Comput Visual Sci (2004)
Digital Object Identifier (DOI) 10.1007/s00791-004-0142-3

Computing and
Visualization in Science

Regular article

The NumLab numerical laboratory for computation and visualisation

J. Maubach, A. Telea

Department of Mathematics and Computer Science, Eindhoven University of Technology, Den Dolech 2, 5600 MB Eindhoven, The Netherlands
(e-mail: {maubach, alext}@win.tue.nl)

Received: 29 May 2000 / Accepted: 13 May 2002
Published online: 17 August 2004 – © Springer-Verlag 2004

Communicated by: G. Wittum

Abstract. A large range of software environments addresses numerical simulation, interactive visualisation and computational steering. Most such environments are designed to cover a limited application domain, such as finite element or finite difference packages, symbolic or linear algebra computations or image processing. Their software structure rarely provides a simple and extensible mathematical model for the underlying mathematics. Thus, assembling numerical simulations from computational and visualisation blocks, as well as building such blocks is a difficult task for the researcher in numerical simulation.

This paper presents the NumLab environment, a single numerical laboratory for computational and visualisation applications. Its software architecture one-to-one models fundamental numerical mathematical concepts and presents a generic framework for a large class of computational applications. Partial and ordinary differential equations, transient boundary value problems, linear and non-linear systems, matrix computations, image and signal processing, and other applications all use the same software architecture and are built in a simple and interactive visual manner. NumLab's one-to-one modelled mathematical concepts are illustrated with various applications.

1 Introduction

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

From a structural point of view, such software environments can be classified into three categories (see for instance [39]): Libraries, turnkey systems, and application frameworks.

Libraries for numerics such as LAPACK [2], NAGLIB [37], or IMSL [26], or for visualisation such as OpenGL [27], Open Inventor [50], or VTK [44], provide services in the form of data structures and functions. Libraries are usually easy to extend with new data types and functions. However, using libraries to build a complete computational or visualisation application requires involved programming.

Turnkey systems, such as Matlab [33], Mathematica [32], or the many existing dedicated numerical simulators on the market, are simpler to use than libraries to build a complete application. However, extending the functionality of such systems is usually limited to a given application domain, as in the case of the dedicated simulators, or to a fixed set of supported data types, as in the case of the Matlab programming environment.

Application (computational) frameworks, such as the Diffpack and SciLab systems for solving differential equations [11, 43] or the Oorange system for experimental mathematics [23] combine the advantages of the libraries and turnkey systems. On one hand, frameworks have an open structure, similarly to libraries, so they can be extended with new components, such as solvers, matrix storage schemes, or mesh generators. On the other hand, some (notably visualisation) frameworks offer an easy manner to construct a complete application that combines visualisation, numerics, and user interaction. This is usually provided by means of visual programming tools such as Matlab's Simulink [33] or the dataflow network editing tools of the AVS [49], IRIS Explorer [1], or Oorange [23] frameworks. In these frameworks, applications are constructed by assembling visual representations (icons) of the computational or visualisation components in a network. Program execution is implemented in terms of computational operations on the network nodes and data flows between these nodes respectively.

With the above in mind, let us consider how the NumLab environment integrates the advantages of the above architectures. On the level of libraries, NumLab's C++ routines call Fortran, Pascal, C, and C++. Next, similar to a turnkey system, NumLab offers full integration of visualisation and numerical computation, and implements communication with other environments such as Simulink [33] and MathLink [32]. On the application framework level, NUMLAB provides *interactive application construction* with its visual programming dataflow system VISSION [46,47]. Furthermore, NUMLAB provides an object-level (subroutine-level) make-concept which allows for interactive program validification.

In order to better address NUMLAB's merits on all levels, we need a closer look at computational frameworks. Though efficient and effective, most existing computational frameworks are limited in several respects. First, limitations exist from the perspectives of the end user, application designer, and component developer [4, 19, 39, 46].

First, few computational frameworks facilitate convenient interaction between visualisation (data exploration) and computations (numerical exploration), both essential to the end user.

Secondly, from the application designer perspective, the visual programming facility, often provided in visualisation frameworks such as AVS or Explorer [1,49], usually is not available for numerical frameworks. Conversely, it is quite difficult to integrate large scale computational libraries in visualisation frameworks.

Finally, from the numerical component developer perspective, understanding and extending a framework's architecture is still (usually) a very complex task, albeit noticeably simplified in object-oriented environments such as [11,44].

Next to limitation with respect to the three types of users, many computational frameworks are constrained in a more structural manner: Similar mathematical concepts are not factored out into similar software components. As a consequence, most existing numerical software is heterogeneous, thus hard to deploy and understand. For instance, in order to speed up the iterative solution of a system of linear equations, a preconditioner is often used. Though iterative solvers and preconditioners fit into the same mathematical concept – that of an approximation $\boldsymbol{x}$ which is mapped into a subsequent approximation $z \approx \boldsymbol{F}(\boldsymbol{x})$ – most computational software implements them incompatibly, so preconditioners can not be used as iterative solvers and vice versa [11].

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

The design of NUMLAB addresses all the above problems. NUMLAB is a numerical framework which provides C++ software components (objects) for the development of a large range of interdisciplinary applications (PDEs, ODEs, non-linear systems, signal processing, and all combinations). Further, it provides interactive application design/use with its visual programming dataflow system VISSION [46,47], data interchange (e.g. via Simulink and MathLink), and can be used both in a compiled and interpreted fashion. Its computational libraries factor out fundamental notions with respect to numerical computations (such as evaluation of operators $z = \boldsymbol{F}(\boldsymbol{x})$ and their derivatives), which keeps the amount of basic components small. All components of these libraries are aware of dataflow, even in the absence of the VISSION dataflow system, and can for instance call back to see whether provided data is valid.

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

2 The mathematical framework

In order to reduce the complexity of the entire software solution, we show how NUMLAB formulates different mathematical concepts with a few basic mathematical notions. It turns out that in general NUMLAB's components are either operators $\boldsymbol{F}$, or their vector space arguments $\boldsymbol{x}, \boldsymbol{y}$. The most frequent NUMLAB operations are therefore operator evaluations $\boldsymbol{F}(\boldsymbol{x})$ and vector space operations such as $\boldsymbol{x}+\boldsymbol{y}$. Important is the manner in which NUMLAB facilitates the construction of complex problem-specific operators (for instance transient Navier–Stokes equation with heat transfer), and related complex solvers. NUMLAB offers:

1. *Problem-specific operators*: Transient Finite Element, Volume, Difference operators $\boldsymbol{F}$ for transient boundary value problems (BVPs); Operators which formulate systems of ordinary differential equations (ODEs); operators which act on linear operators (for instance image filters). The operator framework is open, users can define customised operators $z = \boldsymbol{F}(\boldsymbol{x})$.
2. *Problem-specific solvers for systems of ODEs*: Time-step and time-integration operators formulated with the use of (parts of) the problem-specific operators mentioned above. The former operators require non-linear solvers for the computation of solutions.
3. *Solvers for systems of non-linear equations*: Such systems are operators, and their solution is reduced to the solution of a sequence of linear systems.
4. *Solvers for systems of linear equations*: Such systems are also operators $\boldsymbol{F}(\boldsymbol{x}) = \boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}$. Their solution is reduced to a sequence of operator evaluations and vector space operations.

The reduction from one type of operator into another is commented on in the subsections of Sect. 2, in the reverse order of the itemisation above. Thus, Sect. 2.1, examines systems of (non-)linear equations and preconditioners, Sect. 2.2 considers the reduction of systems of ODEs to non-linear systems, and Sect. 2.3 deals with an initial boundary value problem. The presented mathematical reductions are de facto standards, new is NumLab's software implementation which maps one to one with these techniques.

2.1 Non-linear systems and preconditioners

This subsection presents NUMLAB's operator approach, and demonstrates how operator evaluations reduce to repeated vector space operations and operator evaluations. This is illustrated by means of examples, which include (non-)linear systems, and preconditioning techniques.

First, consider linear systems of the form $\boldsymbol{F}(\boldsymbol{x}) = \boldsymbol{f}$. Here $\boldsymbol{F}(\boldsymbol{x}) = \boldsymbol{A}\boldsymbol{x}$ is a linear operator, with an N by N coefficient matrix $\boldsymbol{A}$, and $\boldsymbol{f} \in \mathbb{R}^N$ is a right hand side. The NumLab implementation of the evaluation $z = \boldsymbol{F}(\boldsymbol{x})$ is:

```
F.eval(z, x);
```

The actual implementation of `eval()` varies with the application type (for instance full matrix, sparse matrix, image, etc.). Though z is a resulting value, its initial value can be used for computations (for instance as an initial guess).

Next, NumLab formulates a linear system $\boldsymbol{F}(\boldsymbol{x}) = \boldsymbol{f}$ with the use of an affine operator $\boldsymbol{G}$:

$$\boldsymbol{G}(\boldsymbol{x}) = \boldsymbol{F}(\boldsymbol{x}) - \boldsymbol{f}. \tag{1}$$

The user constructs this NumLab system with

```
G.setO(F);
G.setI(f);
```

and computes the residual $z = \boldsymbol{G}(\boldsymbol{x})$ with:

```
G.eval(z, x);
```

The routines with name `set-` provide $\boldsymbol{G}$ with the linear operator and a right hand side vector.

Next, let $\boldsymbol{x} \in \mathbb{R}^N$ be a given vector, and focus on the solution(s) of $\boldsymbol{G}(z) = \boldsymbol{x}$, i.e, on solution methods for affine operators. Assume that operator $\boldsymbol{R}$ approximates $\boldsymbol{G}^{-1}$:

$$\boldsymbol{G}(z) = \boldsymbol{x} \Longleftrightarrow z = \boldsymbol{G}^{-1}(\boldsymbol{x}) \Longleftrightarrow z \approx \boldsymbol{R}(\boldsymbol{x}) . \tag{2}$$

For the sake of demonstration, and without loss of generality, we assume that $\boldsymbol{R}$ is a left-preconditioned Richardson iterative solution method, with preconditioner $\boldsymbol{P}$. Such a method is based on a successive substition process:

$$z^{(k+1)} = z^{(k)} - \boldsymbol{P}\left(\boldsymbol{G}\left(z^{(k)}\right) - \boldsymbol{x}\right), \tag{3}$$

which terminates as soon as $\boldsymbol{S}(\boldsymbol{P}(\boldsymbol{G}(z^{(k)}) - \boldsymbol{x})) = 0$, for a user-provided stopping criterion $\boldsymbol{S}$: $\mathbb{R}^n \mapsto \{0, 1\}$. This recursion will converge if for instance $\boldsymbol{G}$ is as in (1) with $\boldsymbol{F}$ positive definite, and if $\boldsymbol{P}(\boldsymbol{x}) = h\boldsymbol{x}$ with h positive and small enough.

The related NumLab operator $\boldsymbol{R}$ for (3) is defined by its implementation of its `eval()` method:

```
R.eval(z, x)
{
 P.setO(G);
 repeat
 {
  G.eval(r, z);
  r -= x;
  P.eval(s = 0, r);
  z -= s;
 }
 while (S(s) > 0);
}
```

The system $\boldsymbol{G}(z) = \boldsymbol{x}$ is solved with a few instructions:

```
R.setO(G).setP(T).setS(S);
R.eval(z, x);
```

Observe that solver $\boldsymbol{R}$ uses z both as initial guess $z^{(0)} \in \mathbb{R}^N$ and final approximate solution, whereas preconditioner $\boldsymbol{P}$ must use $\boldsymbol{0}$ as an initial guess. If a preconditioner is not provided, a default – the identity operator – is substituted. The stopping critria are similarly dealt with. Next, as could be observed, operators can make use of operators: The preconditioner for Richardson's algorithm could have been Richardson's algorithm itself, a diagonal preconditioner, an (incomplete) LU factorisation, and so forth. Furthermore, the `eval()` methods of the solver $\boldsymbol{R}$, preconditioner $\boldsymbol{P}$ and system $\boldsymbol{G}$ are syntax-wise identical.

