important in dealing with situations with loops because we need to know what to execute: an initialisation of the loop, or repeat the body for a next step in the loop. This problem gives rise to the question "who was last updated?" but as we have seen in the examples above, it is a complicated task to find the answer to this question, if there is any, and if this answer contributes to solving

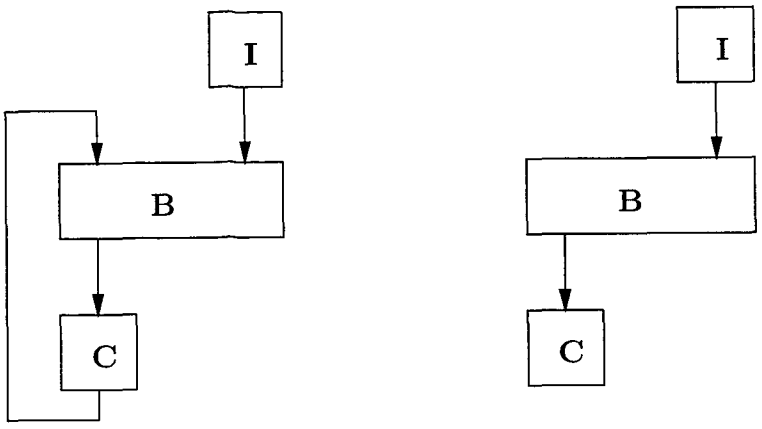


Figure 27: Who was last updated?

our initial problem: what code to execute?.

6.3 The Newton-Raphson Algorithm

In order to really put our previous considerations to the test, we created modules and built a NumLab network for the Newton-Raphson algorithm from section 2.2. It makes use of the especially for this purpose implemented NumLab modules:

- `Duplicate`, which copies its value from the input to output;
- `Increment`, which adds two values and overwrites the first with the sum;
- `Control` compares a real to a given accuracy and/or an iteration integer to a given max_iterations;
- `Newton` computes $\Delta \mathbf{x} = -F(x,a)/dF(x,a)$ for given $\mathbf{x}$ and a;
- `Fx`, which represents the function $f(x,a) = x^2 - a$;
- `dFx`, the derivative of Fx, $df(x,a) = 2x$;
- `AbsValue`, which takes the absolute value of its input;
- `ZDoubler`;
- `double`.

Schematically, this algorithm is turned into a network as shown in figure 28. Data flow

starts at $\mathbf{x}_0$. This value is duplicated to **x**. Next module `Newton` uses **x** and modules `Fx` and `dFx` to compute $\Delta\mathbf{x}$. This value is added to **x** through *incr*. On the other end, the absolute value is taken from $\Delta\mathbf{x}$, compared to input value `epsilon`, the required accuracy. From here, the network either loops back to **x** or the network terminates. At the end of computations, module **r** will hold the solution of the equation $x^2 - a = 0$, for given real $a > 0$.

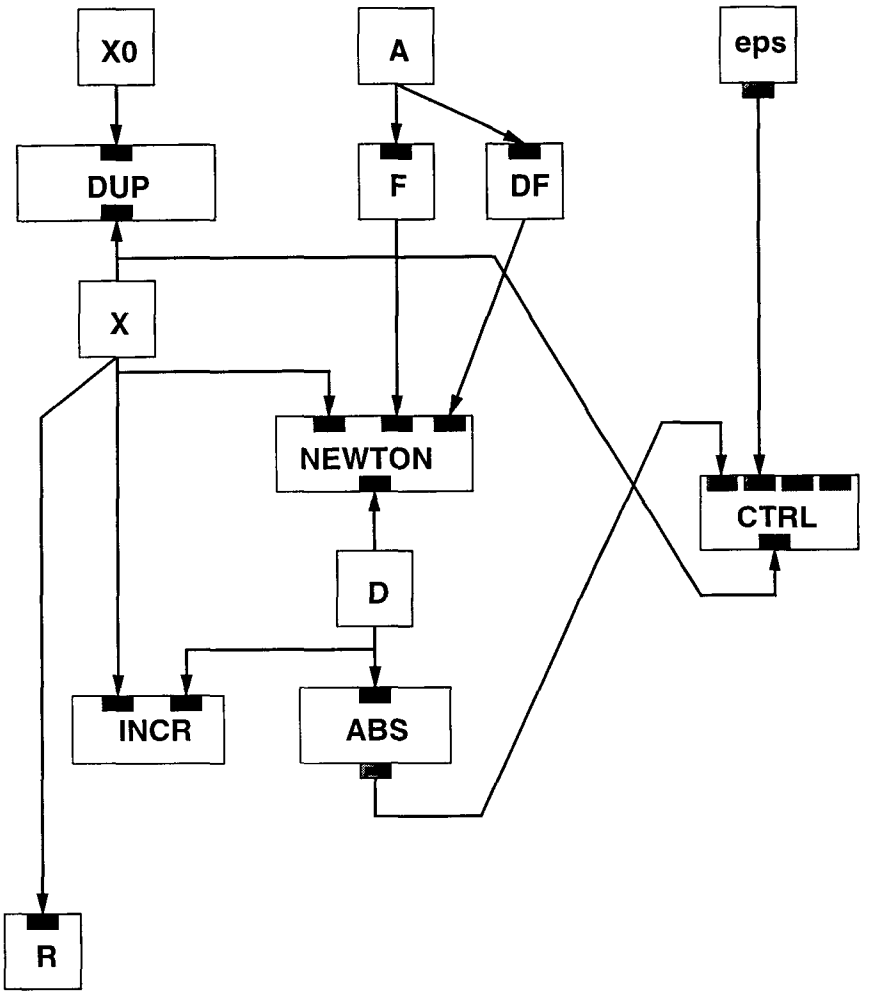


Figure 28: Newton network with external loop

The corresponding NumLab network for this algorithm is shown in 29.

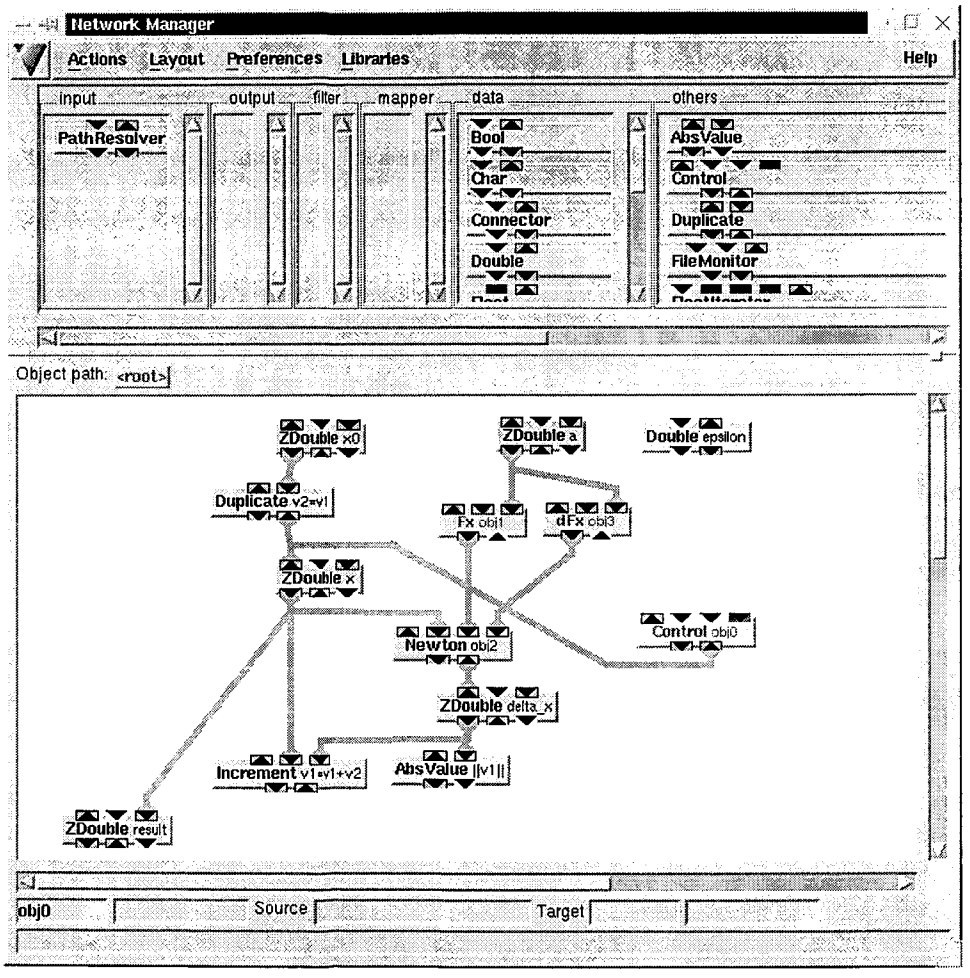


Figure 29: A NumLab Newton-Raphson network

6.4 The Newton-Raphson Algorithm: Restarting

Because loops seem difficult for the network manager to handle, restarting networks turns out to be not a trivial task. This section comments on problems encountered with the network manager (as implemented in NumLab).

Encountered Problems: During the implementation and testing of networks with loops, we encountered several problems with the network manager. When building such a network and sending data through it, it seemed impossible to re-initialise the network by calling the the initialisation module. Re-initialisation is needed when one alters input data, such as the number of output points or the method used. This would cause the loop module as well as the condition module (the guard) to end up in a endless cycle with both modules returning a networkhalt and calling one another. Of course, an obvious way around this problem is deleting the modules, and create them again and make the proper connections. However, we found way to restart the network without deleting and reconnecting nodes.

- Make the desired input changes
- Lock the condition module
- Lock the loop module
- Re-initialise, i.e. update the 'pre-block'
- Unlock the condition module
- Unlock the loop module

The network now restarts and calculates the new values. Perhaps a newer version of the canvas will cope with the problem.

7 Modules for BOUNDPACK

This section analyses the multiple shooting method and its implementation in the FORTRAN 77 library *BOUNDPACK*. It comments on the design of modules for this package, and discusses the merits of an actual implementation of this design.

7.1 Multiple Shooting Methods

7.1.1 Introduction

In this section we examine the *formal component specification* for multiple shooting methods. Such methods are used to numerically solve boundary value problems in one state variable.

We will initially discuss the principles of the multiple shooting method applied to ordinary differential equations. This method leads to a system of equations to be solved.

We will discuss some methods to find these solutions, among which QU-factorisation. Then we will develop and test the software to be able to use BOUNDPACK within the NumLab context and on the canvas. Finally we will focus on the multiple shooting method applied to some non-linear ODE's.

The shooting methods have been chosen based on the Scientific Computing Group's expertise in this field (see [8] and [1]) related available software (*BOUNDPACK* FORTRAN 77 libraries, implemented by Dr. G. Staarink and Prof. R. Mattheij), and a traineeship regarding the integration of time-integration methods in NumLab (see [16]).

Part of the presented material in this section is discussed in the books [8] and [1]. But, the presentation, using one consistent set of notations, all examples, and data flow analysis are original.
First we will define the boundary value problem we are interested in in section 7.1.2 and briefly explain how the multiple shooting method works. This will result into a linear system in section 7.1.3 that we will discretise in time in section 7.1.4. In sections 7.1.5 and 7.1.6 we will discuss some possibilities to solve the remaining problem. In sections 7.2 and 7.3 we will show how we can implement these methods.

7.1.2 The Boundary Value Problem

We are initially interested in second order linear ordinary differential equations (ODE): Given $x_a, x_b \in \mathbf{R}$

$$\begin{aligned} &\ddot{x}(t) = cx(t) + d\dot{x}(t) + f, \quad t \in [a,b] \\ &x(a) = x_a, \quad x(b) = x_b \end{aligned} \tag{4}$$

where c, d and f are in principle continuous functions of $t \in [a,b]$. Problem (4) can be reformulated as a first order ODE by writing

$$\mathbf{x}(t) = \begin{bmatrix} x(t) \\ y(t) \end{bmatrix} = \begin{bmatrix} x(t) \\ \dot{x}(t) \end{bmatrix} \tag{5}$$

By doing so, the second order differential equation becomes a first order ODE. In general, we can reduce a p-th order ODE into a first order system of p ODE's. The resulting system in our case is:

$$\begin{bmatrix} \dot{x}(t) \\ \dot{y}(t) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ c & d \end{bmatrix} \begin{bmatrix} x(t) \\ y(t) \end{bmatrix} + \begin{bmatrix} 0 \\ f(t) \end{bmatrix}. \tag{6}$$

In combination with the boundary values (4) we can denote the BVP shortly as:

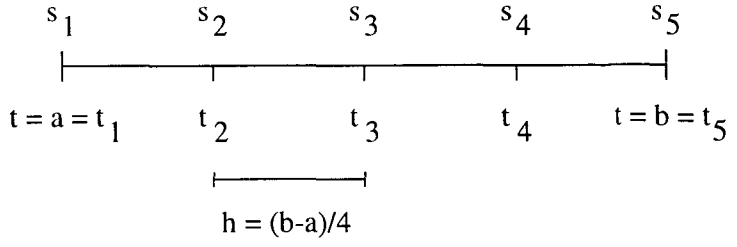
$$\begin{array}{ll} \dot{\mathbf{x}}(t) & = \mathbf{A}(t)\mathbf{x}(t) + \mathbf{f}(t), \quad t \in [a,b] \\ g(\mathbf{x}(a), \mathbf{x}(b)) & := \mathbf{B}_a\mathbf{x}(a) + \mathbf{B}_b\mathbf{x}(b) - \beta = 0 \end{array}. \tag{7}$$

with $\mathbf{A}(t)$, $\mathbf{B}_a$, $\mathbf{B}_b$, $\mathbf{f}(t)$ and β defined by

$$\begin{array}{lll} \mathbf{A}(t) = \begin{bmatrix} 0 & 1 \\ c & d \end{bmatrix}, & \beta = \begin{bmatrix} x_a \\ x_b \end{bmatrix}, & \mathbf{f}(t) = \begin{bmatrix} 0 \\ f(t) \end{bmatrix}, \\ \mathbf{B}_a = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}, & \mathbf{B}_b = \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}. & \end{array} \tag{8}$$

The shooting method applied to a boundary value problem involves temporarily ignoring one of the boundary conditions and simply integrating straight forwardly, starting from the remaining boundary condition. In the end we will have to make sure that a solution obtained in this way will eventually match the dropped boundary condition.

In order to apply the method of multiple shooting to our problem, we partition the time interval $[a,b]$ with the use of N, ($N \in \mathbf{N}$, $N \geqslant 2$) time nodes into $N-1$ subintervals, i.e. $[t_i, t_{i+1}]$ for $i = 1, 2, .., N-1$ with $t_1 = a$ and $t_N = b$. As an example, we will set $N = 5$. At the nodes, we introduce estimates $\mathbf{s}_i = [s_{i1}, s_{i2}]^T$ for the solution $\mathbf{x} = [x, \dot{x}]^T$ at t_i, ($i = 1, 2, ..., N-1$). Naturally, we would like to find $s_{11} = x_a$ in order to meet with the initial condition at $t = a$, but s_{12} (the derivative) can still be chosen freely so $\mathbf{s}_1$ is not fixed yet.



We have replaced the original problem with $N-1$ sub-problems. Our next step is solve these following family of problems on each of the intervals $[t_i, t_{i+1}]$ for $i = 1...N-1$

$$\begin{array}{ll} \dot{\mathbf{y}}_{\mathbf{s}_i}(t) & = \mathbf{A}(t)\mathbf{y}_{\mathbf{s}_i}(t) + \mathbf{f}(t) \\ \mathbf{y}_{\mathbf{s}_i}(t_i) & = \mathbf{s}_i \end{array}. \tag{9}$$

Any solution of the ODE (7) can be written as

$$\mathbf{x}(t) = \mathbf{Y}_1(t)\mathbf{a} + \mathbf{w}(t)$$

where $\mathbf{Y}_1(t)$ is a fundamental solution (i.e. $\mathbf{Y}_1(t)$ solves $\dot{\mathbf{Y}} = \mathbf{A}\mathbf{Y}$, $\mathbf{Y}(t_1) = \mathbf{I}$) and $\mathbf{w}(t)$ a particular solution of ODE (7) for some $\mathbf{a} \in \mathbf{R}^n$.

7.1.3 The Linearised System

In order to find a continuous solution, we will require matching conditions for the solutions $\mathbf{y}_{\mathbf{s}_i}(t)$ at the nodes $t = t_i$, $i = 2, 3, .., N-1$. In other words, we want $\mathbf{y}_{\mathbf{s}_i}(t_{i+1}) = \mathbf{s}_{i+1}$ for $i = 1, 2, ..., N-2$. Finally, we would like $\mathbf{s}_1$ and $\mathbf{y}_{\mathbf{s}_{N-1}}(t_N)$ to satisfy the boundary condition (4'), i.e. $g(\mathbf{s}_1, \mathbf{y}_{\mathbf{s}_{N-1}}(t_N)) = 0$. By writing $\mathbf{S} = (\mathbf{s}_1, ..., \mathbf{s}_{N-1})^T$ we can write the requirements above as $\mathbf{F}(\mathbf{S}) = 0$, with $\mathbf{F}$ specified by

$$\mathbf{F}(\mathbf{S}) = \begin{bmatrix} \mathbf{F}_1(\mathbf{S}) \\ \mathbf{F}_2(\mathbf{S}) \\ \vdots \\ \mathbf{F}_{N-1}(\mathbf{S}) \end{bmatrix} = \begin{bmatrix} \mathbf{s}_2 - \mathbf{y}_{\mathbf{s}_1}(t_2) \\ \mathbf{s}_3 - \mathbf{y}_{\mathbf{s}_2}(t_3) \\ \vdots \\ \mathbf{s}_{N-1} - \mathbf{y}_{\mathbf{s}_{N-2}}(t_{N-1}) \\ g(\mathbf{s}_1, \mathbf{y}_{\mathbf{s}_{N-1}}(t_N)) \end{bmatrix} = 0. \tag{10}$$

Keep in mind that in general the dimension of the domain of $\mathbf{F}$ is precisely $(N-1)p$, with $p = dim(\beta)$. For instance for problem (4) the dimension of $\mathbf{F}$ is $4 \times 2 = 8$.

We solve this system of equations by applying Newton:

$$\begin{array}{ll} \mathbf{F}'(\mathbf{S}^k)\triangle\mathbf{S}^k & = -\mathbf{F}(\mathbf{S}^k) \\ \mathbf{S}^{k+1} & = \mathbf{S}^k + \triangle\mathbf{S}^k \end{array} \tag{11}$$

where $\triangle\mathbf{S}^k = \mathbf{S}^{k+1} - \mathbf{S}^k$ with an initial $\mathbf{S}^0 = \mathbf{S}_0$ specified. It seems logical to take $\mathbf{S}_0$ such that $s_{11} = x_a$ according our boundary value at $t = a$.

$\mathbf{F}'$ is simply the Jacobian matrix $\mathbf{F}'_{i,j} = \frac{\partial \mathbf{F}_i}{\partial \mathbf{s}_j}$. Differentiation with respect to $\mathbf{s}_j$ is trivial for all elements except for those on the diagonal but even those can easily be simplified into

$$\mathbf{F}'(\mathbf{S}) = \begin{bmatrix} -\frac{\partial \mathbf{y}_{\mathbf{s}_1}}{\partial \mathbf{s}_1}(t_2) & \mathbf{I} & \emptyset & \cdots & \emptyset & \emptyset \\ \emptyset & -\frac{\partial \mathbf{y}_{\mathbf{s}_2}}{\partial \mathbf{s}_2}(t_3) & \mathbf{I} & \ddots & \emptyset & \emptyset \\ \emptyset & \emptyset & -\frac{\partial \mathbf{y}_{\mathbf{s}_3}}{\partial \mathbf{s}_3}(t_4) & \ddots & \emptyset & \emptyset \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \emptyset & \emptyset & \emptyset & \ddots & -\frac{\partial \mathbf{y}_{\mathbf{s}_{N-2}}}{\partial \mathbf{s}_{N-2}}(t_{N-1}) & \mathbf{I} \\ \mathbf{B_a} & \emptyset & \emptyset & \cdots & \emptyset & \mathbf{B_b}\frac{\partial \mathbf{y}_{\mathbf{s}_{N-1}}}{\partial \mathbf{s}_{N-1}}(t_N) \end{bmatrix}. \tag{12}$$

These equations determine the required solution of our linear system, but we still need to find the diagonal elements of the matrix $\mathbf{F}'$.
We will do this by solving the following system for $\mathbf{y}_{s_i}(t)$

$$\begin{array}{ll} \dot{\mathbf{y}}_{\mathbf{s}_i}(t) & = \mathbf{A}(t)\mathbf{y}_{\mathbf{s}_i}(t) + \mathbf{f}(t) \\ \mathbf{y}_{\mathbf{s}_i}(t_i) & = \mathbf{s}_i \end{array} . \tag{13}$$

In the following sections we will show two methods to solve this problem.

7.1.4 The Time Integration

Because usually $\mathbf{y}_{\mathbf{s}_i}$ and $-\frac{\partial \mathbf{y}_{\mathbf{s}_i}}{\partial \mathbf{s}_i}$ for computation of $\mathbf{F}$ can not be computed exactly, we need to discretise the problem in time. We have several choices here but we chose to use the θ-method with only **one** step, i.e. $h = \frac{b-a}{N-1}$. Define for $P, Q \in \mathbf{R}^{2\times 2}$

$$P/Q := Q^{-1} \cdot P$$

and define for $1 = 1...N$

$$\begin{array}{lll} \mathbf{A}_i & := & \mathbf{A}(t_i) \\ \mathbf{f}_i & := & \mathbf{f}(t_i). \end{array}$$

Then we find with the $\theta-$method

$$\frac{\mathbf{y}^h_{\mathbf{s}_i}(t_{i+1}) - \mathbf{y}^h_{\mathbf{s}_i}(t_i)}{h} = \theta\{\mathbf{A}_i \mathbf{y}^h_{\mathbf{s}_i}(t_i) + \mathbf{f}_i\} + (1-\theta)\{\mathbf{A}_{i+1}\mathbf{y}^h_{\mathbf{s}_i}(t_{i+1}) + \mathbf{f}_{i+1}\} \tag{14}$$

and from this

$$\begin{array}{lll} \mathbf{y}^h_{\mathbf{s}_i}(t_{i+1}) & = & \frac{I+\theta h\mathbf{A}_i}{I-(1-\theta)h\mathbf{A}_{i+1}}\mathbf{y}^h_{\mathbf{s}_i}(t_i) + \frac{\theta h\mathbf{f}_i + (1-\theta)h\mathbf{f}_{i+1}}{I-(1-\theta)h\mathbf{A}_{i+1}} \\ & = & \frac{I+\theta h\mathbf{A}_i}{I-(1-\theta)h\mathbf{A}_{i+1}}\mathbf{s}_i + \frac{\theta h\mathbf{f}_i + (1-\theta)h\mathbf{f}_{i+1}}{I-(1-\theta)h\mathbf{A}_{i+1}}. \end{array} \tag{15}$$

We will repeat this procedure now for the perturbed problems for $\mathbf{z}_{\mathbf{s}_{1,j}}, ..., \mathbf{z}_{\mathbf{s}_{N,j}}$, $(j = 1, 2)$, defined by:

$$\begin{array}{lll} \dot{\mathbf{z}}_{\mathbf{s}_{i,j}} & = & \mathbf{A}(t)\mathbf{z}_{\mathbf{s}_{i,j}} + \mathbf{f}(t) \\ \mathbf{z}_{\mathbf{s}_{i,j}}(t_i) & = & \mathbf{s}_i + \delta\mathbf{e}_j. \end{array} \tag{16}$$

where we will take for $\mathbf{e}_j$ subsequently the unit vectors $\mathbf{e}_1 = [1,0]^T$ and $\mathbf{e}_2 = [0,1]^T$, thus $\mathbf{Y}_1(a) = I$. The variable δ is a small number, for instance equal to $\sqrt{\eta}$, the square root of the machine precision.

Applying the (same) θ-method to the perturbed problems results after similar steps as above in:

$$\begin{aligned} \mathbf{z}^h_{\mathbf{s}_{i,j}}(t_{i+1}) &= \tfrac{I+\theta h\mathbf{A}_i}{I-(1-\theta)h\mathbf{A}_{i+1}}\mathbf{z}^h_{\mathbf{s}_i}(t_i)) + \tfrac{\theta h\mathbf{f}_i+(1-\theta)h\mathbf{f}_{i+1}}{I-(1-\theta)h\mathbf{A}_{\mathbf{i+1}i+1}} \\ &= \tfrac{I+\theta h\mathbf{A}_\mathbf{i}}{I-(1-\theta)h\mathbf{A}_{\mathbf{i+1}}}\{\mathbf{s}_i+\delta\mathbf{e}_j\} + \tfrac{\theta h\mathbf{f}_i+(1-\theta)h\mathbf{f}_{i+1}}{I-(1-\theta)h\mathbf{A}_{i+1}}. \end{aligned} \tag{17}$$

As an estimate for $\frac{\partial \mathbf{y}_{\mathbf{s}_i}}{\partial \mathbf{s}_i}$ we can now take

$$\begin{aligned} \mathbf{Y}_{\mathbf{s}_{i,j}} &\doteq \tfrac{\mathbf{z}^h_{\mathbf{s}_{i,j}}(t_{i+1})-\mathbf{y}^h_{\mathbf{s}_i}(t_{i+1})}{\delta} \\ &= \tfrac{I+\theta h\mathbf{A}_\mathbf{i}}{I-(1-\theta)h\mathbf{A}_{\mathbf{i+1}}}\mathbf{e}_j. \end{aligned}$$

This way we can find all of the components of $\mathbf{F}'$. In sections 7.2 we will demonstrate the method in a special example.

7.1.5 The Direct Elimination (DE) Solution Method

For small sized problems, the equations (11) and (12) can simply be solved with a direct solution method, i.e. by multiplication with the inverse of $\mathbf{F}'(\mathbf{S}^k)$. So then we have

$$\Delta\mathbf{S}^k = -[\mathbf{F}'(\mathbf{S}^k]^{-1}\mathbf{F}(\mathbf{S}^k).$$

For larger problems, or simply generally speaking, calculating the inverse may not be efficient or not easy to find. We can then try a recursive method by defining $\mathbf{Y}_i := \frac{\partial \mathbf{y}_{\mathbf{s}_i}}{\partial \mathbf{s}_i}(t_{i+1})$ for $i = 1, 2, ...N-1$ and we 'sweeping' the matrix $\mathbf{F}'$, we can transform problem (11), (12) into

$$\begin{bmatrix} -\mathbf{Y}_1 & \mathbf{I} & \emptyset & \cdots & \emptyset & \emptyset \\ \emptyset & -\mathbf{Y}_2 & \mathbf{I} & \ddots & \emptyset & \emptyset \\ \emptyset & \emptyset & -\mathbf{Y}_3 & \ddots & \emptyset & \emptyset \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \emptyset & \emptyset & \dots & & -\mathbf{Y}_{N-2} & \mathbf{I} \\ \emptyset & \emptyset & \dots & & \emptyset & \mathbf{B_b}\mathbf{Y}_{N-1}+\mathbf{B_a}\mathbf{Y}_1^{-1}\mathbf{Y}_2^{-1}...\mathbf{Y}_{N-2}^{-1} \end{bmatrix} \Delta\mathbf{S}^k = -\tilde{\mathbf{F}}, \tag{18}$$

where $\tilde{\mathbf{F}}$ is

$$\tilde{\mathbf{F}} = \begin{bmatrix} \mathbf{F}_1 \\ \mathbf{F}_2 \\ \vdots \\ \mathbf{F}_{N-1}+\mathbf{B_a}\mathbf{Y}_1^{-1}\mathbf{F}_1+\mathbf{B_a}\mathbf{Y}_1^{-1}\mathbf{Y}_2^{-1}\mathbf{F}_2+\dots \\ +\mathbf{B_a}\mathbf{Y}_1^{-1}\mathbf{Y}_2^{-1}\dots\mathbf{Y}_{N-2}^{-1}\mathbf{F}_{N-2} \end{bmatrix}. \tag{19}$$

The system (18),(19) can be solved by means of a backward substitution. Starting at the last row, we can sequentially solve the equation and proceed to the row above. However, due to the appearance of the matrices in the last row, this is still an expensive computational task. Therefore, in the next section we will examine the QU method that uses the special structure of the matrix (12) in order to reduce the computation time.

7.1.6 The QU Solution Method

This section presents the QU solution method as part of the process of solving an ODE by mean of a forward/backward recursion. (see [8] and [1] for more details)

The target is to get detailed insight in the involved data-types and functions, for a possible formal component specification.