The pseudo code for $\boldsymbol{R}$ above executes `P.setO(G)`, so preconditioner $\boldsymbol{P}$ can use (has access to) $\boldsymbol{G}$ and its Jacobian. Further, the linear system $\boldsymbol{F}(z) = \boldsymbol{f}$ could have been solved directly with $\boldsymbol{R}$:

```
R.setO(F);
R.eval(z, \f);
```

NumLab formulates systems, solvers and preconditioners all with the use of `set-` and `eval()` syntax – though related mathematical concepts differ. Few other methods such as `update()` exist, and relate to data flow concepts, outside the current paper's scope.

A closer examination of the Richardson operator reveals more information of interest. NumLab implements all its operator evaluations with: (1) Vector space operations; and (2) all which remains: Nested operator evaluations. This is clearly demonstrated by $\boldsymbol{R}$'s implementation above:

$$\begin{aligned} \boldsymbol{r} &\overset{(1)}{=} \boldsymbol{G}\left(z^{(k)}\right) ; \\ \boldsymbol{r} &\overset{(2)}{=} \boldsymbol{r} - \boldsymbol{x} ; \\ \boldsymbol{s} &\overset{(1)}{=} \boldsymbol{P}(\boldsymbol{r}) ; \\ z^{(k+1)} &\overset{(2)}{=} z^{(k)} - \boldsymbol{s} , \end{aligned} \tag{4}$$

where, $^{(1)}$ denotes *operator evaluation* and $^{(2)}$ *vector space operation*. This clear cut classification of operations thoroughly simplifies the mathematical framework.

Though NumLab regards preconditioning as approximate function evaluation – which simplifies its framework – this does not solve the problem of proper preconditioning. Specific iterative solution methods might require preconditioners to preserve for instance symmetry (such as the preconditioned gradient method PCG [8]) or at least positive definiteness of the symmetric part (for minimal residual methods, see [42] for GMRES and [5] for GCGLS). All iterative solvers have some requirements: Robust methods (e.g. [30]), multi-level methods (e.g. [7] and [31]), multi-grid methods (e.g. [25]), and incomplete factorisation methods (e.g. [24]).

The application designer should keep these mathematical restrictions in mind, when designing a suitable solver for the problem at hand.

Similar to linear systems, NumLab also formulates non-linear systems with the use of operators $\boldsymbol{G}$, and looks for solutions of $\boldsymbol{G}(z) = \boldsymbol{x}$. The Jacobian (Frechet derivative) of a (non-linear) operator $\boldsymbol{G}$ at point $\boldsymbol{x}$ is denoted by $D\boldsymbol{G}(\boldsymbol{x})$ – or by $D\boldsymbol{G}$ if $\boldsymbol{G}$ is linear.

Related non-linear solvers are again formulated as operators. Non-linear operators $\boldsymbol{G}$ which do not provide derivative evaluation, can be solved with the use of a fixed point method (comparable to the Richardson method above), or with a combinatorial fixed point method [48] (a multi-dimensional variant of the bisection method). Non-linear operators $\boldsymbol{G}$ which provide derivative evaluation can also be solved with (damped, inexact) Newton methods (see [18] and [20]). A typical NUMLAB code for an undamped Newton method is:

```
Newton.eval(z, x)
{
 repeat
 {
  G.eval(r, z);
  r -= x;

  Solver.setO(G.getJacobian(z));
  Solver.eval(s, r);
  z -= s;
 }
 while (S(s) > 0);
}
```

and a system $\boldsymbol{G}(\boldsymbol{z}) = \boldsymbol{x}$ is solved by this method with:

```
Newton.setO(G).setSolver(R);
Newton.eval(z, x);
```

where Richardson's method is used to solve the linear systems.

Again, the application designer should take care that the fixed point function is chosen properly, so it preserves properties of $\boldsymbol{F}$, such as symmetry and positive definiteness of the symmetric part.

In order to close this section on systems of equations and solvers, note that images are also treated as operators

$$\boldsymbol{F}(\boldsymbol{x}) = \boldsymbol{A}\boldsymbol{x}\,, \tag{5}$$

where $\boldsymbol{A}$ is a matrix (or block-diagonal) matrix of colour intensities. Thus, image visualisation reduces to Jacobian visualisation. An application of the above to image processing is illustrated in Fig. 4m.

2.2 Ordinary differential equations

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

$$\frac{d}{dt}\boldsymbol{x}(t) = \boldsymbol{E}(t, \boldsymbol{x}(t)) \quad (t > 0)\,, \qquad \boldsymbol{x}(0) = \boldsymbol{x}_0\,. \tag{6}$$

Let $h > 0$ denote the discrete time-step, and define $t_k = kh$ for all $k = 0, 1, 2, \ldots$. Provided with an approximation $\boldsymbol{x}^{(k)}$ of $\boldsymbol{x}(t_k)$, a fixed-step Euler backward method determines an approximation $\boldsymbol{x}^{(k+1)}$ of $\boldsymbol{x}(t_{k+1})$

$$\boldsymbol{x}^{(k+1)} - \boldsymbol{x}^{(k)} = h\boldsymbol{E}\left(t_{k+1}, \boldsymbol{x}^{(k+1)}\right), \tag{7}$$

which can be rewritten as

$$\boldsymbol{x}^{(k+1)} - \boldsymbol{x}^{(k)} - h\boldsymbol{E}\left(t_{k+1}, \boldsymbol{x}^{(k+1)}\right) = \boldsymbol{0}\,. \tag{8}$$

Define the operator $\boldsymbol{T}$ as follows:

$$\boldsymbol{T}(\boldsymbol{x}) = \boldsymbol{x} - \boldsymbol{x}^{(k)} - h\boldsymbol{E}\left(t_{k+1}, \boldsymbol{x}\right). \tag{9}$$

Then $\boldsymbol{x}^{(k+1)}$ is a solution of $\boldsymbol{T}(\boldsymbol{x}) = \boldsymbol{0}$. Of course, $\boldsymbol{T}$ depends on the user-provided values $\boldsymbol{x}^{(k)}$, t_k and h. The NUMLAB evaluation code of $\boldsymbol{z} = \boldsymbol{T}(\boldsymbol{x})$ is:

```
T.eval(z, x)
{
 E.setT(t_k + h);
 E.eval(z, x);
 z *= h;
 z += x;
 z -= x_k;
}
```

Next, the approximate solution $\boldsymbol{x}^{(k+1)}$ at time t_{k+1} is computed with:

```
T.setT(t_k).setX(x_k).setH(h);
Newton.setO(T).setSolver(R);
Newton.eval(z, 0);
```

The operator formulation $\boldsymbol{x}^{(k+1)} = \boldsymbol{T}^{-1}(\boldsymbol{0})$ applies to all explicit methods such as Runge–Kutta type methods [13], as well as to all implicit discretisation methods, such as Euler Backward and Backward Difference Formulas (BDF) [22].

It is obvious that the evaluation of $\boldsymbol{T}$ at a given $\boldsymbol{x}$ again only involves *vector space operations* and *operator evaluations*. Solving $\boldsymbol{T}(\boldsymbol{x}) = \boldsymbol{0}$ can thus be done by several methods: Successive substitution, Newton type methods, preconditioned methods, etc.

Naturally, a time-step integrator complements the time-step mechanism. NUMLAB provides standard fixed time-step methods and – required for stiff problems – adaptive time-step integrators of the PEC and PECE type [34]. An example is the solution of the Lotka-Volterra predator-prey problem, shown in Fig. 4l. Phase-plane plots can also be generated.

2.3 Partial differential equations and initial boundary value problems

In order to show how partial differential equations (PDEs) are reduced to (non-)linear systems of equations, consider an initial boundary value problem. Let $\Omega \subset \mathbb{R}^d$ be the bounded region of interest, and let $\partial\Omega$ denote its boundary. We denote points in this region with $\boldsymbol{c} \in \Omega$ ($\boldsymbol{x}$ is reserved for vectors and related iterands). The problem of interest is: Find a solution u on $[0, \infty) \times \Omega$ which satisfies

$$\frac{\partial}{\partial t}u = \Delta u + f \qquad (t > 0)\,, \tag{10}$$

subject to initial condition $u(0, \boldsymbol{c}) = u_0(\boldsymbol{c})$ for all $\boldsymbol{c} \in \Omega$, and boundary conditions $u(t, \boldsymbol{c}) = \gamma(\boldsymbol{c})$ for all $t \in [0, \infty)$ and $\boldsymbol{c} \in \partial\Omega$. For the sake of presentation, the boundary conditions are all assumed to be of Dirichlet type. With a method of Lines (MOL) approach, (10) fits into the framework (6), for a suitable operator $\boldsymbol{E}$, to be defined.

As an alternative, one can first discretise in time, and next discretise in space, or simultaneously discretise with respect to both (see for instance [6]).

For a MOL solution of (10), the region of interest Ω is covered with a grid of elements (with the use of a uniform, Delaunay, or bisection type [35] grid generator). Next, the static equation $-\Delta u = f$ is discretised with one of the available methods (standard conforming higher order finite elements, and non-forming elements as for instance in [29]).

A standard Galerkin approach assumes that the solution u is in a linear vector space V with basis $\{v_j\}_{j=1}^N$. For the method of lines approach applied to (10) one sets

$$u(t, \boldsymbol{c}) = \sum_i x_i(t) v_i(\boldsymbol{c}) \,. \tag{11}$$

for all time t and $\boldsymbol{c} \in \Omega$. Functions in V are identified with their coefficient vectors in $\mathbb{R}^n$, so u is identified with $\boldsymbol{x}$. Multiplication of (10) with (test) functions $\{v_j\}_{j=1}^N$, followed by partial integration over Ω leads to a system of ODEs

$$\boldsymbol{M}\frac{d}{dt}\boldsymbol{x}(t) = -\boldsymbol{G}(\boldsymbol{x}(t))\,, \qquad (t > 0)\,. \tag{12}$$

Here,

$$[\boldsymbol{G}(\boldsymbol{x})]_i = \int \left[\nabla \left(\sum_{j=1}^N x_j v_j \right) \nabla v_i - f v_i \right] , \tag{13}$$

if variable $i = 1, \dots, N$ is not related to a Dirichlet point and $[\boldsymbol{G}(\boldsymbol{x})]_i = 0$ otherwise, and

$$[\boldsymbol{M}]_{ij} = \int [v_j v_i], \tag{14}$$

if neither variable i nor variable j is related to a Dirichlet point, and $[\boldsymbol{M}]_{ij} = \delta_{ij}$, the Kronecker Delta otherwise. $\boldsymbol{M}$ is a standard finite element mass-matrix.

The Jacobian $D\boldsymbol{G}$ of $\boldsymbol{G}$ is

$$[D\boldsymbol{G}]_{ij} = \int [\nabla v_j \nabla v_i] \tag{15}$$

if neither variable i nor variable j is related to a Dirichlet point, and $[D\boldsymbol{G}]_{ij} = \delta_{ij}$, the Kronecker Delta otherwise.

Functions in the linear vector space V do not need to satisfy the Dirichlet boundary conditions. Let $\alpha \colon \partial\Omega \mapsto \mathbb{R}$. Define the set (not necessarily a vector space)

$$V^\alpha = \{x \in V : x = \alpha \text{ at } \partial\Omega\}. \tag{16}$$

Then V^0 (homogeneous boundary conditions) is a vector space, and V^γ is the set of all function which satisfy the Dirichlet boundary conditions.

The solution z of $\boldsymbol{G}(z) = \boldsymbol{0}$ is obtained by application of a full Newton step

$$z- > z + [D\boldsymbol{G}(z)]^{-1}[-\boldsymbol{G}(z)] \tag{17}$$

to an initial guess $z^{(0)}$.

The NumLab code for the related undamped Newton method is:

```
Newton.set0(G).setSolver(R);
Newton.eval(z, 0);
```

In the example code, Richardon's iterative solver `R` is used for the solution of $\boldsymbol{G}(\boldsymbol{x}) = \boldsymbol{0}$.