In the same manner that we can represent the solution of problem (7) by

$$\mathbf{x}(t) = \mathbf{Y}_1(t)\mathbf{a} + \mathbf{w}(t),$$

with $\mathbf{Y}_1(t)$ a fundamental solution ($\mathbf{Y}_1(t)$ solves $\dot{\mathbf{Y}} = \mathbf{AY}$, $\mathbf{Y}(t_1) = \mathbf{Y}_1$) we can do this for the solution but now considered as solutions belonging to the intervals. Thus we have for every interval i, $i = 1...N-1$:

$$\mathbf{x}(t) = \mathbf{Y}_i(t)\mathbf{a}_i + \mathbf{w}_i(t) \tag{20}$$

with $\mathbf{Y}_i(t)$ a fundamental solution ($\mathbf{Y}_i(t)$ solves $\dot{\mathbf{Y}} = \mathbf{AY}$, $\mathbf{Y}(t_i) = \mathbf{Y}_i$) and $\mathbf{w}_i(t)$ a particular solution on the interval $[t_i, t_{i+1}]$, all still to be determined.

Choose $\mathbf{x}(t_i)$ as starting condition, in other words, choose $\mathbf{w}_i(t_i)$ as starting condition. Often this choice will be $\mathbf{w}_i(t_i) = 0$. After having chosen the initial $\mathbf{w}_i(t_i)$ we can compute a fundamental matrix $\mathbf{Y}_i(t)$ (for example by numerically differentiation of the initial solution as we have done before). Now that we have a particular and a fundamental solution on each of the intervals, we will now determine the unknown $\mathbf{a}_i$ for $i = 1, 2, .., N$.

Remark : The possibility to choose the initial $\mathbf{Y}_i$ on every interval is based on the following fact: Suppose $\mathbf{Y}$ is a fundamental matrix such that

$$\begin{aligned} \dot{\mathbf{Y}} &= \mathbf{AY} \\ \mathbf{Y}(t_1) &= \mathbf{I}. \end{aligned} \tag{21}$$

Then

$$\begin{aligned} (\dot{\mathbf{YB}}) &= \dot{\mathbf{Y}}\mathbf{B} = \mathbf{AYB} = \mathbf{A}(\mathbf{YB}) \\ (\mathbf{YB})(t_1) &= \mathbf{Y}(t_1)\mathbf{B} = \mathbf{B}. \end{aligned} \tag{22}$$

Clearly we can choose freely the initial $\mathbf{Y}_i$. The fundamental solution belonging to this new problem is simply the principle fundamental matrix $\mathbf{Y}$ multiplied by our chosen $\mathbf{Y}_i = \mathbf{B}$.

Now start with $\mathbf{Y}_1 = \mathbf{I}$ and compute $\mathbf{Y}_1(t_2)$, this gives $\mathbf{x}(t_2)$. After QU decomposition of $\mathbf{Y}_1(t_2)$ we find

$$\mathbf{Y}_1(t_2) = \mathbf{Q}_2\mathbf{U}_2$$

with $\mathbf{Q}_2$ and $\mathbf{U}_2$ orthogonal and upper triangular matrices respectively.

If we continued integrating from the point $\mathbf{Y}_1(t_2)$ (i.e. if we chose $\mathbf{Y}_2 = \mathbf{Y}_1(t_2)$) then we would find exactly the same solution as with a single shooting method. The key point now is to choose a new orthogonal fundamental matrix to start from at $t = t_2$ so we take $\mathbf{Y}_2 = \mathbf{Q}_2$ and with this matrix we calculate $\mathbf{Y}_2(t_3)$. Then perform a new QU decomposition onto $\mathbf{Y}_2(t_3)$:

$$\mathbf{Y}_2(t_3) = \mathbf{Q}_3\mathbf{U}_3$$

with $\mathbf{Q}_2$ and $\mathbf{U}_2$ orthogonal and upper triangular matrices respectively.
In general on interval $[t_i, t_{i+1}]$ starting with an initial $\mathbf{Y}_i$, we calculate $\mathbf{Y}_i(t_{i+1})$ and apply a QU-decomposition

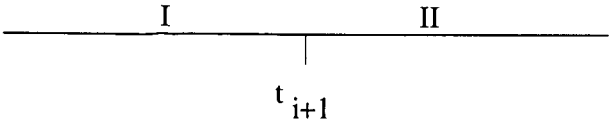
$$\mathbf{Y}_i(t_{i+1}) = \mathbf{Q}_{i+1}\mathbf{U}_{i+1}.$$

As the initial condition for the next interval $[t_{i+1}, t_{i+2}]$ we will choose $\mathbf{Y}_{i+1} = \mathbf{Q}_{i+1}$.
We can repeat this procedure until we have arrived at $t = t_N$.
Then, we have on the interval $[t_i, t_{i+1}]$ (cf I in the figure below)

$$\mathbf{x}(t_{i+1}) = \mathbf{Y}_i(t_{i+1})\mathbf{a}_i + \mathbf{w}_i(t_{i+1}) = \mathbf{Q}_{i+1}\mathbf{U}_{i+1}\mathbf{a}_i + \mathbf{w}_i(t_{i+1}) \tag{23}$$

and on the interval $[t_{i+1}, t_{i+2}]$ (cf II in the figure below)

$$\mathbf{x}(t_{i+1}) = \mathbf{Y}_{i+1}(t_{i+1})\mathbf{a}_{i+1} + \mathbf{w}_{i+1}(t_{i+1}) = \mathbf{Q}_{i+1}\mathbf{a}_{i+1} + \mathbf{w}_{i+1}(t_{i+1}). \tag{24}$$



Matching conditions at $t = t_{1+1}$ require that

$$\mathbf{Y}_i(t_{i+1})\mathbf{a}_i + \mathbf{w}_i(t_{i+1}) \overset{(23)}{=} \mathbf{x}(t_{i+1}) \overset{(24)}{=} \mathbf{Y}_{i+1}(t_{i+1})\mathbf{a}_{i+1} + \mathbf{w}_{i+1}(t_{i+1}) \Leftrightarrow$$
$$\mathbf{Q}_{i+1}\mathbf{U}_{i+1}\mathbf{a}_i + \mathbf{w}_i(t_{i+1}) = \mathbf{Q}_{i+1}\mathbf{a}_{i+1} + \mathbf{w}_{i+1}(t_{i+1}).$$

From (7.1.6) we find can derive a recursion for $\mathbf{a}_i$

$$\mathbf{a}_{i+1} = \mathbf{U}_{i+1}\mathbf{a}_i + \mathbf{d}_i \tag{25}$$

with

$$\mathbf{d}_i = \mathbf{Q}_{i+1}^{-1}\{\mathbf{w}_i(t_{i+1}) - \mathbf{w}_{i+1}(t_{i+1})\}.$$

The solution $\mathbf{a}_i$ of equation (25) can be represented as:

$$\mathbf{a}_i = \Phi_i\xi + \mathbf{z}_i,$$

if Φ_i is a solution of the homogenous problem of equation (25):

$$\Phi_{i+1} = \mathbf{U}_{i+1}\Phi_i \tag{26}$$

and $\mathbf{z}_i$ is a particular solution:

$$\mathbf{z}_{i+1} = \mathbf{U}_{i+1}\mathbf{z}_i + \mathbf{d}_i. \tag{27}$$

For well-conditioned problems it can be shown that the solution space S of the homogenous problem is dichotomic. This means that we can find a subspace (of dimension k, say) S_1 of S of solutions that do not increase significantly for decreasing t and a complementary $(n-k)$-dimensional subspace S_2 with solutions that do not increase significantly for increasing t. Under fairly general assumptions we can use this dichotomy and subdivide the matrix $\mathbf{U}_i$ as

$$\mathbf{U}_i = \begin{bmatrix} \mathbf{B}_i & \mathbf{C}_i \\ \emptyset & \mathbf{E}_i \end{bmatrix}, \tag{28}$$

with $\mathbf{B}_i$ and $\mathbf{E}_i$ both upper triangular square matrices (since $\mathbf{B}_i$ is upper triangular). Using the appropriate partitioning of $\Phi_i = \begin{bmatrix} \Phi_i^1 \\ \Phi_i^2 \end{bmatrix}$ we may split recursion (26) into:

$$\Phi_{i+1}^1 = \mathbf{B}_{i+1}\Phi_i^1 + \mathbf{C}_{i+1}\Phi_i^2 \tag{29}$$

$$\Phi_{i+1}^2 = \mathbf{E}_{i+1}\Phi_i^2. \tag{30}$$

The key factor now is that we will use the dichotomy to solve Φ_i and $\mathbf{z}_i$ stably by solving them sequentially in the right directions. Observe that $\mathbf{B}_i$ represents the incrementing modes of the solution (i.e. S_1) and $\mathbf{E}_i$ the decrementing modes (i.e. S_2). In order to minimise errors during computation, it is necessary to solve Φ_i^2 in forward direction given an initial Φ_1^2, for example we may take $\Phi_1^2 = [\emptyset|\mathbf{I}_{n-k}]$. We can now solve all Φ_i^2 for $i = 1, 2, .., N$. With these results and an initial $\Phi_N^1 = [\mathbf{I}_k|\emptyset]$ we can solve stably in backward direction the remaining $\Phi_{N-1}^1, \Phi_{N-2}^1, .., \Phi_1^1$ since $\mathbf{B}_i$ was said to represent the solutions

that do not increase significantly for decreasing t. This way we find $\Phi_1, \Phi_2, .., \Phi_N$. Note that this is in fact a decoupled recursion.
We use a similar approach to solve the particular solution $\mathbf{z}_i$ from recursion (27). With $\mathbf{z}_i = \begin{bmatrix} \mathbf{z}_i^1 \\ \mathbf{z}_i^2 \end{bmatrix}$ and $\mathbf{d}_i = \begin{bmatrix} \mathbf{d}_i^1 \\ \mathbf{d}_i^2 \end{bmatrix}$, we rewrite it as:

$$\begin{array}{rcl} \mathbf{z}_{i+1}^1 & = & \mathbf{B}_{i+1}\mathbf{z}_i^1 + \mathbf{C}_{i+1}\mathbf{z}_i^2 + \mathbf{d}_{i+1}^1 \\ \mathbf{z}_{i+1}^2 & = & \mathbf{E}_{i+1}\mathbf{z}_i^2 + \mathbf{d}_{i+1}^2. \end{array} \tag{31}$$

Note that all $\mathbf{d}_i$ are known already so starting at $\mathbf{z}_1^2 = 0$ we may find sequentially $\mathbf{z}_i^2$ for $i = 2, 3, ..., N$. Adding $\mathbf{C}_{i+1}\mathbf{z}_i^2$ to the source term $\mathbf{d}_{i+1}^1$ and starting with $\mathbf{z}_N^1 = 0$ we can solve in backward direction the remaining $\mathbf{z}_i^1$.
So now we have both the fundamental solutions Φ_i and the particular solutions $\mathbf{z}_i$.
Consider

$$\mathbf{a}_i = \Phi_i \xi + \mathbf{p}_i \tag{32}$$

for some still to be determined ξ. We will verify that the formula for $\mathbf{a}_i$ in fact satisfies recursion (25) by inspecting both $\mathbf{a}_{i+1}$ and $\mathbf{U}_{i+1}\mathbf{a}_i + \mathbf{d}_i$. We see that

$$\begin{array}{rcl} \mathbf{a}_{i+1} & \overset{(32)}{=} & \Phi_{i+1}\xi + \mathbf{p}_{i+1} \\ & \overset{(26),(27)}{=} & \mathbf{U}_{i+1}\Phi_i\xi + \mathbf{U}_{i+1}\mathbf{p}_i + \mathbf{d}_i, \end{array} \tag{33}$$

and

$$\begin{array}{rcl} \mathbf{U}_{i+1}\mathbf{a}_i + \mathbf{d}_i & \overset{(32)}{=} & \mathbf{U}_{i+1}\{\Phi_i\xi + \mathbf{p}_i\} + \mathbf{d}_i \\ & = & \mathbf{U}_{i+1}\Phi_i\xi + \mathbf{U}_{i+1}\mathbf{p}_i + \mathbf{d}_i. \end{array} \tag{34}$$

Indeed, it shows from (33) and (34) that the expression for $\mathbf{a}_i$ in (32) solves the recursion (25).
So after substituting the previous results into (20) we find for $\mathbf{x}(t_i)$:

$$\begin{array}{rcl} \mathbf{x}(t_i) & = & \mathbf{w}_i(t_i) + \mathbf{Y}_i(t_i)(\mathbf{z}_i + \Phi_i\xi) \\ & = & \mathbf{w}_i(t_i) + \mathbf{Q}_i(\mathbf{z}_i + \Phi_i\xi). \end{array} \tag{35}$$

Finally, we still have the unknown ξ. This value will follow after substitution of (35) into the boundary conditions $\mathbf{B}_a\mathbf{x}(a) + B_b\mathbf{x}(b) = \beta$:

$$\{\mathbf{B}_a\mathbf{Q}_1\Phi_1 + \mathbf{B}_b\mathbf{Q}_{N+1}\Phi_{N+1}\}\mathbf{c} = \mathbf{b} - \mathbf{B}_a\mathbf{w}_1(a) - \mathbf{B}_b\mathbf{w}_{N+1}(b) - \mathbf{B}_a\mathbf{Q}_1\mathbf{z}_1 - \mathbf{B}_b\mathbf{Q}_{N+1}\mathbf{z}_{N+1}. \tag{36}$$

7.2 The NumLab DE Implementation

As part of this thesis, the direct elimination method has been implemented. The merits of the DE-implementation are discussed using several different example boundary value problems.