Note that operator $\boldsymbol{G}$ in (13) maps V onto V^0, and that its Jacobian matrix $D\boldsymbol{G}$ in (15) maps V^0 onto V^0. Assume that the (iterative) solver for the solution of the linear maps (1) V^0 onto V^0 and (2) is the identity on $V - V^0$. Then, by induction, also (17) satisfies both assumptions. Because all common linear solvers (PCG, GCGLS, CGS, Peaceman–Rachford, etc.) map V^0 onto V^0, *all of* NumLab*'s solvers map* V^γ onto V^γ. This holds for the (non-)linear (iterative) solvers, as well as for the solvers for systems of ordinary differential equations.

For linear systems $\boldsymbol{G}(z) = \boldsymbol{0}$ with $\boldsymbol{G}$ affine, as for instance in (13), the use of Newton's method (17) may seem an overkill. However, this is not the case: Under the assumption that all coefficients of $\boldsymbol{x}$ are degrees of freedom – including the ones related to Dirichlet points – the solution of $\boldsymbol{G}(z) = \boldsymbol{0}$ requires the solution of (17) above.

In order to see this, define the linear system $\boldsymbol{F}(\boldsymbol{x}) = \boldsymbol{f}$ with

$$[\boldsymbol{F}(\boldsymbol{x})]_i = \int \left[\nabla \left(\sum_{j=1}^N x_j v_j \right) \nabla v_i \right] , \tag{18}$$

and

$$[f]_i = \int [f v_i]\,, \tag{19}$$

for all i. The problem $\boldsymbol{F}(\boldsymbol{x}) = \boldsymbol{f}$ has no unique solution because $\boldsymbol{F}$ is singular.

Under the assumption that we compute with all coefficients x_i, $\boldsymbol{F}(\boldsymbol{x}) = \boldsymbol{f}$ is below transformed in a standard manner, which results in the system $\boldsymbol{G}(\boldsymbol{x}) = \boldsymbol{0}$, and requires solution method (17).

First, define the projection $C \colon \mathbb{R}^N \mapsto \mathbb{R}^N$ by

$$\begin{aligned}[\boldsymbol{C}(\boldsymbol{x})]_i &= x_i \text{ for all related non-Dirichlet supports } \boldsymbol{c}_i\,, \\ &= 0 \;\text{ elsewise}\,. \end{aligned} \tag{20}$$

Next, the vector $\boldsymbol{x}$ is coefficient-wise split into a vector which contains all Dirichlet related function values $\boldsymbol{x}^{(0)}$ and interior degrees of freedom $\boldsymbol{d}$, i.e., we set

$$\boldsymbol{x} = \boldsymbol{x}^{(0)} + \boldsymbol{d}. \tag{21}$$

Then $\boldsymbol{x}^{(0)}$ turns out to be the solution to

$$\left((I - \boldsymbol{C}) + \boldsymbol{C}D\boldsymbol{F}\left(\boldsymbol{x}^{(0)}\right)\boldsymbol{C}^T\right)\boldsymbol{d} = \boldsymbol{C}\left(\boldsymbol{f} - \boldsymbol{F}\left(\boldsymbol{x}^{(0)}\right)\right)\,. \tag{22}$$

This shows that the for the solution of a linear boundary value problem $\boldsymbol{F}(\boldsymbol{x}) = \boldsymbol{f}$, we must solve (17), and in fact exactly solve $\boldsymbol{G}(\boldsymbol{x}) = \boldsymbol{0}$.

The standard splitting (21) for linear systems makes use of an $x^{(0)} \in V^0$ (in (22)) which is zero at all nodal points, except those at the Dirichlet boundary. This – socalled elimination of boundary conditions – is a poor choice because x_0 has steep gradients near Dirichlet boundaries, whence the induced initial residual $\boldsymbol{r}^{(0)}$ for the iterative solver is large. Fortunately, from (21) it follows that we can also take different $x^{(0)} \in V^0$. In order to minimize the amount of iterative solution steps, we best use a smooth $x^{(0)}$.

Finally, we consider the NumLab formulation of an operator for the solution of (10). This operator, of which the discrete solution of (12) is a root, is contructed similar to the operator constructed in (6)–(9), for $\boldsymbol{E}(t, \boldsymbol{x}) := -\boldsymbol{G}(\boldsymbol{x})$ and $\boldsymbol{T}(\boldsymbol{x}) = \boldsymbol{M}(\boldsymbol{x} - \boldsymbol{x}^{(k)}) - h\boldsymbol{E}(t_{k+1}, \boldsymbol{x})$. This construction is such that a symmetric positive definite Jacobian of $\boldsymbol{G}(\boldsymbol{x})$ implies a likewise Jacobian of $\boldsymbol{T}(\boldsymbol{x})$. Therefore, the initial boundary value problem (10) reduces to a sequence of systems of nonlinear equations.

2.4 Conclusions

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

3 From the mathematical to the software framework

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

Consider the mathematical framework for spaces $\boldsymbol{V}$ and operators $\boldsymbol{F}$ in more detail. In general, let Ω be the bounded polygonal/polyhedral domain of interest, with smooth enough boundary $\partial\Omega$. The linear vector space $\boldsymbol{V} = V_1 \times \cdots \times V_n$ is a cross-product space of n spaces (n is the amount of degrees of freedom). Each space V_i is spanned by basis functions $\{v_{ij}\}_{j=1}^{N_i}$ where $v_{ij}\colon\ \Omega \mapsto \mathbb{R}$. An element $\boldsymbol{x} \in \boldsymbol{V}$ is a vector function from Ω to $\mathbb{R}^n$, and is written as $\boldsymbol{x} = [x_1, \dots, x_n]$, a vector of component functions. Each component $x_i \in V_i$ is a linear combination of basis functions: for all $\boldsymbol{c} \in \Omega$

$$x_i(\boldsymbol{c}) = \sum_{j=1}^{N_i} x_{ij}(t) v_{ij}(\boldsymbol{c})\,. \tag{23}$$

Each element x_i is associated with a unique scalar vector $X_i = [x_{i1}, \dots, x_{iN_i}] \in \mathbb{R}^{N_i}$. At its turn, $\boldsymbol{X}$ denotes the aggregate of these vectors: $\boldsymbol{X} = [X_1, \dots, X_n]$, and $\boldsymbol{X}_{ij} = [X_i]_j$. Summarised, we have vector functions $\boldsymbol{x} = [x_1, \dots, x_n]$ and related vectors of coefficient vectors $\boldsymbol{X} = [X_1, \dots, X_n]$.

Whenever $n = 1$, we use a more standard notation. In this case, the space is V, spanned by basis functions $\{v_j\}_{j=1}^{N}$, and elements $x \in V$ are related to coefficient vector $\boldsymbol{x} = [x_1, \dots, x_N]$.

For most finite element computations, the basis functions v_{ij} of V_i have local support. However,basis functions have global support in spectral finite elements computations. The local supports, also called elements, are created with the use of a triangulation algorithm.

The next subsections describe NUMLAB's software components related to the mathematical concepts discussed in this section: Grid generation in Sect. 3.1, bases generation in Sect. 3.2, vector functions in Sect. 3.3, and related operators in Sect. 3.4.

3.1 The `Grid` module

To be able to define local support for the basis functions v_{ij} later on, we need to discretise the function's domain Ω. This is modelled in the software framework by the `Grid` module, which covers the function's domain with elements e. This `Grid` module takes a `Contour` as input, which describes the boundary $\partial\Omega$ of Ω. The default contour is the unit square's contour.

In NUMLAB, the grid covers regions Ω in any dimension (e.g. 2D planar, manifold or 3D spatial), and consists of a variety of element shapes, such as triangles, quadrilaterals, tetrahedra, prisms, hexahedrals, n-simplices (see [35]), and so on. All grids implement a common interface. This interface provides several but few services. These include: Iteration over the grid elements and their related vertices, topological queries such as the element which contains a given point. The amount of services is a minimum: Modules which use a grid generator and need more service must compute the required relations from the provided information.

Specific `Grid` generator modules produce grids in different manners. NUMLAB contains Delaunay generators, simplicial generators, and regular generators, and "generators" which read an existing grid from a file. An example generator is illustrated in Fig. 4k, which shows a cubic finite element interpolant on a 2-manifold in $\mathbb{R}^3$.

3.2 The `Space` module

The linear vector space $\boldsymbol{V}$ is implemented by the software module `Space`. `Space` takes a `Grid` and `Boundary-Conditions` as inputs. The grid's discretisation in combination with the boundary conditions are used to build the supports of its basis functions v_{ij}. The default boundary conditions are Dirichlet type conditions for all solution components. None, Robin, Neumann and vectorial boundary conditions are specified per boundary part. Recall that elements in $\boldsymbol{V}$ do not have to satisfy the Dirichlet boundary conditions. Recall that elements of $\boldsymbol{V}$ do not have to satisfy the essential boundary conditions.

Because `Grid` has a minimal interface, some information – required by `Space` for the construction of the basis functions – is not provided. Whenever this happens, `Space` internally computes the required information with the use of `Grid`'s services.

A specific `Space` module implements a specific set of basis functions, such as constant, linear, quadratic, or even higher order polynomial degree, matched to the elements' geometry. The interface of the `Space` module follows the mathematical properties of the vector space $\boldsymbol{V}$ presented so far: Elements $\boldsymbol{x}, \boldsymbol{y} \in \boldsymbol{V}$ can be added together or scaled by real values. Furthermore, elements v_{ij} of $\boldsymbol{V}$ are functions, and $\boldsymbol{V}$ permits evaluation at points $\boldsymbol{c} \in \Omega$ of such functions and their derivatives.

It should be kept in mind that elements of $\boldsymbol{V}$ are functions, not linear combinations of functions. Therefore, the name SPACE is somewhat misleading. However, for the brevity of demonstration, the name SPACE will also be used in the sequel.

In most cases, the required basis functions have local support, also called element-wise support. The restriction of global basis function v_{ij} to support e is said to be local function v_{ir}. In software, this is coded as follows: For space component i (so V_i), element e, and local basis function r thereon, `j := j(i, r)` induces basis function v_{ij}. The software implementation is on element-level for efficiency purposes: Given a point $\boldsymbol{c} \in \Omega$, `Space` determines which support e contains $\boldsymbol{c}$ for the evaluation of $v_{ij}(\boldsymbol{c})$.

3.3 The `Function` module

As discussed, a vector function $\boldsymbol{x}\colon \Omega \mapsto \mathbb{R}^n$ in a space $\boldsymbol{V}$ generated by v_{ij} is uniquely related to a coefficient vector $\boldsymbol{X}$ with coefficients X_{ij}. Based on this observation, NumLab software module `Function` implements a vector function $\boldsymbol{x}$ as a block vector of real-valued coefficients $\boldsymbol{X}_{ij}$, combined with a reference to the related `Space` – which contains related functions v_{ij}.

The `Function` module provides services to evaluate the function and its derivatives at a given point $\boldsymbol{c} \in \Omega$. To this end, both $\boldsymbol{x}$'s coefficient vector X and the point $\boldsymbol{c}$ are passed to the `Space` module referred to by $\boldsymbol{x}$. At its turn, the `Space` module returns the value of $\boldsymbol{x}(\boldsymbol{c})$. This is computed following the definition $\boldsymbol{x}(\boldsymbol{c}) = [\sum_j x_{ij} v_{ij}(\boldsymbol{c})]$, as described in the previous section. The computation of the partial derivatives of a given function $\boldsymbol{x}$ in a point $\boldsymbol{c}$ follows a similar implementation.

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

3.4 The `Operator` module

As described previously, an operator $\boldsymbol{F}\colon \boldsymbol{V} \mapsto \boldsymbol{W}$ maps an element $\boldsymbol{x} \in \boldsymbol{V}$ to an element $z \in \boldsymbol{W}$. The evaluation $z = \boldsymbol{G}(\boldsymbol{x})$ computes the coefficients z_{ij} of z from the coefficients x_{ij} of $\boldsymbol{x}$, as well as from the bases $\{v_{ij}\}$ and $\{w_{ij}\}$ of $\boldsymbol{V}$ and $\boldsymbol{W}$ respectively. Next to the evaluation of $\boldsymbol{G}$, derivatives such as the Jacobian operator $\boldsymbol{DG}$ of $\boldsymbol{G}$ are evaluated in a similar manner. Such derivatives are important in several applications. For example, they can be used in order to find a solution of $\boldsymbol{G}(z) = \boldsymbol{x}$, with Newton's method.