Example 1: A Linear Homogeneous Problem.
Here we demonstrate the method of (single) shooting by solving the following begin value problem.

$$\begin{aligned} \ddot{x}(t) &= 0, \quad t \in [a,b] = [0,1] \\ x(0) &= 0, \quad x(1) = 1 \end{aligned} \tag{37}$$

or reformulated with the matrix notation from the section 7.1.2 as:

$$\begin{aligned} \dot{\mathbf{x}} &= \mathbf{A}\mathbf{x} \\ g(\mathbf{x}(0), \mathbf{x}(1)) &= \mathbf{B}_a\mathbf{x}(0) + \mathbf{B}_b\mathbf{x}(1) - \beta = 0 \end{aligned} \tag{38}$$

with $\mathbf{A}$, $\mathbf{B}_a$, $\mathbf{B}_b$ and β for this specific example

$$\mathbf{A} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \qquad \beta = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$
$$\mathbf{B}_a = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \quad \mathbf{B}_b = \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}.$$

Clearly the solution to this problem is

$$\mathbf{x}(t) = \begin{bmatrix} x(t) \\ \dot{x}(t) \end{bmatrix} = \begin{bmatrix} t \\ 1 \end{bmatrix}, \quad t \in [0,1]$$

but we will let NumLab find this solution by applying the shooting method. We will divide the interval $[0,1]$ in 4 subintervals, so $N = 4$. So we have $[t_i, t_{i+1}]$ for $i = 1,2,3,4$ with $t_1 = 0$ and $t_5 = 1$ and the step size h becomes $h = \frac{1}{N} = \frac{1}{4}$. We can now calculate the entries of the matrix $\mathbf{F}'$, provided we choose a value for θ. If we set $\theta = 1$ (Euler Forward) equation 7.1.4 becomes quite simple:

$$\begin{aligned} Y_{\mathbf{s}_{i,j}} &\doteq \tfrac{I+\theta h\mathbf{A}}{I-(1-\theta)h\mathbf{A}}\mathbf{e}_j = (I + \tfrac{1}{4}\mathbf{A})\mathbf{e}_j \\ &= \begin{bmatrix} 1 & \frac{1}{4} \\ 0 & 1 \end{bmatrix} \mathbf{e}_j. \end{aligned} \tag{39}$$

Substituting the unit vectors $\mathbf{e}_1$ and $\mathbf{e}_2$ for $\mathbf{e}_j$ we find

$$Y_{\mathbf{s}_{i,1}} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \qquad Y_{\mathbf{s}_{i,2}} = \begin{bmatrix} \frac{1}{4} \\ 1 \end{bmatrix}, \tag{40}$$

or put together into one block matrix:

$$\frac{\partial \mathbf{y}_{\mathbf{s}_i}}{\partial \mathbf{s}_i} = \begin{bmatrix} 1 & \frac{1}{4} \\ 0 & 1 \end{bmatrix}. \tag{41}$$

Finally, the lower right block matrix in $\mathbf{F}'$ becomes

$$\mathbf{B_b} \begin{bmatrix} 1 & \frac{1}{4} \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 1 & \frac{1}{4} \end{bmatrix}, \tag{42}$$

so if we put these results together, we find $\mathbf{F}'$:

$$\mathbf{F}' = \mathbf{F}'(\mathbf{S}^0) = \begin{bmatrix} -1 & -\frac{1}{4} & \mathbf{I_2} & \emptyset & \emptyset \\ 0 & -1 & & & \\ \emptyset & & -1 & -\frac{1}{4} & \mathbf{I_2} & \emptyset \\ & & 0 & -1 & & \\ \emptyset & & \emptyset & -1 & -\frac{1}{4} & \mathbf{I_2} \\ & & & 0 & -1 & \\ 1 & 0 & \emptyset & \emptyset & 0 & 0 \\ 0 & 0 & & & 1 & \frac{1}{4} \end{bmatrix}. \tag{43}$$

The method has been tested for $N = 4$ and additionally for $N = 5, 6, 10, 20$. The results show that the solution is $\mathbf{x}(t) = t$ with derivative $\dot{x}(t) = 1$, in accordance with the theory. Below we have included the output for the case $N = 4$. The vector **gs** stands for $\mathbf{F}(\mathbf{S})$ (cf. equation (10)) during step $k = 0, 1$ and $\mathbf{S}^k$ and $\triangle\mathbf{S}^k$ are represented by **s** and **ds** respectively. All vectors are printed as $\mathbf{s} = [[s_{11}, s_{12}][s_{21}, s_{22}][s_{31}, s_{32}][s_{41}, s_{42}]$. We see that $\mathbf{S}^0 = 0$ (our first estimate of the solution) and in the next step has been incremented with **ds** or $\triangle\mathbf{S}^k$. Additionally, $\triangle\mathbf{S}^1$ is now zero, so the algorithm stops. Notice that at the same time $\mathbf{gs} = \mathbf{F}(\mathbf{S}) = 0$ which means that the solution is continuous. S.

The results for $N = 5, 6, 10, 20$ are similar, except of course for more output points. Naturally, the matrices in equations (41) and (42) change into

$$\frac{\partial \mathbf{y}_{\mathbf{s}_i}}{\partial \mathbf{s}_i} = \begin{bmatrix} 1 & \frac{1}{N} \\ 0 & 1 \end{bmatrix}$$

and

$$\mathbf{B_b} \begin{bmatrix} 1 & \frac{1}{N} \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 1 & \frac{1}{N} \end{bmatrix},$$

and so does $\mathbf{F}'$ in equation (43) naturally.

```
****************************************
Msg(Main): Loop 0
Msg(Main): gs =              [[0, 0][0, 0][0, 0][0, -1]]
Msg(Main): s  =              [[0, 0][0, 0][0, 0][0, 0]]
Msg(Main): ds =              [[-0, 1][0.25, 1][0.5, 1][0.75, 1]]
****************************************
Msg(Main): Loop 1
Msg(Main): gs =              [[0, 0][0, 0][0, 0][0, 0]]
Msg(Main): s  =              [[0, 1][0.25, 1][0.5, 1][0.75, 1]]
Msg(Main): ds =              [[-0, -0][-0, -0][-0, -0][-0, -0]]
Solution x at t=0 : 0
Solution x at t=0.25 : 0.25
Solution x at t=0.5 : 0.5
Solution x at t=0.75 : 0.75
```

The NumLab implementation of this procedure has been included in the appendix 11.1. It contains a do-loop in the main routine that in essence represents the Newton iteration equations in formula (11). The loop does not terminate until ds is practically zero, that is, $\mathbf{S}^{k+1}$ and $\mathbf{S}^k$ have become equal and do not change anymore. This means that we have found an $\mathbf{S}$ such that $\mathbf{F}(\mathbf{S}) = 0$.

Example 2: A Linear Inhomogeneous Problem.
As a second example we will discuss a more complex problem defined by:

$$\begin{aligned} &\ddot{x}(t) = x(t) + \dot{x}(t) + 2 - t^2 - 2t, \qquad t \in [a, b] = [0, 1] \\ &x(0) = 0, \quad x(1) = 1 \end{aligned} \tag{44}$$

The difference with the example 1 is the righthand side of the ODE. In matrix representation the righthand side now becomes:

$$\mathbf{A} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$$

and $f(t)$ now is

$$f(t) = 2 - t^2 - 2t.$$

The exact solution (in matrix form) to the problem is

$$\mathbf{x}(t) = \begin{bmatrix} x(t) \\ \dot{x}(t) \end{bmatrix} = \begin{bmatrix} t^2 \\ 2t \end{bmatrix}$$

Instead of Euler Forward we use the Trapezoidal rule to compute $Y_{\mathbf{s}_{i,j}}$, because we expect this method to be exact for quadratic solutions. So $\theta = \frac{1}{2}$. Furthermore, we will divide

the interval $[0,1]$ into $N=4$ subintervals so $h=\frac{1}{4}$. The NumLab output is shown below. The results are printed in the same format as we did for example 1.

We see that the solution $\mathbf{x}(t)$ is indeed quadratic and that it is exact, corresponding with our expectations. This is why we used the Trapezoidal rule of order 2 to calculate the elements of $\mathbf{F}'$ and not Euler Forward or Backward of order 1.

```
****************************************
Msg(Main): Loop 0
Msg(Main): gs =              [[-0.0625, -0.5][-0.0397727, -0.318182]
                              [-0.0125, -0.1][0, -1.01932]]
Msg(Main): s  =              [[0, 0][0, 0][0, 0][0, 0]]
Msg(Main): ds =              [[-0, -0][0.0625, 0.5][0.25, 1][0.5625, 1.5]]
****************************************
Msg(Main): Loop 1
Msg(Main): gs =              [[0, 0][0, 0][0, 0][0, 0]]
Msg(Main): s  =              [[0, 0][0.0625, 0.5][0.25, 1][0.5625, 1.5]]
Msg(Main): ds =              [[-0, -0][-0, -0][-0, -0][-0, -0]]
Solution x at t=0 : 0
Solution x at t=0.25 : 0.0625
Solution x at t=0.5 : 0.25
Solution x at t=0.75 : 0.5625
```

Example 3: A Linear Homogeneous Problem with Dichotomy. The method described earlier in this section would typically fail for problems with increasing and decreasing modes. Let $a>0$ Consider

$$\begin{aligned} &\ddot{x}(t)=a^2x(t) \qquad t\in(0,1)\\ &x(0)=1+e^{-a}=:x_0, \qquad x(1)=1+e^{-a}=:x_1. \end{aligned} \tag{45}$$

The solution is

$$x(t)=e^{-at}+e^{-a(1-t)}. \tag{46}$$

We can rewrite this problem as a first order ODE:

$$\begin{aligned} &\dot{\mathbf{x}}=\mathbf{A}\mathbf{x}=\begin{bmatrix}0 & 1\\ a^2 & 0\end{bmatrix}\mathbf{x}\\ &\mathbf{B}_a\mathbf{x}(0)+\mathbf{B}_b\mathbf{x}(1)-\begin{bmatrix}x_0\\ x_1\end{bmatrix}=0 \end{aligned} \tag{47}$$

.

Clearly the general solution for equation (47) is

$$\mathbf{x}(t) = e^{\mathbf{A}t}\mathbf{x}(0) = e^{\mathbf{A}t}\begin{bmatrix} x(0) \\ \dot{x}(0) \end{bmatrix}. \tag{48}$$

We will now use Fulmer's method to calculate $\mathbf{x}(t)$. First we need to find the characteristic polynomial $c(\lambda)$ of the matrix $\mathbf{A}$

$$c(\lambda) = \det(\mathbf{A} - \lambda I) = \lambda^2 - a^2$$

so we find $\lambda_1 = -a$ and $\lambda_2 = a$ as characteristic roots. This implies that the general solution of

$$c(\mathbf{D})y = (\mathbf{D} - a)(\mathbf{D} + a) = 0$$

can be written as

$$\mathbf{y}(t) = c_1 e^{-at} + c_2 e^{at}$$

for some constants c_1 and c_2.
Fulmer's method now states that we can write $\mathbf{x}(t) = e^{\mathbf{A}t}$ as

$$e^{\mathbf{A}t} = \mathbf{E}_1 e^{-at} + \mathbf{E}_2 e^{at} \tag{49}$$

for some matrices $\mathbf{E}_1, \mathbf{E}_2$. We can find these matrices by evaluating equation (49) at $t = 0$ and doing the same for the (time) derivative of the same equation. Therefore

$$\begin{aligned} e^{\mathbf{A}t}(t=0) = I &= (\mathbf{E}_1 e^{-at} + \mathbf{E}_2 e^{at})(t=0) = \mathbf{E}_1 + \mathbf{E}_2 \\ \mathbf{A}e^{\mathbf{A}t}(t=0) = \mathbf{A} &= (-a\mathbf{E}_1 e^{-at} + a\mathbf{E}_2 e^{at})(t=0) = -a\mathbf{E}_1 + a\mathbf{E}_2. \end{aligned}$$

So we need to solve a linear system of equations

$$\begin{bmatrix} 1 & 1 \\ -a & a \end{bmatrix}\begin{bmatrix} \mathbf{E}_1 \\ \mathbf{E}_2 \end{bmatrix} = \begin{bmatrix} I \\ A \end{bmatrix}.$$

with solutions for $\mathbf{E}_1$ and $\mathbf{E}_2$

$$\begin{aligned} \mathbf{E}_1 = &\ \tfrac{1}{2}\{I - \tfrac{1}{a}\mathbf{A}\} = \tfrac{1}{2}\begin{bmatrix} 1 & -\frac{1}{a} \\ -a & 1 \end{bmatrix} \\ \mathbf{E}_2 = &\ \tfrac{1}{2}\{I + \tfrac{1}{a}\mathbf{A}\} = \tfrac{1}{2}\begin{bmatrix} 1 & +\frac{1}{a} \\ +a & 1 \end{bmatrix}. \end{aligned}$$

If we substitute the matrices $\mathbf{E}_1$ and $\mathbf{E}_2$ in equation (49) we find $e^{\mathbf{A}t}$

$$e^{\mathbf{A}t} = \frac{1}{2}\begin{bmatrix} 1 & -\frac{1}{a} \\ -a & 1 \end{bmatrix} e^{-at} + \frac{1}{2}\begin{bmatrix} 1 & \frac{1}{a} \\ a & 1 \end{bmatrix} e^{at}.$$

It follows from (46) that

$$\begin{array}{rcll} x(0) & = & [e^{-at} + e^{-a(1-t)}]_{t=0} & = & 1 + e^{-a} \\ \dot{x}(0) & = & [-ae^{-at} + ae^{-a(1-t)}]_{t=0} & = & -a + ae^{-a}. \end{array} \tag{50}$$

After substituting (50) in (48) we find

$$\mathbf{x}(t) = \begin{bmatrix} x \\ \dot{x} \end{bmatrix} = \begin{bmatrix} e^{-at} + e^{-a(1-t)} \\ -ae^{-at} + ae^{-a(1-t)} \end{bmatrix}.$$

Another way to find this solution is to diagonalise matrix $\mathbf{A}(t)$ as $\mathbf{A} = \mathbf{Q}\Lambda\mathbf{Q}^{-1}$ with $\Lambda = diag(\lambda_i)$ and Q the matrix of eigenvectors and then compute $e^{\mathbf{A}t}$. It follows from a Taylor expansion that

$$e^{\mathbf{A}t} = e^{\mathbf{Q}\Lambda\mathbf{Q}^{-1}t} = \sum_{n=0}^{\infty} \frac{(\mathbf{Q}\Lambda\mathbf{Q}^{-1}t)^n}{n!} = \mathbf{Q}e^{\Lambda t}\mathbf{Q}^{-1},$$

but naturally this results in the same solution.

7.3 BOUNDPACK: The NumLab QU Integration

This section comments on the integration of the multiple shooting FORTRAN 77 software package (libraries and tests) BOUNDPACK in NumLab. It shows how visual modules are constructed which use the subroutines from BoundPack. The employed techniques were discussed in (subsections of) section 5.

First, we conducted a *data flow analysis* of the BOUNDPACK package (details are in the appendix, see section 11.2). From this analysis we conclude that most tests (for instance test subroutine `MUTSGE()` in `BOUND1.FOR`) can be divided into four parts (see figure 19):

- Input;
- Initialisation (called `premuts`);
- The MUTS routine (called `themuts`);
- Post processing (called `postmuts`).

First, `premuts` sets variables corresponding to the (test) problem we want to solve, specified through the input. Then, `themuts` solves the ODE. Finally, `postmuts` prints the computed solution. The `themuts` is a so-called *driver*: A function which solves the highest level problem.

The (FORTRAN 77) test program on which our modules are based is one long list of

statements, schematically represented on the left hand side in figure 19. By examining the code, we categorised these statements into the four groups mentioned above. Typically, the input is split up in several input blocks (two in our example). The rest of the program was literally cut up in a subroutine `premuts`, `themuts` and `postmuts`. Before data could be read, written and inspected anywhere in the code, as illustrated by the boxes and lines. After the split up, this is no longer possible without the appropriate changes. Data now exists within a subroutine and by default not outside it. So the data connections need to change from direct access in FORTRAN 77 to indirect access in C++. If one subroutine needs data from another subroutine, it will need to address the other subroutine first and then its data, shown as dotted lines in figure 19.
The fact that we split up the test program actually comes back in the test programs we wrote for the different levels **X**, **Y**, **Z**. All of these tests consist of setting up some input, and calls to `premuts`, `themuts`, and `postmuts` on their respective levels.
It might seem that we are striving for a perfect isomorphism between the three levels plus the canvas. This is actually not the case as this is not a feasible goal. It is impossible to map a language through a one to one and onto function onto another language.

7.3.1 Complexity Analysis

During programming, it turned out that we needed more subroutines and (member) functions than we thought initially. Often extension of functionality of one feature led to the necessity of building a number of new auxiliary functions. Current number of subroutines is roughly 750. However, with the possibility of many new things still to be added and improvements of existing routines, this number can still grow significantly.

7.3.2 Modules

Many new modules have been written. They are listed below, categorised in data types, routines and data filters. All are functional on the canvas. First the data types. They exist as single value, in vector -, matrix and tensor form for integers, doubles and floats. Tensoren (a 3-dimensional array) were only needed for doubles and thus they were not implemented for integers and floats. Naturally, this can easily be done.

	Single value	Vectors of:	Matrices of:	Triple arrays of:
Integers	ZInt	ZIVector	ZIMatrix	n/a
Doubles	ZZDouble	ZDVector	ZDMatrix	ZDListMatrix
Floats	ZFloat	ZFVector	ZFMatrix	n/a

The following table shows the routines derived from the MUTS examples. As stated before, the concept was to split the FORTRAN 77 example program into three separate modules.

Example			
Testing purposes	PreworkZ	TheworkZ	PostworkZ
For the MUTSGE example	PremutsGEZ	ThemutsGEZ	PostmutsGEZ
For the MUTSPS example	PremutsPSZ	ThemutsPSZ	PostmutsPSZ
For the MUTSMI example	PremutsMIZ	ThemutsMIZ	PostmutsMIZ
For the MUTSEI example	PremutsEIZ	ThemutsEIZ	PostmutsEIZ

Furthermore, we built the following data filters for communication purposes:

- ZIVectorPrinter
- ZIMatrixPrinter
- ZIVectorPrinter
- ZIMatrixPrinter
- ZIVectorPrinter
- ZIMatrixPrinter

The vector and matrix printers will export their input vector or matrix and write it to a file in MATLAB format if a valid filename is provided. Otherwise they will print the values on the standard output (screen).
In conclusion, we also created a some standard matrices and vectors that were used boundary conditions in tests for BOUNDPACK. These modules allow the user to specify the ODE of interest. Modules for boundary conditions for new ODE's can be constructed relatively easily. Then, simply connect another boundary matrix and or vector and the boundary conditions of the problem change and we have a new ODE.

- ZDMatrixIdentity,
- ZDVectorBCV_GE
- ZDMatrixBMA_PS
- ZDMatrixBMB_PS
- ZDVectorBCV_PS
- ZDVectorBCV_MI
- ZDMatrixBMA_EI
- ZDMatrixBMB_EI

Boundary conditions of an ODE can be written as

$$\mathbf{B}_a\mathbf{x}(a) + \mathbf{B}_b\mathbf{x}(b) = \mathbf{bcv}.$$

As an example, module ZDVectorBCV_GE was designed to represent the boundary vector in the MUTSGE example in [10] and module ZDMatrixBMA_EI represents the left boundary matrix in the MUTSEI example, see also [10]. However, usage in this way is not compulsory. In the end, they remain matrices and vectors and can be used whenever a matrix or vector is required. It just may be the case that by 'mixing' modules, an ill-posed problem is created.

7.3.3 BOUNDPACK Networks

In this section we will show how the available BOUNDPACK subroutines can be used on the canvas. We will do this by two example problems.

Network example 1:
Let a, b be real numbers and $a < b$. The first example we will examine is based on the following ODE:

$$\frac{d}{dt}\mathbf{x}(t) = \mathbf{A}(t)\mathbf{x}(t) + \mathbf{f}(t) \quad a \leqq t \leqq b \tag{51}$$

with **general** boundary conditions:

$$\mathbf{B}_a\mathbf{x}(a) + \mathbf{B}_b\mathbf{x}(b) = \beta. \tag{52}$$

In this particular example we set $a = 0$, $b = 6$ and define $\mathbf{A}(t)$, $\mathbf{f}(t)$ and β by:

$$\mathbf{A}(t) = \begin{bmatrix} 1 - 2\cos(2t) & 0 & 1 + 2\sin(2t)0 & 1 \\ 0 & 2 & 0 & \\ -1 - 2\sin(2t) & 0 & 1 + 2\cos(2t) & \end{bmatrix}, \tag{53}$$

$$\mathbf{f}(t) = \begin{bmatrix} (-1 + 2\cos(2t) - 2\sin(2t))e^t \\ -e^t \\ (1 - 2\cos(2t) - 2\sin(2t))e^t \end{bmatrix}, \tag{54}$$

and

$$\beta = \begin{bmatrix} 1 + e^6 \\ 1 + e^6 \\ 1 + e^6 \end{bmatrix}. \tag{55}$$

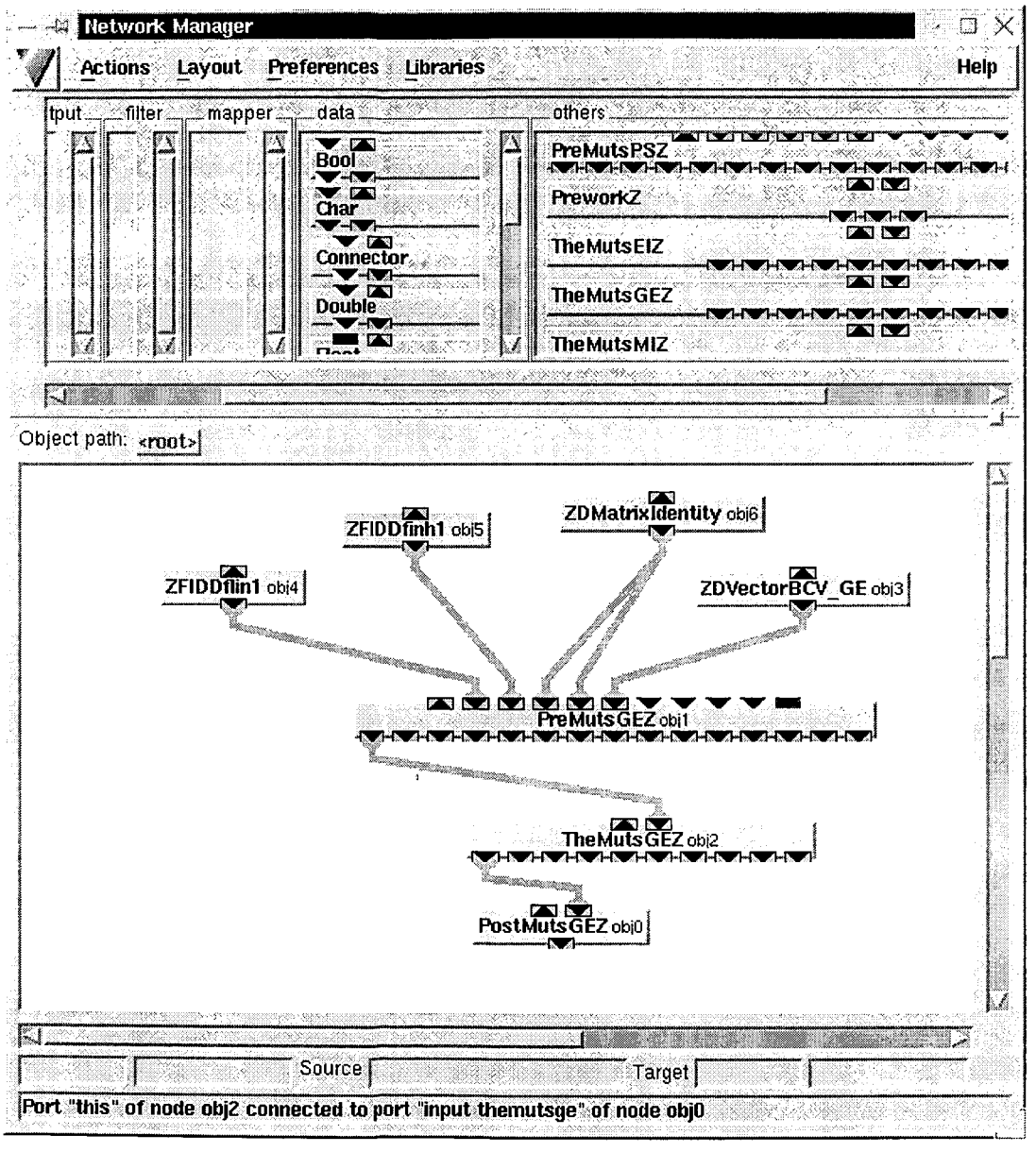


Figure 30: NumLab network with the MUTSGE example

and the boundary matrices $\mathbf{B}_a$ and $\mathbf{B}_b$ are set to the identity matrix

The first important matter is to decide what *kind* of ODE we are dealing with. More specific, what kind of boundary conditions are specified. The ODE above has general boundary conditions, so we use the routines from BOUNDPACK that were designed to solve these kind of problems. In this example, we need the MUTSGE routines where "GE" indicates the **general** nature of the boundary conditions.

After splitting up the original FORTRAN 77 source code into three separate blocks and creating modules from these blocks, we can build a network with them and construct the example problem. We use a module PreMutsGEZ to set up default variables. Furthermore we can choose boundary conditions and plug them into the PreMutsGEZ module so they will be used during computations. We connect the PreMutsGEZ module to the TheMutsGEZ module, where a solution to the problem defined by PreMutsGEZ is calculated. Finally, the module PostMutsGEZ will print the results on screen.

In figure 30 we can identify the following modules:

- ZFIDDflin1, representing the linear part of the ODE: matrix $\mathbf{A}(t)$ in (53);
- ZFIDDfinh1, the inhomogeneous part of the ODE: vector $\mathbf{f}(t)$ in (54);
- ZDMatrixIdentity, a 3×3 identity matrix: our choice for $\mathbf{B}_a$ and $\mathbf{B}_b$;
- ZDVectorBCV_GE, representing β from (55), the righthand side of the boundary conditions;
- PreMutsGEZ, the module that sets up several (default) variables and defines the ODE to be solved;
- TheMutsGEZ, the module that calls back to the original BOUNDPACK source code and computes the solution to the problem defined by PreMutsGEZ;
- PostMutsGEZ, serves to export the results to the screen;

So the "main" part of the network is the three MUTSGE modules. If we would like to compute the solution of another problem, we can do so by plugging in other modules into the PreMutsGEZ. Note that we use the identity matrix twice, as input for $\mathbf{B}_a$ as well as for $\mathbf{B}_b$. The only thing that defines the ODE and that is missing on the canvas as a module, is the interval $[a, b]$. We opted to leave them away from the canvas. Nevertheless, the values a and b can be specified by opening the interactor of the PreMutsGEZ module. So all data that defines the ODE can be changed.

The size or the actual width of the modules is quite large. For example, the PreMutsGEZ module has eleven data ports on top, and fifteen data ports at the bottom. Especially the ports at the bottom have been created to export internal data (owned data) through these data ports to the outside for inspection by the user.

The same is true about the nine unused data ports at the bottom of the TheMutsGEZ module. They make the internal data accessible for us. But, as mentioned earlier, the program does not need these ports because all relevant data will flow through the single connection from TheMutsGEZ to PostMutsGEZ. For instance, part of this data flow is a variable $\mathbf{x}$ that holds the final solution, per component and per output point ($\mathbf{x}$ is a matrix in fact). Since $\mathbf{x}$ is owned by TheMutsGEZ, it is an internal variable with an output port at the bottom of the module.

Network Example 2:
As a second example we have a network for the following

$$\frac{d}{dt}\mathbf{x}(t) = \mathbf{A}(t)\mathbf{x}(t) + \mathbf{f}(t) \quad a \leqq t \leq b, \tag{56}$$

with **integral** boundary conditions:

$$\int_a^b \mathbf{M}(t)\mathbf{x}(t)dt = \beta. \tag{57}$$

In 56 $\mathbf{A}(t)$, $\mathbf{f}(t)$ and β are defined by:

$$\mathbf{A}(t) = \begin{bmatrix} 1 & 0 \\ 0 & -2t \end{bmatrix}, \tag{58}$$

$$\mathbf{f}(t) = \begin{bmatrix} -e^{-t} \\ (2t-1)e^{-t} \end{bmatrix}, \tag{59}$$

$$\beta = \begin{bmatrix} 2\sinh(4) \\ 2\sinh(4) \end{bmatrix}. \tag{60}$$

Matrix $\mathbf{M}(t) = \mathbf{I}_2$, the 2×2-identity matrix.
Obviously, this problem has integral boundary conditions. BOUNDPACK has a MUTSMI routine (MI indicating *Multipoint Integral* that solves ODE's with this kind of boundary conditions. Hence, we use modules that were derived from this routine in order to build