The software implementation of the operator notion follows straightforwardly the mathematical concepts introduced in Sect. 2. The implementation is done by the `Operator` module, which offers two services: evaluation of $z = \boldsymbol{G}(\boldsymbol{x})$, coded as `G.eval(z,x)`, and of the Jacobian of $\boldsymbol{G}$ in point $\boldsymbol{y}$, $z = \boldsymbol{DG}(\boldsymbol{y})\boldsymbol{x}$, coded as `G.getJ(y).eval(z,x)`. To evaluate $z = \boldsymbol{G}(\boldsymbol{x})$, the `Operator` module takes two `Function` objects z and $\boldsymbol{x}$ as input and computes the coefficients z_{ij} using the coefficients x_{ij} and the bases of the `Space` objects z and $\boldsymbol{x}$ carry with them. It is important that both the 'input' z and the 'output' $\boldsymbol{x}$ of the `Operator` module are provided, since it is in this way that `Operators` determine the spaces $\boldsymbol{V}$, respectively $\boldsymbol{W}$.

To evaluate $z = \boldsymbol{DG}(\boldsymbol{y})\boldsymbol{x}$, the `Operator` proceeds similarly. Internally, $\boldsymbol{DG}(\boldsymbol{y})$ is usually implemented as a coefficient matrix, and the operation $\boldsymbol{DG}(\boldsymbol{y})\boldsymbol{x}$ is a matrix-vector multiplication. However, the implementation details are hidden from the user ($\boldsymbol{DG}(\boldsymbol{y})\boldsymbol{x}$ may be computed element-wise, i.e. matrix-free), who works only with the `Function` and `Operator` mathematical notions.

Specific `Operator` implementations differ in the way they compute the above two evaluations. For example, a simple `Diffusion` operator $z = \boldsymbol{G}(\boldsymbol{x})$ may operate on a scalar function and produce a function z where $z_i = x_{i-1} - 2x_i + x_{i+1}$. A generic `Linear` operator may produce a vector of coefficients $z = \boldsymbol{A}\boldsymbol{x}$ where $\boldsymbol{A}$ is a matrix. A `Summator` operator $z = \boldsymbol{G}_1(\boldsymbol{x}) + \boldsymbol{G}_2(\boldsymbol{x})$ may take two inputs $\boldsymbol{G}_1$ and $\boldsymbol{G}_2$ and produce a vector of coefficients $z_i = [\boldsymbol{G}_1(\boldsymbol{x})]_i + [\boldsymbol{G}_2(\boldsymbol{x})]_i$. Remark that the modules implementing the `Linear` and `Summator` operators actually have two inputs each. In both cases the function $\boldsymbol{x}$ is the first input, while the second is the matrix $\boldsymbol{A}$ for the `Linear` operator and the operators $\boldsymbol{G}_1$ and $\boldsymbol{G}_2$ for the `Summator` operator. These values could be as well hard-coded in the operator implementation. In both cases however, we see `Operator` as a function of a *single* variable $\boldsymbol{x}$, as described in the mathematical framework.

3.5 The `Solver` module

We model the solving of $\boldsymbol{G}(z) = \boldsymbol{x}$ by the module `Solver` in our software framework. Mathematically, `Solver` is similar to an operator $\boldsymbol{S}\colon \boldsymbol{V} \mapsto \boldsymbol{W}$, where $\boldsymbol{V}$ and $\boldsymbol{W}$ are function spaces. The interface of `Solver` provides evaluation at functions $\boldsymbol{x} \in \boldsymbol{W}$, similarly to the `Operator` module. The implementation of the `Solver` evaluation operation $z = \boldsymbol{S}(\boldsymbol{x})$ should provide an approximation z to $z \approx \boldsymbol{F}^{-1}(\boldsymbol{x})$. However, `Solver` does not provide evaluation of its Jacobian, as this may be undesirably complex to compute in the general case.

Practically, `Solver` takes as input an initial guess `Function` object $\boldsymbol{x}$ and an `Operator` object $\boldsymbol{G}$. Its output z is such that $\boldsymbol{G}(z) = \boldsymbol{x}$. The operations done by the solver are either vector space operations or `Operator` evaluations, or evaluations of similar operators $\boldsymbol{G}(z)$. In the actual implementation, this is modelled by providing the `Solver` module with one or more extra inputs of type `Solver`. In this way, one can for example connect a nested chain of preconditioners to an iterative solver module.

The implementation of a specific `Solver` follows straightforwardly from its mathematical description. Iterative solvers such as Richardson, GMRES, (bi)conjugate gradient, with or without preconditioners, are easily implemented in this software framework.

The framework makes no distinction between a solver and a preconditioner, as discussed in Sect. 2. The sole difference between a solver and a preconditioner in this framework is semantic, not structural. A solver is supposed to produce an *exact* solution of $\boldsymbol{G}(z) = \boldsymbol{0}$ (up to a desired numerical accuracy), whereas the preconditioner is *supposed* to return an *approximate* one. Both are implemented as `Solver` modules, which allows easy cascading of a chain of preconditioners to an iterative solver as well as using preconditioners and solvers interchangeably in applications. Furthermore, the framework makes no structural distinction between direct and iterative solvers. For example, an `ILUSolver` module is implemented to compute an incomplete LU factorisation of its input operator $\boldsymbol{G}$. The `ILUSolver` module can be used as a preconditioner for a `Conjugate-Gradient` solver module. In the case the `ILUSolver` is not connected to the `ConjugateGradient` module's input, the latter performs non preconditioned computations. Alternatively, a `LUSolver` module is implemented to provide a complete LU factorisation of its input operator $\boldsymbol{G}$. The `LUSolver` can be used either directly to solve the equation $\boldsymbol{G}(z) = \boldsymbol{x}$, or as preconditioner for another `Solver` module.

3.6 An object-oriented approach to the software framework

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

As stated in the beginning of this section, a large range of numerical problems can be modelled with these few generic modules. In order to capture the specifics of a given problem, such as the type of PDE to be solved or the basis functions of an approximation space, the generic modules have to be specialised. The specialised modules provide the interface declared by their class, but can implement it in any desirable fashion. For example, a `ConjugateGradient` module implements the `Solver` interface of evaluating $z = \boldsymbol{G}^{-1}\boldsymbol{x}$ by using the conjugate gradient iterative method.

The above architectural requirements are elegantly and efficiently captured by using an object-oriented approach to software design [10, 12, 36, 41]. Consequently, we have implemented our numerical software framework as an object-oriented library written in the C++ language [45]. This design enabled us to naturally model the concepts of basic and specialised modules as class hierarchies. The software framework implements a few base classes `Grid`, `Function`, `Space`, `Operator`, and `Solver`. These base classes declare the interface to their operations. The interface is next implemented by various specialisations of these base classes. An overview of the implemented specialisations follows:

- `Grid:` 2D and 3D grid generators for regular and unstructured grids, and grid file readers;
- `Function:` Several specific functions $v_i j$ are generated, such as cosines, or piecewise (non-)conforming polynomial functions in several dimensions;
- `Space:` There is a single `Space` class, but a multitude of basis functions are implemented, as described further in Sect. 5;
- `Operator:` Operators for several ODEs, PDEs, and non-linear systems have been implemented, such as Laplace, Stokes, Navier–Stokes, and elasticity problems. Next, several operators for matrix manipulation and image processing have been implemented. For example, matrix sparsity patterns can be easily visualised, as in other applications like Matlab (Fig. 4j);
- `Solver:` A range of iterative solvers including bi-conjugate gradient, GMRES, GCGLS, QMR, etc. are implemented. Several preconditioners such as ILU are also provided as `Solver` specialisations, following the common treatment of solver and preconditioner modules previously described.

Besides the natural modelling of the mathematics in terms of class hierarchies, the object-oriented design allows users to easily extend the current framework with new software modules. Implementing a new solver, preconditioner, or operator usually involves writing only a few tens of lines of C++ to extend an existing one. The same approach also facilitates the reuse of existing numerical libraries such as LAPACK [2] or Templates [9] by integrating them in the current object-oriented framework.

4 Transient Navier–Stokes equations

This section examines the mathematical concepts at the foundations of a NumLab solver for transient Navier–Stokes equations. These concepts (1)–(4) in Sect. 2, have been examined in Sects. 2.1–2.3 for small model problems, suited for presentation purposes. Here, these concepts are all worked out in relation to a single problem, the solution of transient Navier–Stokes equations. Section 5 discusses the design of a NumLab application with the NumLab operators discussed here.

The transient Navier–Stokes equations have been chosen since related finite element operators require a finite dimensional cross product vector space $\boldsymbol{V}$ of basis functions, and because the transient formulation leads to differential algebraic equations, and requires solution techniques related to ODEs. The DAE class of equations is non-trivial to solve, and common in industrial problems. Our claim is – see Sect. 5 for details – that NumLab provides a sophisticated framework for the integration of complex Navier–Stokes solvers, not that NumLab provides solvers better than those found in the literature.

First we examine the static problem. For a particular discretisation, we show that there exists a straightforward and lucid relation between the mathematical formulas and the NumLab software implementation: The NumLab implementation of $\boldsymbol{F}$ accomplishes the finite element required (numerical) integration without space $\boldsymbol{V}$ exposing its basis functions and element geometries to $\boldsymbol{F}$.

The static case is followed with the mathematical formulation of the transient problem. We demonstrate that (components of) the static problem operator $\boldsymbol{F}$ can be used in combination with all suitable time-integrators $\boldsymbol{S}$ – suited for indefinite/stiff problems.

Due to the high degree of orthogonality between $\boldsymbol{F}$, $\boldsymbol{V}$ and time-stepper methods, NumLab can and does offer a range of finite element types – higher order, as well as non-conforming Crouzeix-Raviart (see [17]) – on rather arbitrary support geometries: simplices, parallelipipeda, prisms, etc. It facilitates and supports user-defined reference bases and geometries, as well as user-supplied geometries and grid generators. Existing applications do not have to be adapted for new bases and geometries, as long as all required mathematical conditions hold.

4.1 The Navier–Stokes equations

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

$$\begin{cases} -\varepsilon\Delta\boldsymbol{u} + \boldsymbol{u}\nabla\boldsymbol{u} + \nabla p = \boldsymbol{f}\,, \\ \qquad\qquad \nabla\cdot\boldsymbol{u} = 0\,. \end{cases} \tag{24}$$

For the sake of demonstration, all boundary conditions are presumed to be of Dirichlet type (parabolic in/outflow profiles and no-slip along walls).

Problem (24) is discretised with the use of a finite element method. To this end, one first covers Ω by elements with the use of a grid generator module `Grid` (the construction and refinement of a suitable computational grid is a problem of its own (see for instance [21]). Then a triplet of finite dimensional (Hilbert) finite element spaces $\boldsymbol{V} := V_1 \times V_2 \times V_3$ is chosen such that $V_1 \times V_2$ and V_3 satisfy the L.B.B. condition [3]. The NUMLAB implementation creates one `Space` module $\boldsymbol{V}$, provided with three reference `Basis` modules. For the sake of presentation, quadratic conforming finite element bases are used for the velocities (V_1 and V_2), and a piecewise linear conforming finite element basis is used for the pressure (V_3).

Next, the equations (24) are multiplied by test functions $(\boldsymbol{v}, q) \in \boldsymbol{V}$, after which the first one is partially integrated. This procedure results in a variational problem: Find $\boldsymbol{x} = [u_1, u_2, p] = (\boldsymbol{u}, p) \in \boldsymbol{V}$ such that for all $(\boldsymbol{v}, q) \in \boldsymbol{V}$

$$\begin{cases} \displaystyle\int \varepsilon\nabla\boldsymbol{u} : \nabla\boldsymbol{v} - p\nabla\boldsymbol{v} + (\boldsymbol{u}\nabla\boldsymbol{u} - \boldsymbol{f})\boldsymbol{v} = \boldsymbol{0}\,, \\ \qquad\qquad\qquad \displaystyle\int \nabla\cdot\boldsymbol{u}\, q = 0\,. \end{cases} \tag{25}$$

Different finite element discretisations of (24), for instance an O'Seen discretisation, are also possible. In order to facilitate the formulation of a NUMLAB application for our problem, system (25) is now reformulated into operator form: $\boldsymbol{F}(\boldsymbol{X}) = \boldsymbol{0}$.