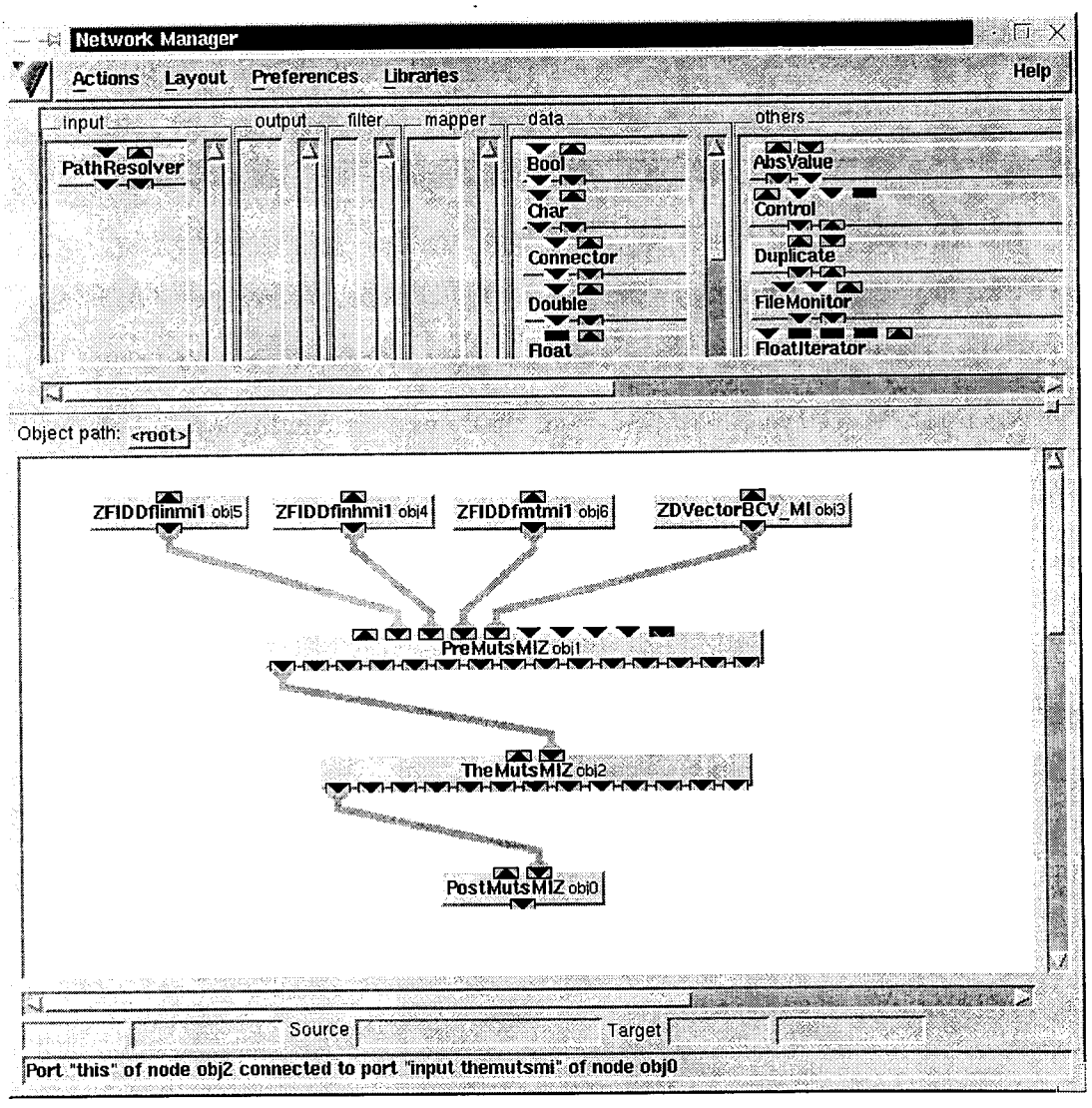


Figure 31: NumLab network with the MUTSMI example

a network to solve the ODE.

In figure 31 the BOUNDPACK network is quite similar to the one from the example 1. We can distinguish the following modules:

- PreMutsMIZ, defining the ODE by means of its input;
- TheMutsMIZ, computing the solution for the problem defined in PreMutsMIZ;
- PostMutsMIZ, output (i.e. printing) the solution;
- ZFIDDflinmi1, the linear part of the ODE (i.e. $\mathbf{A}(t)$ from equation (58));
- ZFIDDfinhmi1, the non-linear part of the ODE (i.e. $f(t)$ in equation (59));
- ZFIDDfmtmi1, the matrix $\mathbf{M}(t)$ from the integral boundary conditions;
- ZDVectorBCV_GE, vector β, also from the boundary conditions and as defined in (60).

Just as in example 1, the begin and end point of the time interval $[a, b]$ can be specified by opening the interactor of the PreMutsMIZ module.

Compared to figure 30 there are nevertheless some differences.

- Different modules that callback to different BOUNDPACK routines;
- Different (nature of) boundary conditions.

Despite these fundamental differences, both networks described in this section still function identically: The ODE is defined at the top and solved by a callback to BOUNDPACK routines.

8 Turing Completeness

8.1 Introduction

This section demonstrates that, with the addition of certain basic modules, the NumLab visual programming language is Turing complete. To this end, first, section 8.2 introduces the concepts of alphabet and language. Next, section 8.3 introduces Turing machines. Then section 8.4 presents the NumLab module design for primitive recursive functions. The last section 8.5 presents the NumLab module design for μ-recursive functions. Able to imitate μ-recursive functions, NumLab is Turing complete.

8.2 Alphabets and Language

The definitions of string, alphabet and language are taken from [7], pages 29-31:

Definition 6

- *An alphabet is finite set of symbols, such as the Roman alphabet* {a, b, c, ... , z} *or the binary alphabet* {0, 1}.
- *A string over an alphabet* Σ *is a finite sequence of symbols from alphabet* Σ.
- *The set of all strings - including the empty string - over an alphabet* Σ *is denoted by* Σ^*.

These definitions permit the definition of a language:

Definition 7 *A language is set of strings over an alphabet* Σ.

Note that in particular a language over σ is a subset of Σ^*. Turing machines are defined using languages.

8.3 The Turing Machine

We start by introducing the concept of a *Turing machine*. We describe a Turing machine keeping figure 32 in mind.

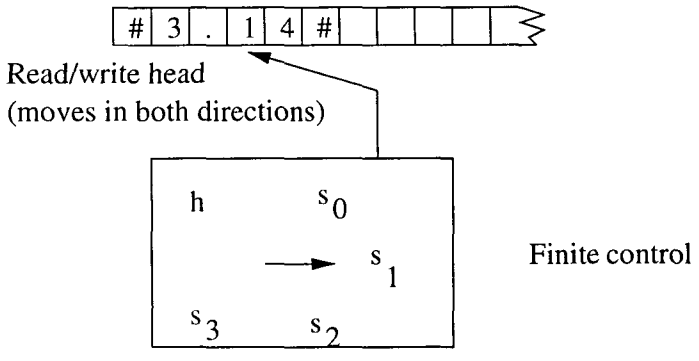


Figure 32: A Turing machine

A Turing machine consists of a *tape* and a *finite-state machine*, called *control unit*. The control unit disposes over a head to read from and/or write to the tape. In each step, the control unit reads the tape and then performs the following tasks:

- Put the control unit in a new state;
- Either:
  - Write a symbol on the current square on the tape *or*
  - Move the read/write head one position to the left ('L') or to the right ('R').

The tape has a left end, but is unbounded on the right side. However, in a finite numbers of steps, the machine can only visit a finite number of squares on the tape. (In case the machine tries to move its head to the left off the end of the tape, it ceases to operate).

Initially, the tape contains only symbols at the left end. The rest of the tape consists of blank symbols. The machine is free to alter its input or write on the blank end of the tape. The message (data) left at the end of the computation is called the answer. The end of computations is reached when the control unit reaches the halt state. The *blank symbol* will be denoted by #.

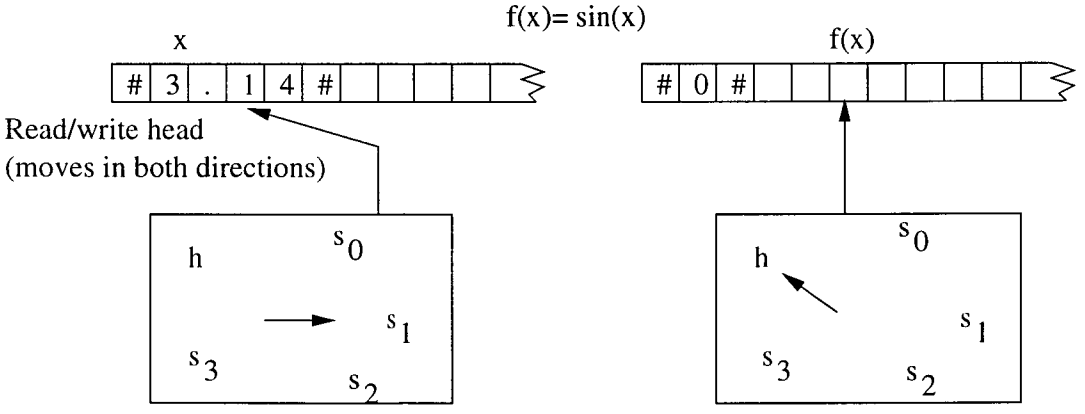


Figure 33: A Turing machine which imitates a the sine-function

As an example, consider figure 33. The Turing machine shown computes $f(x) = \sin(x)$. Here $x = 3.14$ is the input the tape. So, to start with, the tape contains the symbols 3, ., 1 and 4, and the control unit is in state s_1. At the end of the computations, the control unit is in the halt state h and has left behind the answer 0 on the tape.

The following definition of a Turing machine comes from [7]:

Definition 8 *A Turing machine is a quadruple (K, Σ, δ, s) where*

- *K is a finite set of states, not containing the halt state denoted by h;*

- Σ *is an alphabet, containing the blank symbol* $\#$*, but not containing the the symbols* L *and* R*;*
- $s \in K$ *is the initial state;*
- δ *is a function from* $K \times \Sigma$ *to* $(K \cup h) \times (\Sigma \cup \{L, R\})$.

With the use of this definition, we can define when a (mathematical) function is Turing computable – required to prove the Turing completeness of NumLab:

Definition 9 *Turing computable functions:*
Let Σ_0 *and* Σ_1 *be alphabets not containing the blank symbol* $\#$*. Let* f *be a a function from* Σ_0^* *to* Σ_1^**. A Turing machine* $M = (K, \Sigma, \delta, s)$ *is said to compute* f *if* $\Sigma_0,\ \Sigma_1 \subseteq \Sigma$ *and for any* $w \in \Sigma_0^*$*, if* $f(w) = u$ *then*

$$(s, \#w\underline{\#} \vdash_M^* (h, \#u\underline{\#}). \tag{61}$$

If such a Turing machine M *exists, then* f *is said to be a Turing computable function.*

Because a Turing machine can carry out any computation that can be carried out by any similar type of automata, and because these automata seem to capture the essential features of real computing machines, we take the Turing machine to be a precise formal equivalent of the intuitive notion of "algorithm". Following *Church's Thesis* or *Church-Turing's Thesis*, nothing will be considered an algorithm if it cannot be rendered as a Turing machine. It is a thesis, not a theorem, because it is not a mathematical result: It simply asserts that a certain informal concept corresponds to a certain mathematical object. It is theoretically possible, however, that Church's Thesis could be overthrown at some future date, if someone were to propose an alternative model of computation that was publicly acceptable as fulfilling the requirement of "finite labour at each step" and yet was provably capable of carrying out computations that cannot be carried out by any Turing machine. No one considers this likely.

A language is called *Turing complete*, if it can generate all Turing computable functions (see also [7]). In this small section, it is shown that NumLab is Turing complete, if a few fundamental modules are added.

8.4 Primitive Recursive Functions

The primitive recursive functions are defined by three types of initial functions and two combining rules. These can all be presented in a straight-forward manner.

Definition 10 *The initial functions are the following three functions:*

- *The **0**-place function ζ is the function from $\mathbf{N}^0$ to $\mathbf{N}$ such that*

$$\zeta() = 0. \tag{62}$$

- *Let $k \geq 1$ and let $1 \leq i \leq k$. Then the i-th k-place projection function π_i^k is the function from $\mathbf{N}^k$ to $\mathbf{N}$ such that*

$$\pi_i^k(n_1, ..., n_k) = n_i, \qquad \text{for any } \; n_1, ...n_k \in \mathbf{N}. \tag{63}$$

Remark 1 *Point of notation: Hereafter we write $\bar{n}$ for the k-tuple $(n_1, ..., n_k)$. Thus the above statement would be rewritten*

$$\pi_i^k(\bar{n}) = n_i, \qquad \text{for any } \; \bar{n} \in \mathbf{N}^k. \tag{64}$$

- *The successor function σ is the function from $\mathbf{N}$ to $\mathbf{N}$ such that*

$$\sigma(n) = n + 1, \qquad \text{for any } \; n \in \mathbf{N}. \tag{65}$$

We introduce the related three NumLab basic modules, and an additional one. These modules are:

- A zero module;
- An increment module;
- A decrement module;
- A decision or switch module,

and shown in figure 34.

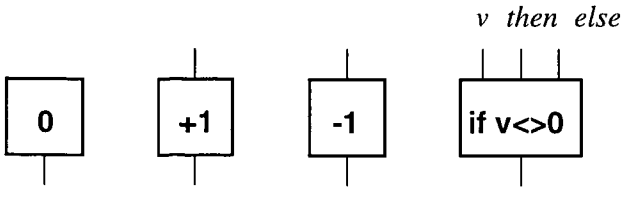


Figure 34: Basic elements

Obviously, the zero module itself is already the 0-place function ζ. Figure 35 shows the

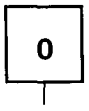


Figure 35: Initial function zero module

zero element from the initial functions

Furthermore the increment module acts as the successor function σ. Using these two modules, we already dispose over the natural numbers, by consecutive incrementing 0.

The k-place projection is shown in figure 36. The figure shows on the left side a k-place projection function π^k. In this particular case, three input values n_1, n_2 and n_3 are entered into the module. The 2-projection selects the second component out of three input values.