The operator $\boldsymbol{F}$ related to the discrete variational formulation (25) has three components $\boldsymbol{F} := [\boldsymbol{F}_1, \boldsymbol{F}_2, \boldsymbol{F}_3]$, each related to one equation. For the definition of these components, first define $\boldsymbol{x} = [x_1, x_2, x_3] := [u_1, u_2, p] \in \boldsymbol{V}$, and set $\boldsymbol{z} = [z_1, z_2, z_3] \in \boldsymbol{V}$ (assume we use a Galerkin procedure). Recall that each vector function $\boldsymbol{x}$ is uniquely related to coefficients x_{ij}, at their turn related to functions v_{ij} from Ω to $\mathbb{R}$. The discrete Navier–Stokes operator, discretised in space, now is:

$$Z_{1j} = \boldsymbol{F}(\boldsymbol{X}) = [\boldsymbol{F}_1(X_1, X_2, X_3)]_j$$
$$= \int \varepsilon\nabla x_1\nabla v_{1j} - x_3\partial_x v_{1j} + (x_1\partial_x x_1 + x_2\partial_y x_1 - f_1)v_{1j}$$
$$Z_{2j} = \boldsymbol{F}(\boldsymbol{X}) = [\boldsymbol{F}_2(X_1, X_2, X_3)]_j$$
$$= \int \varepsilon\nabla x_2\nabla v_{2j} - x_3\partial_y v_{2j} + (x_1\partial_x x_2 + x_2\partial_y x_2 - f_2)v_{2j}$$
$$Z_{3j} = \boldsymbol{F}(\boldsymbol{X}) = [\boldsymbol{F}_3(X_1, X_2, X_3)]_j$$
$$= \int (\partial_x x_1 + \partial_y x_2)v_{3j}\,. \tag{26}$$

Here, x_1 is the function related to coefficients X_1, and so forth. It is evident – as stated earlier – that $\boldsymbol{F}$ uses the coefficients of $\boldsymbol{x}$ as well as the bases functions in order to compute the coefficients of the result $\boldsymbol{z}$.

The integrals in (26) are computed support-wise, with the use numerical integration, involving integration points $\boldsymbol{x}_k$. As can be deduced from (26), required are the values $v_{i,r}(\boldsymbol{x}_k)$ and $\nabla v_{i,r}(\boldsymbol{x}_k)$ as well as the values $x_i(\boldsymbol{x}_k)$ and $\nabla x_i(\boldsymbol{x}_k)$. In the NUMLAB code below, these values are returned in arrays `v(i)(k)(r)`, `dv(i)(k)(r)`, `x(i)(k)`, respectively `dx(i)(k)`. The selection `dv(i)(k)(r)(dY)` returns the individual gradient component $\partial_y v_{i,r}(\boldsymbol{x}_k)$. Define `U1 = 0, U2 = 1, P = 2`. The NUMLAB evaluation of `z = F(x)` and `z = DF(x)*y` for support e (typeset to fit this layout) is:

```
Operator z = F(x):

 z(U1)(j(U1)(r)) += qw(k)*
   (eps*dx(U1)(k)*dv(U1)(k)(r) -
    x(P)(k)*dv(U1)(k)(r)(dX) +
   (x(U1)(k)*dx(U1)(k)(dX) +
    x(U2)(k)*dx(U1)(k)(dY) -
    f1(qp(k)) * v(U1)(k)(r)));
 z(U2)(j(U2)(r)) += qw(k)*
   (eps*dx(U2)(k)*dv(U2)(k)(r) -
    x(P)(k)*dv(U2)(k)(r)(dY) +
   (x(U1)(k)*dx(U2)(k)(dX) +
    x(U2)(k)*dx(U2)(k)(dY) -
    f2(qp(k)) * v(U2)(k)(r)));
 z(P)(j(P)(r)) += qw(k)*
   ((dx(U1)(k)(dX) +
     dx(U2)(k)(dY)) * v(P)(k)(r));

Jacobian z = DF(x)*y:

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

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


 z = DF * y;
```

Both evaluation operations have an almost identical loop structure:

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

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

– First, $\boldsymbol{F}$ does not have spaces $\boldsymbol{V}$ and $\boldsymbol{W}$ as input (i.e., as auxiliary variables). The spaces are obtained from the input/output variables. This technique simplifies computational networks.
– Secondly, because $\boldsymbol{F}$ performs numerical integration, it solely requires *the value* of (partial derivatives of) the basis functions at the quadrature points. The basis functions themselves are not required, so $\boldsymbol{F}$ operates orthogonal to $\boldsymbol{V}$ and $\boldsymbol{W}$.
– Finally, the NUMLAB operator models the discrete Navier–Stokes equations in (25) in a convenient fashion. The software implementation is one-to-one with the mathematical syntax, and can in fact be automated.

Finally, recall that the derivative operator acts as the identity operator on Dirichlet point related variables, which requires `fetch` to deliver the related information. This information is also required for non-homogeneous Neumann boundary conditions and Robin conditions.

4.2 The time discretisation

A transient version of the Navier–Stokes equations in the previous section can be formulated as so-called differential algebraical equations (DAEs):

$$\begin{cases} \frac{\partial}{\partial t}\boldsymbol{u} = \varepsilon\Delta\boldsymbol{u} - \boldsymbol{u}\nabla\boldsymbol{u} - \nabla p + \boldsymbol{f}\,, \\ \nabla\cdot\boldsymbol{u} = 0\,, \end{cases} \qquad (t > 0) \tag{27}$$

with initial condition $\boldsymbol{u}(0, \boldsymbol{c}) = \boldsymbol{u}_0(\boldsymbol{c})$ on Ω and boundary conditions $u(t, \boldsymbol{c}) = u_1(\boldsymbol{c})$ for all $t \in [0, \infty)$ and $\boldsymbol{c} \in \partial\Omega$. We now construct a non-linear NUMLAB time-step operator $\boldsymbol{F}$ for a MOL discretisation of (27), which is implicit with respect to the constraint $\nabla\cdot\boldsymbol{u} = 0$.

For the sake of presentation, for a discretisation of the first vectorial equation in (27), we will use a rather basic time-step method: the θ-method – recall the constrained will be treated in an implicit manner below. In practice, for stiff problems – high Reynolds number – one would rather use a backward difference method. For $\theta \in [0, 1]$, the θ-method for a general non-linear system of ODEs

$$\frac{d}{dt}\boldsymbol{u}(t) = \boldsymbol{E}(t, \boldsymbol{u}(t))\,, \tag{28}$$

leads to the recursion:

$$u_0 = \boldsymbol{u}(0)\,,$$
$$\boldsymbol{u}_{k+1} - \boldsymbol{u}_k = h\theta\boldsymbol{E}(t_k, \boldsymbol{u}_k) + h(1-\theta)\boldsymbol{E}(t_{k+1}, \boldsymbol{u}_{k+1})\,. \tag{29}$$

This all fits into the NUMLAB `Operator` style, if we define the time-step operator $\boldsymbol{T}$ – similar to (9) – as follows:

$$\boldsymbol{T}(\boldsymbol{u}) := \boldsymbol{u} - \boldsymbol{u}^{(k)} - h\theta\boldsymbol{E}\left(t_k, \boldsymbol{u}^{(k)}\right) - h(1-\theta)\boldsymbol{E}(t_{k+1}, \boldsymbol{u})\,. \tag{30}$$

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

$$\boldsymbol{T}(\boldsymbol{u}) = \boldsymbol{0}\,. \tag{31}$$

For the solution of (27), we first discretise (27), in a manner similar to how (24) was discretised to obtain (26), with a discrete solution as formulated in (11). This leads to a discretised version of (27):

$$\begin{cases} \boldsymbol{M}\frac{d}{dt}X_1(t) = -\boldsymbol{F}_1(\boldsymbol{X}(t)) \\ \boldsymbol{M}\frac{d}{dt}X_2(t) = -\boldsymbol{F}_2(\boldsymbol{X}(t)) \\ \qquad 0 = \boldsymbol{F}_3(\boldsymbol{X}(t))\,. \end{cases} \tag{32}$$

subject to initial conditions on the two velocity components $X_1(0) = g_0$, $X_1(0) = g_1$. The operators $\boldsymbol{F}_i$ are those defined in (26), and $\boldsymbol{M}$ is the mass matrix. Define $\boldsymbol{Y}(t) = [X_1(t), X_2(t)]$, i.e., $\boldsymbol{X}(t) = [\boldsymbol{Y}(t), X_3(t)]$. Let operator $\boldsymbol{E}(\boldsymbol{X}) := [-\boldsymbol{F}_1(\boldsymbol{X}), -\boldsymbol{F}_2(\boldsymbol{X})]$, then (32) reduces to

$$\begin{cases} \boldsymbol{M}\frac{d}{dt}\boldsymbol{Y} = \boldsymbol{E}(\boldsymbol{X}) \\ \qquad 0 = \boldsymbol{F}_3(\boldsymbol{X})\,, \end{cases} \tag{33}$$

with related initial and boundary conditions. Finally, with the application of the θ-method (30), the discrete solution $\boldsymbol{X}^{(k+1)} = [\boldsymbol{Y}^{(k+1)}, X_3^{(k+1)}]$ of (27) is a root of

$$\boldsymbol{G}(\boldsymbol{X}) := [\boldsymbol{W}(\boldsymbol{X}), \boldsymbol{F}_3(\boldsymbol{X})] = [\boldsymbol{0}, 0]\,, \tag{34}$$

where $\boldsymbol{W}(\boldsymbol{X})$ is defined by

$$\boldsymbol{M}\left(\boldsymbol{Y} - \boldsymbol{Y}^{(k)}\right) - h\theta\boldsymbol{E}\left(t_k, \boldsymbol{X}^{(k)}\right) - h(1-\theta)\boldsymbol{E}\left(t_{k+1}, \boldsymbol{X}\right)\,. \tag{35}$$

Summarising (27)–(35), we have shown that each approximate solution $\boldsymbol{X}^{(k)}$ of $\boldsymbol{X}(t_k)$ must solve a non-linear system of equations $\boldsymbol{G}(\boldsymbol{X}) = \boldsymbol{0}$, which can be supplied to a NUMLAB non-linear solver.

Finally, some remarks and observations. First, the value which operator $\boldsymbol{G}$ attains at $\boldsymbol{X}$, is composed of the values which $\boldsymbol{F}$ attains at related points. Therefore, the Jacobian $\boldsymbol{DG}(\boldsymbol{X})$ can be formulated in terms of $\boldsymbol{DF}$ at related points. The NUMLAB implementation of time-steps exploits this: The basic Jacobian implementation of $\boldsymbol{DG}(\boldsymbol{X})$ is a sequence of call-backs to the Jacobians $\boldsymbol{DF}$. It it pointed out that the saddle point problems related to (32) are hard to solve ([21,40]).

5 Application design and use

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

As stated in Sect. 1, a numerical framework should provide an easy way to construct numerical experiments by assembling predefined components such as grids, problem definitions, solvers, and preconditioners. Next, one should be able to interactively change all parameters of the constructed application and monitor the produced results in a numerical or visual form. Shortly, we need to address the three roles of component development, application design, and interactive use for the scientific computing domain.

We have approached the above by integrating the NumLab component library in the VISSION system. VISSION is a general-purpose environment for VIsualisation and SImulation with Object-oriented Networks. The main feature of VISSION is its capability to load independently developed C++ component libraries and to display them in a visual, iconic form in its network editor user interface (Fig. 1a). The application designer can construct the desired computational or visualisation application by visually assembling the desired components in a dataflow network. VISSION automatically provides graphical user interfaces for all the loaded components, as the example shown in Fig. 1b. Overall, VISSION provides similar code integration, application construction, and steering mechanisms as the AVS, IRIS Explorer, or Oorange environments, and generalises and simplifies their use for arbitrary component libraries written in C++ (as detailed in [46,47]).