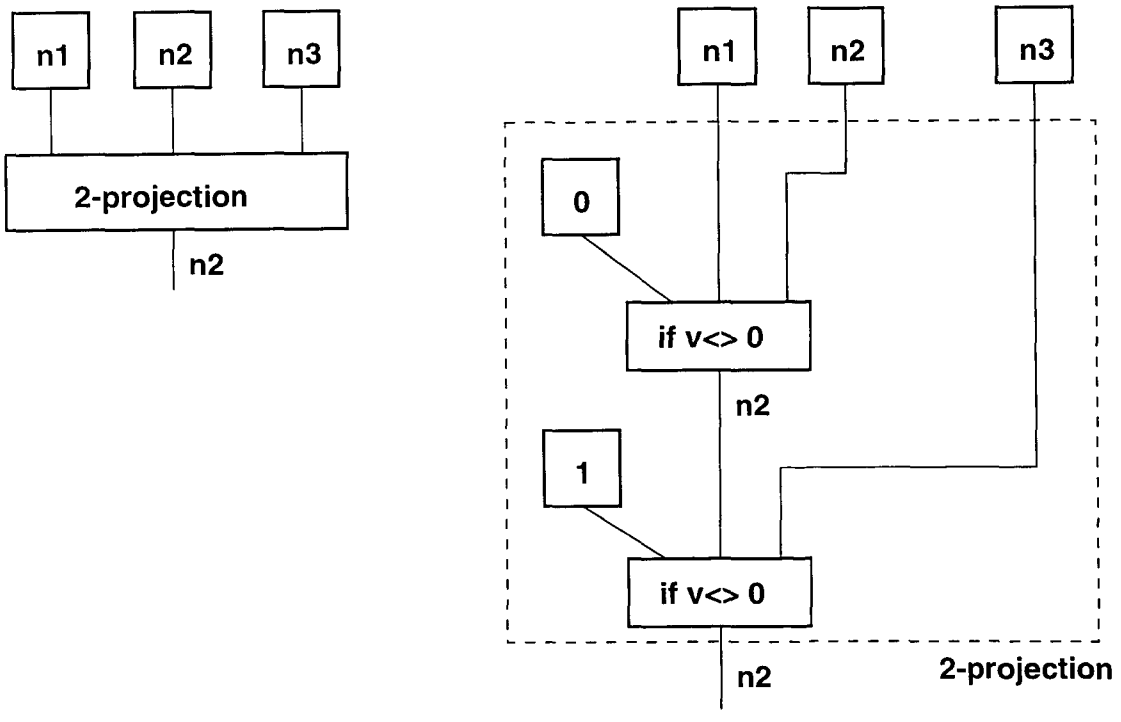


Figure 36: The 2-projection module

On the righthand side, it shows how we built the k-place projection function π^k from the axioms before. The module first selects n_2 out of n_1 and n_2, by setting the right value

0 on the decision module. Next, it selects n_2 from n_2 and n_3 in the same manner but now by setting a 1 on the second decision module. Finally, value n_2 is exporting to the outside.

As an example, we provide a possible NumLab implementation of the decision module, using pseudo code:

```
class decision: public module
{
 void set(double *v)    { this->v = v; }
   ...
 double update()
 {
  return (v->update()) ? then->update() : else->update();
 }
 private:
  module *v;
   ...
}
```

We now proceed with the definitions of *composition* and *Primitive Recursions*:

Definition 11

- *Let $l > 0$ and $k \geq 0$, let g be an l-place function, and let $h_1, ..., h_l$ be k-place functions. Let f be the k-place function such that, for every $\bar{n} \in \mathbf{N}^k$,*

$$f(\bar{n}) = g(h_1(\bar{n}), ..., h_l(\bar{n})). \tag{66}$$

  Then f is said to be obtained from g, h_1, ...h_l by composition.

- *Let $k \geq 0$, let g be an k-place function, and let h be a $(k+2)$-place function. Let f be the $(k+1)$-place function such that for every $\bar{n} \in \mathbf{N}^k$,*

$$f(\bar{n}, 0) = g(\bar{n}) \tag{67}$$

  and for every $\bar{n} \in \mathbf{N}^k$ and $m \in \mathbf{N}$

$$f(\bar{n}, m+1) = h(\bar{n}, m, f(\bar{n}, m)) \tag{68}$$

  Then f is said to be obtained from g and h by primitive recursion.

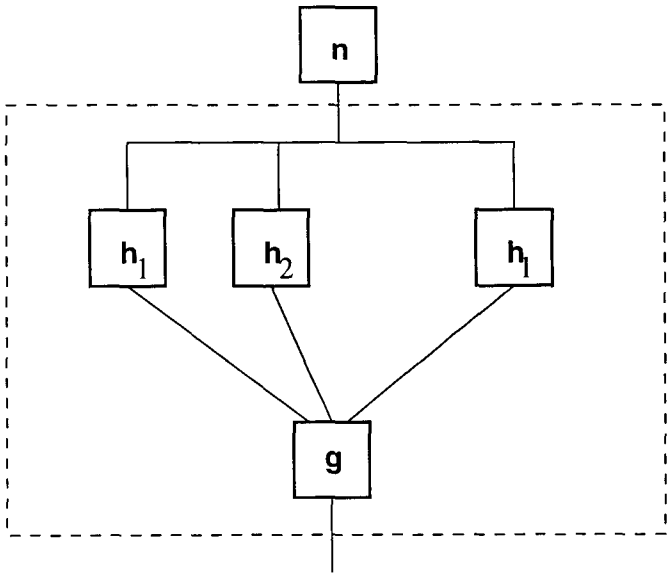


Figure 37: The composition module

Definition 12 *A function is said to be a primitive recursive function if it is an initial function or can be generated from the initial functions by some sequence of operations of composition and primitive recursion. More succinctly, the primitive recursive functions are the smallest class of functions containing the initial function and closed under composition and primitive recursion.*

Because all primitive recursive functions terminate, the set of all primitive recursive functions can not represent the set of all Turing computable functions. Therefore, in order to obtain all computable functions, some extension must be made to the methods used thus far for defining functions.

8.5 μ-Recursive Functions

This section introduces μ-recursive functions and presents a visual module design for their NumLab implementation. Because the functions in the set of μ-recursive functions can *imitate* all Turing machines, NumLab is Turing complete with the addition of this type of module. First, we must define the concept of unbounded minimalisation:

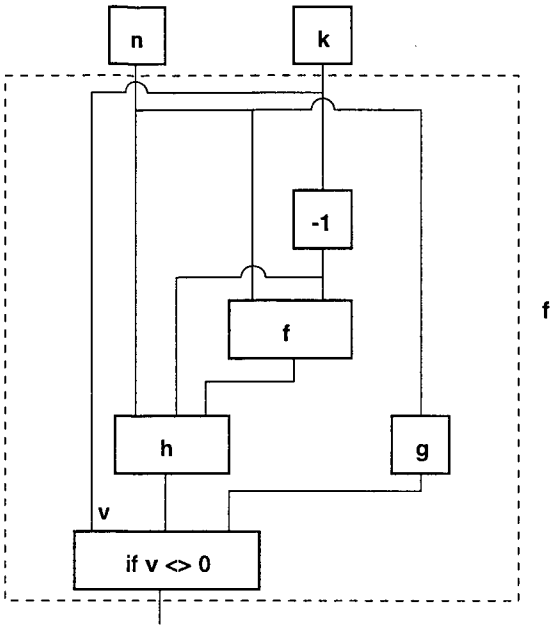


Figure 38: The NumLab design of a primitive-recursive module

Definition 13 *Let $k \geq 0$ and let g be a $(k+1)$-place functions. Then the unbounded minimalisation of g is that k-place function f such that, for any $\bar{n} \in N^k$*

$$f(\bar{n}) = \begin{cases} \textit{the least } m \textit{ such that } g(\bar{n}, m) = 0 \textit{ if such } m \textit{ exists;} \\ 0 \textit{ otherwise.} \end{cases} \tag{69}$$

The second clause guarantees that f is everywhere defined, regardless of what g is. We write

$$f(\bar{n}) = \mu m[g(\bar{n}, m) = 0] \tag{70}$$

and say that f is obtained from g by unbounded minimalisation.

In general, the unbounded minimalisation of a primitive recursive function need to be primitive recursive, or indeed computable in any intuitive sense. The reason, as we shall show later, is that there is no general method of telling whether an m of the required type exists. However, if g has the property that such an m exists for every $\bar{n}$, then f is computable if g is computable: Given $\bar{n}$, we simply need to evaluate all of $g(\bar{n}, 0), g(\bar{n}, 1), ...$ until we find m such that $g(\bar{n}, m) = 0$. However, in this case f need not, in general, be primitive recursive.

These ideas leads to the definition of regular functions:

Definition 14 *A $(k+1)$-place function g is called a* regular function *if and only if, for every $\bar{n} \in \mathbf{N}^k$, there is an m such that $g(\bar{n}, m) = 0$. A function is μ-recursive if and only if it can be obtained from the initial functions ζ, π_i^k, and σ by the following operations:*

- *composition*
- *primitive recursion*
- *application of unbounded minimalisation to regular functions.*

With this definition, each primitive recursive function is also μ-recursive.

Figure 39 shows the NumLab module design for a μ-recursive function.

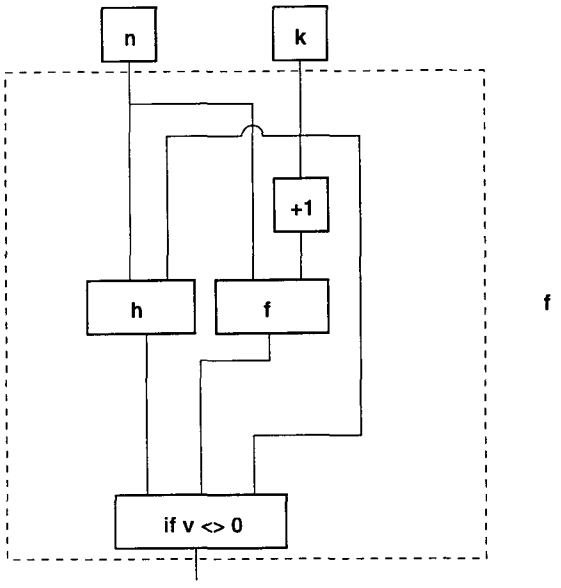


Figure 39: The NumLab design of a μ-recursive module

In order to finish this section, we refer to [7], which shows that the set of all μ-recursive functions can imititate all Turing machines (so all computable functions). Thus, because NumLab can generate modules for all μ-recursive functions, NumLab is Turing complete.

9 Future Research

Future research could address some of the visual programming issues not (completely) solved in this thesis:

- Developing still more mathematical, rigorous definitions of toolbox and module;
- Defining more precisely what the network manager does in the case of controlled loops. Thesis [15] only provides an algorithm;
- Extending all definitions to the case of parallel execution of multiple modules in a single network;
- Developing a formal specification for boundary value problems and finite element solvers for data exchange with *OpenMath/MathML/NumLab*;
- Applying the proposed integration process of *BoundPack* routines, in order to integrate (parts of) *LAPACK* and, if possible, *SEPRAN*

Improving the graphical user interface and the network manager :
We encountered several network editor bugs during the implementation phase. For instance, a load/save bug problem. When saving a fully connected network with a loop, such as the Newton network that calculates the square root of a real number we do not encounter any problems. Problems occur when we try to load it again on the canvas and the system freezes. A temporary solution is to save the network after disconnecting the loop. Then the network can be loaded again and restarted after connecting the missing connection.

Acknowledgements
The author wishes to thank Dr. J. Maubach and Ir. S. Houben for several valuable dicussions and contributions.

10 Conclusions

The following conclusions can be reached:

- Efficient and user-friendly reusing of existing C-linkable (FORTRAN 77, Pascal, C) software is possible. The proposed and implemented solution is described in section 5.
- The above solution allows toolboxes of modules to be developed independent from the NumLab environment and other toolboxes. However, modules from different toolboxes can be combined in one network.
- Efficient visual interactive loop control is possible for NumLab, a solution and implementation is presented in section 6.
- The visual programming environment NumLab is Turing complete (section 8), if a few fundamental blocks are added, and derived blocks are are modelled in a specific manner.
- The existing multiple shooting library BOUNDPACK can only be integrated in a visual programming environment at a very coarse level. The reason is turned out to be the lack of documentation at finer levels.
- The NumLab datum/operator representations (figure 11) significantly simplify the construction of visual interactive modules for scientific computing algorithms.

11 Appendix

11.1 A NumLab DE Implementation

Below is the source code of the program that we used to solve the homogenous problem (38) as stated in section 7.2. The code intends to show that a NumLab-script implementation of the DE multiple shooting method is rather straight forward, making use of NumLab's `Matrix` and `Vector` data types.

```
#include "\numlab.h"
#include "LA/LE/Matrix.hd"
#include "LA/LE/Vector.hd"

#include <iostream.h>

// Block matrices and vectors contain LEMatrices, respectively LEVectors.
LEMatrix Idn(2,2);
LEMatrix Ba(2,2);
LEMatrix Bb(2,2);
LEVector b(2);
LEMatrix C(2,2);

// The ODE is y' = F(t, y). In the example below, F(t, y) = Ay+f(t).
// The matrix A equals {{0,1}{c,d}}

LEVector f(const Real t)          // f(t), source term
{
 LEVector z; z(0) = 0; z(1) = 0;  // f(t)=0 in this example
 return z;
}

Real c(const Real t)              // function c(t)
{
 return 0;                        // c(t)=0 in this example
}

Real d(const Real t)              // function d(t)
{
 return 0;                        // c(t)=0 in this example
}
LEMatrix A(const Real t)      // Matrix A in F(t, y) = Ay+f(t)
{
 LEMatrix L;
 L(0, 0)  = 0;
```

```
 L(0, 1)  = 1;
 L(1, 0)  = c(t);
 L(1, 1)  = d(t);
 return L;
}
LEVector F(const Real t, const LEVector &x) //  z := F(x) = Ax + f(t)
{
 LEVector z(x);
 z = A(t)*x;                    // z = Ax
 z = f(t);                      // z = Ax + f(t)
 return z;
}

Vector G(const Real theta, const Real h, const Real t, const Vector &s)
//  z := G(s). We are looking for an s such that G(s)=0.
{
 Vector z,y;
 Real N = s.size()/2;
 for (Integer i = 0; i < N ; i++)
 {
  Real t_i = t + i*h;
  //Theta method to determine y_i(t_{i+1}).
  LEMatrix Denom = Idn/(Idn-(1-theta)*h*A(t_i+h));
  y.block(i) = (Denom*(Idn+theta*h*A(t_i)))*s.block(i) +
               Denom*(theta*h*f(t_i)+(1-theta)*h*f(t_i+h));
  if(i!=N-1) z.block(i) = s.block(i+1) - y.block(i);
 }
 z.block(N - 1) = Ba * s.block(0) + Bb * y.block(N - 1) - b;
 return z;
}


Matrix  dG(const Real theta, const Real h, const Real h_d,
           const Real t, const Vector &s)
// z := G(s)
{
 Integer p = b.size();
 Integer N = s.size()/2;
 Matrix DG;
 LEMatrix MT="[0,0;0,0]";
 for (Integer i = 0; i < N; i++)
  for (Integer j = 0; j < N; j++)
   DG.block(i, j) = MT;
```

```
 Vector y;
 Vector yp;

 for (Integer i = 0; i < N ; i++)
 {
  Real t_i = t + i*h;
  // Using Theta method to determine y_i(t_{i+1}).
  LEMatrix Denom(2,2);// = Idn-(1-theta)*h*A(t_i+h);
  Denom =  Idn/(Idn-(1-theta)*h*A(t_i+h));
  y.block(i) = (Denom*(Idn+theta*h*A(t_i)))*s.block(i) +
               Denom*(theta*h*f(t_i)+(1-theta)*h*f(t_i+h));
  for (Integer j = 0; j < p; j++)
  {
   LEVector sp; LEVector yp;
   sp = s.block(i); sp(j) += h_d;
   // Theta method to determine z_i(t_{i+1}).
   yp = (Denom*(Idn+theta*h*A(t_i)))*sp +
         Denom*(theta*h*f(t_i)+(1-theta)*h*f(t_i+h));
   DG.block(i, i)(":", j) = -1*(yp - y.block(i)) / h_d;
  }
  if (i!=N-1) DG.block(i , i + 1) = Idn;
 }
 DG.block(N - 1, 0) = Ba;
 DG.block(N - 1, N - 1) = -1*Bb * DG.block(N - 1, N - 1);
 return DG;
}

int main()
{

 // Small number for numerical differentation
 // f'(x) = [f(x+h_d) - f(x)]/ h_d
 Real    h_d = 1.e-7;

 mx_print(10000);
 Idn      = ID(2, 2);
 Ba(0, 0) = 1;
 Bb(1, 0) = 1;
 b(0)     = 0;
 b(1)     = 1;

 Integer N = 4;       // The amount of intervals.
 cout << "Enter the amount of subintervals N: " << endl;
```

```
cin >> N;
Real h = 1.0 / N;    // The interval width.
Integer n = 0;       // Loop counter initialisation

// theta=0 gives Euler Backward,      order 1
// theta=.5 gives Trapezoidal ruled, order 2
// theta=1 gives Euler Forward,       order 1
Real theta=.5;

// Start algorithm.
Real  t_0 = 0; // Start point for integration
Vector  s;
for (Integer i = 0; i < N; i++) s.block(i) = "[0, 0]";
Vector ds;
// Currently: dG and G depend on a single time t_0 and
// a global fixed h, N to determine t_1.
do
{
 cout << "****************************************" << endl;
 cout << "Msg(Main): Loop " << n++ << endl;

 // dgs = dG/ds(S^k)
 Matrix dgs = dG(theta, h, h_d, t_0, s);

 // gs = G(S^k)
 Vector  gs = truncate(G(theta, h, t_0, s),1.e-8);

 cout << "Msg(Main): gs =              " << gs << endl;

 // solve with Householder -- full matrix based.
 ds = -1*truncate(gs / dgs, 1.e-5);
 cout << "Msg(Main): s  =              " << s << endl;
 s += ds;
 cout << "Msg(Main): ds =              " << ds << endl;
}
while (sqrt(ds * ds) > sqrt(machine_epsilon));
Vector t,S;
for (Integer i = 0; i < N; i++)
{
 t(i) = i * h;
 S(i) = s(2*i);
 cout << "Solution x at t="<< t(i) <<" : " << S(i) <<endl;
}
```

```
 plot(t, S);
 wait_for_enter();
 return Ok;
}
```