As NumLab is written as a C++ component library, its integration into VISSION was easy. Moreover, the structure of NumLab as a set of components that communicate by data streams in order to perform the desired computation matches well VISSION's dataflow application model. As no modification of the NumLab code was necessary, its integration in VISSION took only a few hours of work.

Once all the NumLab components were integrated into VISSION, constructing numerical applications with interactive computational steering and visualisation was easily achieved by using VISSION's visual network construction and end user interaction facilities described above. We shall illustrate these with the Navier–Stokes problem discussed in the previous section.

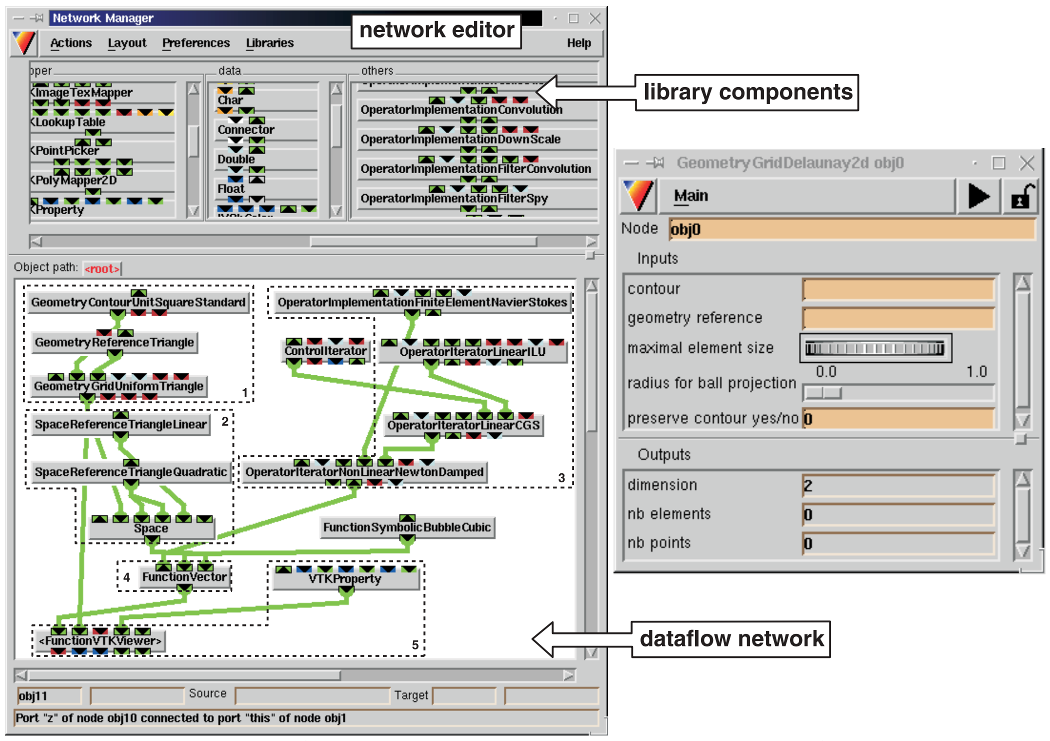
5.1 S Navier–Stokes simulation

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

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

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

In our example, the computational domain is a rectangular region whose boundary is defined by the `GeometryContourUnitSquareStandard` module. This module allows the specification of the rectangle's sizes, as well as a distribution of mesh points on the contour. Next, the `GeometryGridUniformTriangle` module produces a meshing of the rectangle into triangles. The reference triangle geometry is given by the `GeometryReferenceTriangle`. The mesh produced by the `GeometryGridUniformTriangle` module conforms both to the reference element supplied as input and to the boundary points output by the `GeometryContourUnitSquareStandard` module. Different combinations of contour definitions, mesh generators, and reference elements are easily achieved by using different modules. In this way, 2D and 3D regu-


Network Manager
network editor
Actions
Layout
Preferences
Libraries
Help
oper
data
others
<ImageTexMapper
<LookupTable
<PointPicker
<PolyMapper2D
<Property
Char
Connector
Double
Float
OperatorImplementationConvolution
OperatorImplementationDownScale
OperatorImplementationFilterConvolution
OperatorImplementationFilterSpy
library components
Object path: <root>
GeometryContourUnitSquareStandard
GeometryReferenceTriangle
GeometryGridUniformTriangle
1
SpaceReferenceTriangleLinear
2
SpaceReferenceTriangleQuadratic
OperatorImplementationFiniteElementNavierStokes
ControlIterator
OperatorIteratorLinearILU
OperatorIteratorLinearCGS
OperatorIteratorNonLinearNewtonDamped
3
Space
FunctionSymbolicBubbleCubic
4
FunctionVector
VTKProperty
<FunctionVTKViewer>
5
dataflow network
Source
Target
GeometryGridDelaunay2d obj0
Main
Inputs
Outputs
0.0
1.0

obj11

Port "z" of node obj10 connected to port "this" of node obj1

Node obj0

contour

geometry reference

maximal element size

radius for ball projection

preserve contour yes/no 0

dimension 2

nb elements 0

nb points 0

a) b)

Fig. 1. a Navier-Stokes simulation built with NumLab components. **b** User interface for the grid generator module lar and unstructured meshes of various element types such as triangles, quadrilaterals, hexahedra, or tetrahedra can be produced. The produced mesh can be directly visualised or further used to define a computational problem.

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

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

The first task is done by the `SpaceReferenceTriangle-Linear` and `SpaceReferenceTriangleQuadratic` modules, which define linear, respectively quadratic basis functions on the geometric triangles. The functions are next input into the `Space` module, which has already been discussed in the previous sections. The support of the basis functions is defined by the computational domain's discretisation which is also input into `Space`. In our case, `Space` uses the quadratic basis function module twice and the linear basis function module once, as the 2D Navier–Stokes problem has two velocity components to be approximated quadratically and one linearly approximated pressure component.

An important advantage of the design of NUMLAB is the orthogonal combination of basis functions and geometric grids. Several other (e.g. higher order) basis function modules are provided as well, defined on different geometric elements. By combining them as inputs to the `Space` module, one can easily define a large range of approximation spaces for various computational problems. In the case of a diffusion PDE solved on a grid of quadrilaterals, for example, one would use a single `SpaceReferenceQuadLinear` basis function input to the `Space` module.

5.1.3 Operators and solvers. The third group contains modules that define the function $\boldsymbol{F}$ for which the equation $\boldsymbol{F}(\boldsymbol{x}) = \boldsymbol{0}$ is to be solved, as well as the solution method to be used. This group contains thus specialisations of the `Operator` and `Solver` modules described in the previous sections.

In our example, the discrete formulation of (26) discussed in the previous section is implemented by the `Operator-ImplementationFiniteElementNavierStokes` module. The Navier–Stokes problem is solved by a Newton solver implemented by the `OperatorIteratorNon-LinearNewtonDamped` module. The linear system output by the Newton module is then solved by a conjugate gradient solver implemented by the `OperatorIterator-LinearCGS` module. The solution is accelerated by using an incomplete LU preconditioner `OperatorIterator-LinearILU` which is passed as input to the conjugate gradient solver.

Other problems can be readily modelled by choosing other operator implementations. Similarly, to use another solution or preconditioning method, a chain of `Solver` specialisations can be constructed. As solvers have an input of the same `Solver` type, complex solution algorithms can be built on the fly.

5.1.4 Functions. The fourth group contains specialisations of the `Function` module. These model both the solution of a numerical problem as well as its initial conditions or other involved quantities such as material properties. In our example, the `FunctionVector` module holds both the velocity and pressure solution of the Navier–Stokes equation. The solution is updated at every iteration, as this module is connected to the solver module's output. As explained in the previous sections, a function is associated with a space. This is seen in the `Function`'s input connection to the `Space` module.

The solution of the problem is initialised by connecting the `FunctionSymbolicBubble` module to the `Func-tionVector`'s input. When the user changes the initial solution value, by changing an input of the `Function-SymbolicBubble` signal or by replacing it with another function, the network restarts the computations from this new value.

5.1.5 Visualisation. As presented in Sect. 1, a computational environment should provide extensive support for data visualisation and monitoring. Such support should cover the following:

- several *dataset representations*, such as structured, unstructured, curvilinear, rectilinear, uniform and locally refined grids, with several types of values defined per node or per cell (scalar, vector, tensor, colour, etc). Support for image datasets should be provided as well. Besides these discrete datasets, the possibility of defining continuous datasets (e.g. implicit functions) should also be taken into account.
- several *dataset processing* tools, such as dataset readers and writers for various data formats, filters producing streamlines, streamribbons, isosurfaces, warp planes, slices, dataset simplifications, feature extraction, and so on. Imaging operations should also be supported, such as image filtering, Fourier transforms, image segmentation, colour processing, etc.
- several *visualisation primitives*, such as 2D and 3D rendering or objects with various shading models, mapping scalars to colours via various colourmaps, direct manipulation of the viewed objects, interactive data probing and object picking, hard copy options, animation creation, and so on.

A second requirement is that the visualisation tools should be open for extension or customisation, as researchers often need to extend, adapt, optimise, or experiment otherwise with various visualisation algorithms and data structures.

Writing such a library is clearly a task out of the scope of a single person. Moreover, such libraries exist, offering various degrees of application domain specificity and numbers of components. In order to provide NUMLAB with the desired visualisation capabilities, we have integrated the Visualization Toolkit (shortly VTK) [44] library into the VISSION environment. VTK is one of the most powerful freely available scientific visualisation libraries, with over 400 components for scalar, vector, and tensor visualisation, imaging, volume rendering, charting, and more. Similarly to NUMLAB, VTK is implemented as a set of C++ classes that specialise a few basic concepts such as datasets, filters, mappers, actors, viewers, and data readers and writers.

Back to the Navier–Stokes simulation network of Fig. 1, we finally discuss the modules that provide visualisation facilities. The main module is the `FunctionVTKViewer` module group which takes as input the current solution of the Navier–Stokes equation and the grid upon which it is defined. In VISSION, a module group represents a whole subnetwork of modules or groups which are treated as a single module. In our example, the `FunctionVTKViewer` module inputs the velocity and pressure solution components into various visualisation modules, such as stream lines and hedgehogs for the vectorial, respectively colour plots and isolines for the scalar component. These modules are accessible to the interested user by double-clicking on the `FunctionVTKViewer` icon.

Several other visualisation methods can be easily attached to the Navier–Stokes simulation, by editing the contents of the `FunctionVTKViewer` module group. Keeping the visualisation back-end pipeline inside a single module group allows a natural separation of the computational network from the post-processing operations. This also helps to reduce the overall network visual complexity.

5.2 Navier–Stokes simulation steering and monitoring

Once the Navier–Stokes computational network is constructed, one can start an interactive simulation by changing the parameters of the various modules involved, such as mesh refinement, solver tolerance, or initial solution value. All the numerical parameters, as well as the parameters of the visualisation back-end are accessible via the module interactors automatically created by VISSION (Fig. 1b).

Moreover, the evolution of the intermediate solutions produced by the Newton solver can be interactively visualised. This is achieved by constructing a loop which connects the output of the `OperatorIteratorNonLinearNewtonDamped` module to its input. The module will then change the `FunctionVector`, and thus the visualisation pipeline downstream of it, at every iteration. This allows one to interactively monitor the improvement of the solution at a given time step, and eventually change other parameters to experiment new solvers or preconditioners.

Figure 2 shows a snapshot from an interactive Navier–Stokes simulation. The simulation domain, shown meshed in Fig. 2a, consists of a 2D rectangular vessel with an inflow and an outflow. The inflow and outflow have both parabolic essential boundary conditions on the fluid velocity. The sharp obstacle placed in the middle of the container can be interactively manipulated by the end user by dragging its tip with the mouse anywhere inside the upper-left vessel picture. Once the obstacle's shape is changed, the NumLab network re-meshes the new domain, recomputes the stationary solution for the Navier–Stokes simulation defined on this new domain, and displays the pressure and velocity solution (Fig. 2). Various other parameters, such as fluid viscosity, mesh refinement, and solver accuracy, can also be interactively controlled. The computational steering of the above problem proceeds at near-interactive rates, for e.g. 2000 elements on an SGI O2 R5000 machine. Consequently, such NumLab setups can be used for quick, interactive testing of the robustness and accuracy of various solvers, preconditioners, and mesh generators. For example, one can test the speed and robustness of an iterative solver for different combinations of obstacle size and shape, mesh coarseness, and fluid viscosity for the above problem. Alternatively, a fine mesh can be used for obtaining accurate solutions.

NumLab can also be used for solving large computational problems. In the following example, glass pressing in the industry is considered. The process of moulding a hot glass blob pressed by a parison is simulated. The glass is modelled as a viscous fluid, subjected to the Navier–Stokes equations. The pressing simulation is a time-dependent process, where the size and shape of the computational domain is changed at every step, after which the stationary Navier–Stokes equations are solved on the new domain. The flow equations can be solved on a two-dimensional cross-section in the glass, since the real 3D domain is axisymmetric.

The simulation is analogous in many respects to the one previously presented. However, a mesh in the glass pressing simulation involves tenths of thousands of finite elements, whereas the previous example used only a few hundreds.

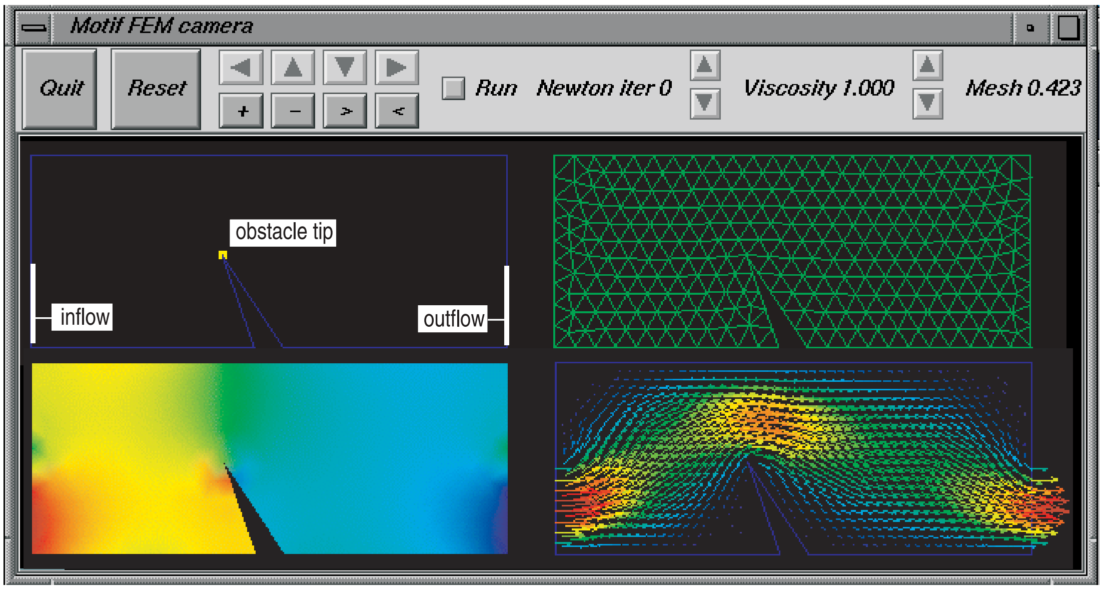


Fig. 2. Interactive Navier–Stokes simulation: domain, mesh, pressure, and velocity solution

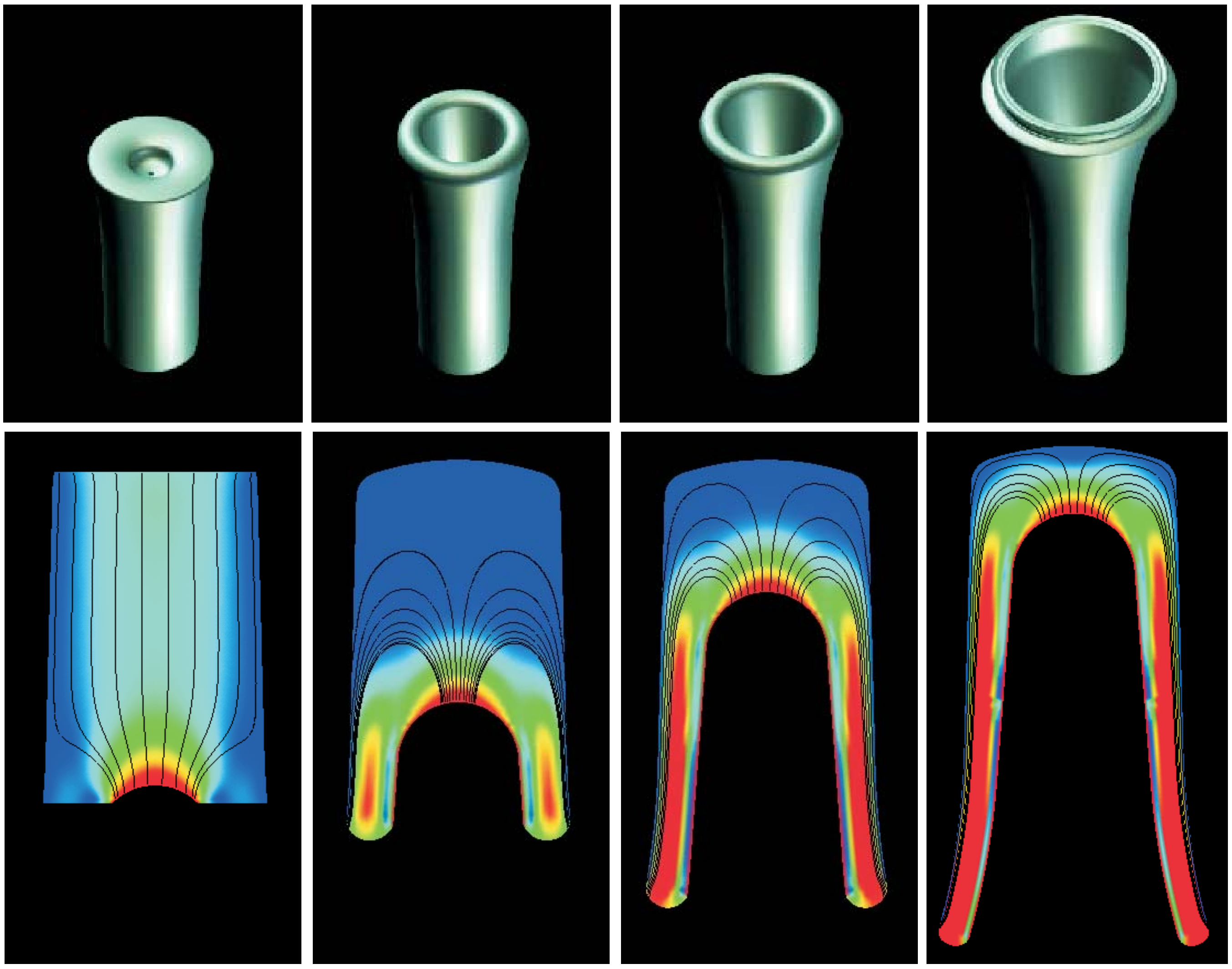


Fig. 3. 3D visualisation of glass pressing (*top row*). Pressure magnitude in 2D cross-section (*bottom row*)

Consequently, the latter simulation can not be steered interactively. However, all computational parameters of the involved NUMLAB network can be interactively controlled at the beginning of the process, or between computation steps. Figure 3 shows several results of the glass pressing simulation. The first row depicts several snapshots of the 3D geometry of the moulded glass, reconstructed and realistically rendered in NUMLAB from the 2D computational domain. The second row in Fig. 3 shows fluid pressure snapshots taken during the 2D numerical simulation. The output of the NUMLAB visualisation pipeline can be connected to the `MPEGCreator` module. In this way, one can produce MPEG movies of the time-dependent simulation which can be visualised outside the VISSION environment as well. For the `MPEGCreator` module, we have actually reused the freely available code of the same module in the AVS system [49], by adding a simple C++ class interface to it. This reuse is typical for the open structure of the NUMLAB-VISSION combination.

The above has presented two computational applications built with the NUMLAB library in the VISSION system. However different in terms of interactivity, computational complexity, and visualisation needs, these applications illustrate well the smooth integration of numerics, user interaction, and on-line visualisation that is achieved by embedding the NUMLAB library in the VISSION environment.

6 Conclusions and future work

The numerical laboratory NUMLAB was designed to address two categories of limitations of current computational environments.

First, NUMLAB addresses the functional limitations of many computational systems by factoring out of a few fundamental mathematical notions: Vector functions $\boldsymbol{x}$, spaces $\boldsymbol{V}$, operators $\boldsymbol{F}$ on such spaces, and implementation of the evaluation $z = \boldsymbol{F}(\boldsymbol{x})$. Based hereon, *all* its (iterative) solution methods, preconditioners, time integrators, finite element/difference/volume operators, etc. are instances of approximate evaluations $z \approx \boldsymbol{F}(\boldsymbol{x})$. Because roots are in general computed with the use of evaluations $z = \boldsymbol{F}(\boldsymbol{x})$ – and sometimes evaluations involving Jacobians – NUMLAB extension are simple. Its objects are close to the modelled mathematics.

Secondly, NUMLAB is easy to extend, customise and simple to use for a large application class. It provides interactive application construction, steering, and visualisation with its

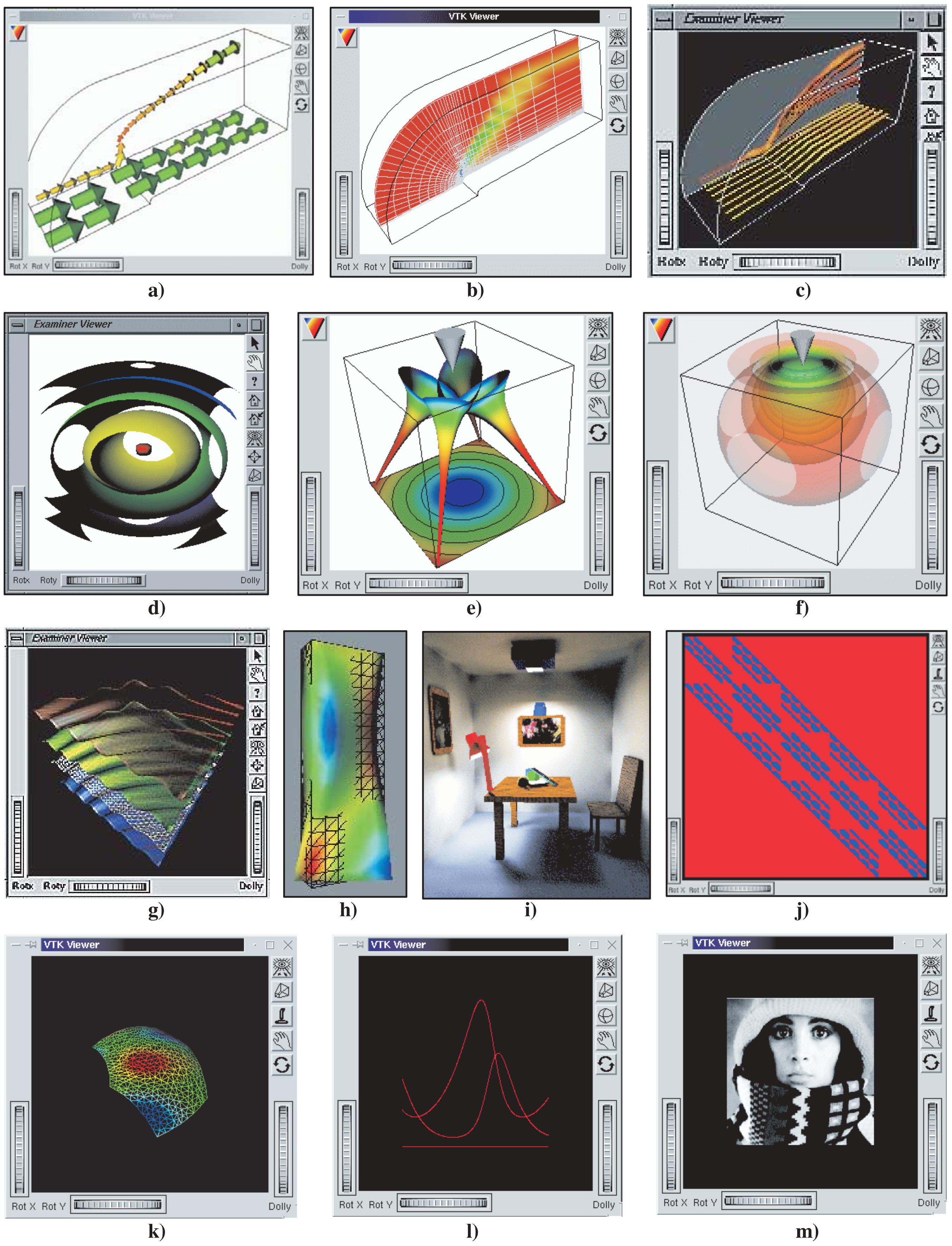


Fig. 4. Visualisation of various numerical computations in the NUMLAB environment

network editor VISSION. In VISSION, but also outside in compiled and interpreted programs, its numerical libraries can be intermixed with other visualisation, data processing, and data interchange libraries.