11.2 BOUNDPACK Functions IO-Analysis

The construction of modules for BOUNDPACK functions required an I/O analysis. This section presents the results of this analysis. Horizontally are listed the main subroutines from BOUNDPACK , denoted by a two characters. For instance, 'GE' refers to the MUTSGE function for solving ODE's with *general* boundary conditions, and 'PS' to the function MUTSPS for problems concerning boundary conditions that are partially separated. More information on these abbreviations can be found in [9] and [10]. Vertically we find all variables and functions used in BOUNDPACK function headers. We need to know whether these variables are used as input, output or both. For each of these variables and functions, we have marked the appearance in the corresponding BOUNDPACK function. Futhermore it is indicated what the initial and final value of such variables is, if applicable. As an example, we can find from this table that the external function `FLIN(N,T,FL(N,N))` appears in the first eight BOUNDPACK functions. In contrast `L` appears only in MUTSPA and SPLS3, indicates the number of parameters and is unchanged on exit. So `L` is an input variable.

ITEM	GE	PS	SE	IN	MP	MI	PA	DD	EI	S1	S2	S3	on entry(input)	on exit(output)
FLIN(N,T,FL(N,N))	y	y	y	y	y	y	y	y					external	
FLINE(N,T,													external	
FL(N,N),ALAM)									y				external	
FINH(N,T,FR(N)))	y	y	y	y	y	y	y	y					external	
FMT(N,T,FM)						y							external	bound. matrix M(t)
FCT(N,L,T,FC(N,L))							y						external	evaluation of C(t)
N	y	y	y	y	y	y	y	y	y	y	y	y	order of the system	unchanged
L							y					y	# parameters	unchanged
NPL							y					y	dimension MA,MB,BCV	unchanged
IHOM	y	y	y	y	y	y	y	y		y	y	y	0/1	unchanged
KSP		y	y										0_iksp$_i$=N	actual ksp of the BC
A	y	y	y			y	y		y				boundary point	unchanged
B	y	y	y			y	y		y				boundary point	unchanged
A				y									particular interval	unchanged
B				y									particular interval	unchanged
A(N,N,NRI)										y	y		recursion matrix A_i	unchanged
B(N,N,NRI)										y	y		recursion matrix B_i	possibly changed
G(N,NRI)										y	y		RHS vector G_i	possibly changed
A(N,N,NREC)												y	recursion matrix A_i	unchanged
B(N,N,NREC)												y	recursion matrix B_i	unchanged
C(N,L,NREC)												y	matrix C_i	unchanged
G(N,NREC)												y	RHS vector G_i	unchanged
C				y									gamma_max or gamma	unchanged
BMA				y									BC matrix Ma	unchanged
BMINF				y									BC matrix Minf	unchanged
TBP					y								switching points	unchanged
NBP					y								# of switching points	unchanged
TSP								y					switching points	unchanged
NSP								y					# switching points	unchanged
EIG(2)									y				endpoints	endpoints of interval lambda was found in

ITEM	GE	PS	SE	IN	MP	MI	PA	DD	EI	S1	S2	S3	on entry(input)	on exit(output)
BCM					y								BC matrix M_j	unchanged
MA(N,N)	y	y	y						y				BC matrix Ma	unchanged
MB(N,N)	y	y	-						y				BC matrix Mb	unchanged
MA(NPL,NPL)							y						BC matrix [Ma—Pa]	unchanged
MB(NPL,NPL)							y						BC matrix [Mb—Pb]	unchanged
NREC										y		y	total # of x_i	unchanged
M1										y		y	M1 from BC	unchanged
MN										y		y	MN from BC	unchanged
MZ												y	MZ from BC	unchanged
MI											y		matrix M_j of the BC	unchanged
KMI											y		dimension IJ,MI,NREC,KP	unchanged
NREC(KMI)											y		total # of x_i	unchanged
IJ(KMI)											y		subindex i_j of x_(i_j)	unchanged
BCV(N)	y	y	y	y	y	y				y	y	y	BC vector	unchanged
BCV(N)								y					BC vector	overwritten
BCV(NPL)							y						BC vector [bx—bz]$\hat{T}$	unchanged
BCM(N,N,k)								y					BC matrix M_j	overwritten
ZM(N,N,k)								y					side cond matrix M_j-	overwritten
ZP(N,N,k)								y					side cond matrix M_j+	overwritten
BI(N,k)								y					side cond vector B_j	overwritten
ALI	y	y	y	y	y	y	y	y	y				allowed incr.factor	actual increm factor
ER(5)	y	y	y	y	y	y	y	y	y				ER(1) rel tol	possibly modified
													ER(2) abstol	unchanged
													ER(3) epsilon	unchanged
													ER(4) –	condition number
													ER(5) –	amplification factor
NRTI	y	y	y	y		y	y		y				>= 0	total # of output pts
NRTI(m¿NBP)					y								>= 0	total # of output pts

ITEM	GE	PS	SE	IN	MP	MI	PA	DD	EI	S1	S2	S3	on entry(input)	on exit(output)
NRTI(k¿NBP)								y					>= 0	total # of output pts on each interval
NRI										y	y		dim A,B,G,X,Q,U	unchanged
TI(NTI)	y	y	y	y	y	y	y	y	y				required output pts	actual output points
NTI	y	y	y	y	y	y	y	y	y				dim X,U,Q,D,PHI,BMI	unchanged
IEXT				y									ext interval flag	unchanged
TSW(m)						y							–	detected switching pts
TSW(NSW)							y						–	switching points
NSW						y	y						# possible sw pnts	unchanged
NRSW						y	y						–	# detected switching pts
X(N,NTI)	y	y	y		y	y	y	y					–	X(i,k)=X_k(i)
X(N,NTI,N)				y					y				–	X(i,k,j)=X_k(i) solution j
X(N,NRI)										y	y		–	solution X_k(i)
X(N,NX)												y	–	solution X_k(i)
NX												y	dim X	unchanged
NRSOL				y									–	solution uniqueness flag
NRSOL									y				–	# indep. eigensolutions
NU	y	y	y	y	y	y	y	y	y	y	y		dim U, PHI,V	unchanged
U(NU,NTI)	y	y	y	y	y	y	y	y	y				–	uppertriangular U_k
U(NU,NRI)										y	y		–	uppertriangular U_k
V(NU,NRI)										y	y		–	uppertriangular V_k
Q(N,N,NTI)	y			y	y	y	y	y	y				–	orthogonal matrix Q_k
Q(N,NQD,NTI)		y	y										–	orthogonal matrix Q_k
Q(N,N,NRI)										y	y		–	orthogonal matrix Q_k
NQD		y	y										dim Q,ZI,D	unchanged
D(N,NTI)	y				y	y	y	y					–	inhom term d_k if applic
D(NQD,NTI)		y	y										–	inhom term d_k if applic
D(N,NRI)										y	y		–	inhom term d_k of transformed recursion
Z(L)							y					y	–	values of the L parameters

ITEM	GE	PS	SE	IN	MP	MI	PA	DD	EI	S1	S2	S3	on entry(input)	on exit(output)
KU				y									–	# unbounded growing modes on [A,C]
KE													none/previous value	# exponentially growing modes on [B,C]
KEXT				y									none/previous value	total number of outputpoints on [A,C]
KPART	y	y		y					y				–	global k-partition of U_k
KPART(m)					y								–	k partition on interval j
KPART(NSW)							y						–	k partition on interval j
KPART(k)								y					–	k partition on interval j
KP										y	y		–	global k part. of transf *uppertriang. recursion*
KP(NSW)						y							–	global partition index on interval j
EPS										y	y	y	machine constant	unchanged
COND										y	y	y	–	estimate of cond. number
AF										y	y	y	–	estimate of amplif. factor
BMI(N,N,NTI)						y							–	BC matrix of discretised integral BC at TI(j)
CI(N,NTI,L)							y						–	NxL matrix C_j
YI(N,NTI,L)							y						–	part. matrix solution
ZI(NQD,NTI)		y	y										–	particular solution of multishooting recursion
PHI(NU,NTI)	y	y	y	y	y	y	y	y	y				–	fund. solution of multishooting recursion
PHI(NU,NRI)										y	y		–	fund. solution of transf uppertriang. recursion
W(LW)	y	y	y	y	y	y	y	y	y	y	y	y	–	used as workspace
LW	y	y	y	y	y	y	y	y	y	y	y	y	–	unchanged
IW	y	y	y	y	y	y	y	y	y	y	y	y	–	used as workspace
LIW	y	y	y	y	y	y	y	y	y	y	y	y	–	unchanged
IERROR	y	y	y	y	y	y	y	y	y	y	y	y	–	errorcode

References

[1] Uri M. Ascher, Robert M.M. Mattheij, and Robert D. Russell. *Numerical Solution of Boundary Value Problems for Ordinary Differential Equations.* Prentice Hall, 1988.

[2] O. Caprotti and A.M. Cohen. On the role of openmath in interactive mathematical documents. *J. Symbolic Computations, special issue on the integration of computer algebra and decuction systems*, 2001.

[3] P. de Haas and J. Maubach. Numlab user's guide. *Internal Report,* `http://www.win.tue.nl/ maubach/numlab/documentation/usersguide/version1.2/usersguide/index.html`, 2001.

[4] P. de Haas and J. Maubach. Numlab user's guide (french). *Internal Report,* `http://www.win.tue.nl/ maubach/numlab/documentation/usersguide/version1.2/usersguidefr/index.html`, 2001.

[5] J. MacCrisken J. Kodosky and G. Rymar. Ieee workshop on visual languages. pages 34–39, 1991.

[6] A. Telea J. Maubach. Numerical laboratory for computation and visualisation. *Computing and Visualization in Science, accepted*, 2001.

[7] H.R. Lewis and C.H. Papadimitriou. *Elements of the Theory of Computation.* Prentice-Hall, International (UK) Limited, London, 1981.

[8] R.M.M. Mattheij and J. Molenaar. *Ordinary differential equations in theory and practice.* Wiley, 1996.

[9] R.M.M. Mattheij and G.W.M. Staarink. Boundpack numerical software for linear boundary value problems, part one, february 1992.

[10] R.M.M. Mattheij and G.W.M. Staarink. Boundpack numerical software for linear boundary value problems, part two, february 1992.

[11] Alfredo Delgado Mikhail Auguston. Ieee proceedings on visual languages. pages 152–159, 1997.

[12] A. Telea. Combining object orientation and dataflow modeling in the vission simulation system. In R. Mitchell, A. C. Wills, J. Bosch, and B. Meyer, editors, *Proceedings of TOOLS'99 Europe, Nancy 3-8 June 1999*, pages 56–65. IEEE Computer Society Press, 1999.

[13] A. Telea. Numlab reference manual. *Internal Report,* `http://www.win.tue.nl/ maubach/numlab/documentation/referenceguide/mcdoc/index.html`, 2000.

[14] A. Telea and J. J. van Wijk. Vission: An object oriented dataflow system for simulation and visualization. In E. Groeller and W. Ribarsky, editors, *Proceedings of IEEE VisSym '99*, pages 95–104. Springer, 1999.

[15] A. C. Telea. *Visualisation and Simulation with Object-Oriented Networks.* Technical University at Eindhoven, The Netherlands, 2001.

[16] R.A. van Es. Ode toolbox elements for numlab, april 2001.

Index

Author R.A. van Es

Title On the Design of a Data-Flow Oriented Visual Programming Language for Scientific Computing

Full report:

Prof.Dr. R.M.M. Mattheij	Scientific Computing Group
Dr. J.M.L. Maubach	Scientific Computing Group
Dr. A.C. Telea	Computer Graphics, Computer Visualisation
	Eindhoven University of Technology
	Dept. of Mathematics and Computing Science, HG8
	P.O. Box 512
	5600 MB Eindhoven

Ir. R.A. van Es