NUMLAB seperates its numerical libraries, visualisation libraries VTK/OI and interaction and dataflow library VISSION. This makes their extension, maintenance, and understanding much easier than in systems where the above libraries are amalgamated in one (source) code.

The scientific researcher who uses NUMLAB can with ease focus on the testing of new mathematical algorithms and applications. This is shown by the large class of applications implemented in our framework (Fig. 4). Large datasets produced by computational flow dynamics simulations are interactively visualised by for instance arrow plots (Fig. 4a), slices (Fig. 4b), or interactively placed stream tubes (Fig. 4c). Mathematical objects can be computed and visualised, such as scalar functions (the isosurface plot in Fig. 4d or a quadric function), or tensor functions (the hyperstreamline and the isosurface plots on Fig. 4e,f of a stress tensor caused by a point load on a semi-infinite domain). Various simulations have been implemented and interactively run, such as wave simulations (Fig. 4g), elasticity problems, (Fig. 4h), and global illumination using the radiosity method (Fig. 4i), as described in [15, 16].

However flexible, the NUMLAB environment has also a number of conceptual and practical limitations, as follows. Conceptually, the network application model it uses can be sometimes unintuitive for its users. A NUMLAB network (see e.g. Fig. 1) reflects directly the object oriented structure of the underlying C++ library. Understanding this structure and the role its fine-grained components play involves a certain learning curve for the end users. In many cases, the complexity of the networks can be hidden from the end user by the usage of groups (Sect. 5.1.5). Overall, we believe that this extra complexity is a reasonable price to pay for the generic nature of the toolkit.

From the practical viewpoint, solving large PDE problems in NUMLAB is still slower compared to using specialised toolkits. This is due to the generic nature of the NUMLAB modules that can not make assumptions about specific data storage or discretisation properties provided by other modules (see e.g. Sect. 3.3). This problem can be tackled in several ways: implementing less generic (optimised) modules, reengineering the generic modules' implementations to make more extensive use of data caching, or parallelising the numerical code, as outlined further in this section. A second limitation involves the need to program new `Operator` subclasses e.g. to model new PDEs (see Sects. 3.4 and 4.1). A better approach would be to design generic `Operators` that accept their definition via a symbolic, interpreted notation. Implementing such generic `Operators` would raise the same efficiency problems outlined above.

Multigrid solvers can be used within the NumLab framework, but right now, no special support is provided. In order to offer convenient multi-grid solver modules, a few modules need to be extended, and others must be added. The grid and solution data types must be extended to handle internal stacks of grids and solutions. Restriction and prolongation require new modules, and application-specific preconditioners are desirable. The strong coupling between the grid and basic iterative solvers need not be a problem: operator evaluation can be grid based.

We plan to extend the NUMLAB library with even more modules, including readers and writers for the standards MathML and OpenMath [14] Along with this, we plan to integrate a new technique for automatic, transparent parallelisation of all numerical code in NUMLAB.

References

1. Abram, G., Treinish, L.: An Extended Data-Flow Architecture for Data Analysis and Visualization. Proc. IEEE Visualization 1995, ACM Press, pp. 263–270
2. Anderson, E., Bai, Z., Bischof, C. et al.: LAPACK user's guide. SIAM Philadelphia, 1995
3. Arnold, D.N., Brezzi, F., Fortin, M.: A stable finite element for the Stokes equations. Calcolo 21, 337–344 (1984)
4. Astheimer, P.: Sonification tools to supplement dataflow visualization. Scientific Visualization: Advances and Challenges, Academic Press 1994, pp. 251–263
5. Axelsson, O.: A generalized conjugate gradient, least square method. Numerische Mathematik 51, 209–227 (1987)
6. Axelsson, O., Maubach, J.: Global space-time finite element methods for time-dependent convection diffusion problems. Advances in Optimization and Numerical Analysis 275, 165–184 (1994)
7. Axelsson, O., Vassilevski, P.S.: Algebraic multilevel preconditioning methods I. Numerische Mathematik 56, 157–177 (1989)
8. Axelsson, O., Barker, V.A.: Finite Element Solution of Boundary Value Problems. Orlando, Florida: Academic Press 1984
9. Barret, R., Berry, M., Dongarra, J., Pozzo, R.: Templates for the Solution of Linear Systems, 2nd edn. SIAM, 1995
10. Booch, G.: Object-Oriented Analysis and Design. 2nd edn., Redwood City, CA: Benjamin/Cummings 1994
11. Bruaset, A.M., Langtangen, H.P.: A Comprehensive Set of Tools for Solving Partial Differential Equations: Diffpack. Numerical Methods and Software Tools in Industrial Mathematics, Daehlen, M., Tveito, A. (eds.), 1996
12. Budd, T.: An Introduction to Object-Oriented Programming. Addison-Wesley 1997
13. Butcher, J.C.: The numerical analysis of ordinary differential equations: Runge–Kutta and general linear methods. Wiley 1987
14. Caprotti, O., Cohen, A.M.: On the role of OpenMath in interactive mathematical documents. J. Symbolic Comput. 32, 351–364 (2001)
15. Cohen, M.F. Wallace, J.R.: Radiosity and Realistic Image Synthesis. San Diego CA: Academic Press 1993
16. Cohen, M.F., Wallace, J.R., Chen, S.E., Greenberg, D.P.: A Progressive Refinement Approach to Fast Radiosity Image Generation. Computer Graphics (SIGGRAPH '95 Proceedings), Vol. 22, No. 4
17. Crouzeix, M., Raviart, P.A.: Conforming and nonconforming finite element methods for solving the stationary Stokes equations, I. R.A.I.R.O. 3, 33–76 (1973)
18. Dembo, R.S., Eisenstat, S.C., Steihaug, T.: Inexact Newton methods. SIAM Journal on Numerical Analysis 19, 400–408 (1982)
19. Duclos, A.M., Grave, M.: Reference models and formal specification for scientific visualization. Scientific Visualization: Advances and Challenges. Academic Press 1994, pp. 251–263
20. Eisenstat, S.C., Walker, H.F.: Globally convergent inexact Newton methods. SIAM Journal on Optimization 4, 393–422 (1994)
21. Ervin, V., Layton, W., Maubach, J.: A Posteriori error estimators for a two level finite element method for the Navier–Stokes equations. Numerical Methods for PDEs 12, 333–346 (1996)
22. Gear, C.W.: Numerical initial value problems in ordinary differential equations. Prentice-Hall 1971
23. Gunn, C., Ortmann, A., Pinkall, U., Polthier, K., Schwarz, U.: Oorange: A Virtual Laboratory for Experimental Mathematics. Sonderforschungsbereich 288, Technical University Berlin. `http://www-sfb288.math.tu-berlin.de/oorange/OorangeDoc.html`
24. Hackbusch, W., Wittum, G. (eds.): ILU algorithms, theory and applications, Proceedings Kiel 1992. In: Notes on numerical fluid mechanics 41. Vieweg 1993

25. Hackbusch. W., Wittum, G. (eds.): Multigrid Methods, Proceedings of the European Conference in: Lecture notes in computational science and engineering. Springer Verlag 1997
26. IMSL: FORTRAN Subroutines for Mathematical Applications, User's Manual. IMSL 1987
27. Jackie, N., Davis, T., Woo, M.: OpenGL Programming Guide. Addison-Wesley 1993
28. The Java 3D Application Programming Interface: `http://java.sun.com/products/java-media/3D/`
29. John, V., Maubach, J.M., Tobiska, L.: A non-conforming streamline diffusion finite element method for convection diffusion problems. Numerische Mathematik 78, 165–188 (1997)
30. Layton, W., Maubach, J.M., Rabier, P.: Robust methods for highly non-symmetric problems. Contemporary Mathematics 180, 265–270 (1994)
31. Margenov, S.D., Maubach, J.M.: Optimal algebraic multilevel preconditioning for local refinement along a line. Journal of Numerical Linear Algebra with Applications 2, 347–362 (1995)
32. Wolfram, S.: The Mathematica Book 4-th edition. Cambridge University Press 1999
33. Matlab: Matlab Reference Guide. The Math Works Inc. 1992
34. Mattheij, R.M.M., Molenaar, J.: Ordinary differential equations in theory and practice. Wiley 1996
35. Maubach, J.: Local bisection refinement for n-simplicial grids generated by reflections. SIAM Journal on Scientific Computing 16, 210–227 (1995)
36. B. Meyer: Object-oriented software construction. Prentice Hall 1997
37. NAG: FORTRAN Library, Introductory Guide, Mark 14. Numerical Analysis Group Limited and Inc. 1990
38. Parker, S.G., Weinstein, D.M., Johnson, C.R.: The SCIRun computational steering software system. In: Arge, E., Bruaset, A.M., Langtangen, H.P. (eds.), Modern Software Tools for Scientific Computing. Switzerland: Birkhaeuser Verlag AG 1997, pp. 1–40
39. Ribarsky, W., Brown, B., Myerson, T., Feldmann, R., Smith, S., Treinish, L.: Object-oriented, dataflow visualization systems – a paradigm shift?. in Scientific Visualization: Advances and Challenges. Academic Press 1994, pp. 251–263
40. Roos, H.G., Stynes, M., Tobiska, L.: Numerical Methods for Singularly Perturbed Differential Equations. Springer Verlag 1996
41. Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W.: Object-Oriented Modelling and Design. Prentice-Hall 1991
42. Saad, Y., Schultz, M.H.: GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. SIAM Journal on Scientific and Statistical Computing 7, 856–869 (1986)
43. INRIA-Rocquencourt: Scilab Documentation for release 2.4.1., 2000 `http://www-rocq.inria.fr/scilab/doc.html`
44. Schroeder, W., Martin, K., Lorensen, B.: The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics. Prentice Hall 1995
45. Stroustrup, B.: The C++ Programming Manual. Addison-Wesley 1993
46. Telea, A., van Wijk, J.J.: VISSION: An Object Oriented Dataflow System for Simulation and Visualization. Proceedings of IEEE VisSym '99, Groeller, E., Ribarsky, W. (eds.), Springer 1999, pp. 95–104
47. Telea, A.: Combining Object Orientation and Dataflow Modeling in the VISSION Simulation System. Proceedings of TOOLS'99 Europe, Nancy 3–8 June 1999, IEEE Computer Society Press, 1999, Mitchell, R., Wills, A.C., Bosch, J., Meyer, B. (eds.) pp. 56–65
48. Todd, M.J.: The Computation of Fixed Points and Applications. Lecture Notes in Economics and Mathematical Systems 124. Springer Verlag 1976
49. Upson, C., Faulhaber, T., Kamins, D., Laidlaw, D., Schlegel, D., Vroom, J., Gurwitz, R., van Dam, A.: The Application Visualization System: A Computational Environment for Scientific Visualization. IEEE Computer Graphics and Applications, July 1989, 30–42. See also `http://www.avs.com`
50. Wernecke, J.: The Inventor Mentor: Programming Object-Oriented 3D Graphics with Open Inventor. Addison-Wesley 1993