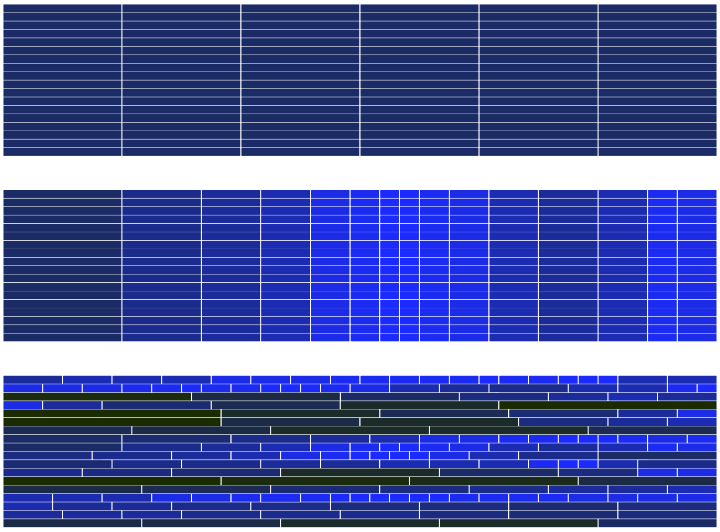
CHALMERS

A Multi-Adaptive ODE-Solver

Anders Logg

Master of Science Thesis in Engineering Physics
Examensarbete för civilingenjörsexamen i teknisk fysik
med inriktning teknisk matematik



Chalmers Finite Element Center
Department of Mathematics
Chalmers University of Technology
Göteborg, Sweden
1998

The front page shows three different discretizations for a multidimensional system of equations. The first one is a non-adaptive discretization, the second one is an adaptive discretization and the third one is a *multi-adaptive* discretization.

This document was generated with LaTeX on Solaris 2.6 at `dd.chalmers.se`. The font is Palatino 10pt.

This report, as well as the code implementing the method proposed in it, are available for download at `http://www.dd.chalmers.se/~f95logg/Tanganyika/`.

Computations have been made with the Tanganyika multi-adaptive ODE-solver library (available for download at `http://www.dd.chalmers.se/~f95logg/Tanganyika/`) on Linux 2.0 (Intel Pentium 200MHz) and on Solaris 2.6 (Sun Ultra 1 Model 170).

Göteborg, October 3, 1998

Abstract

In this work I present a *multi-adaptive* finite element method for initial value problems for ordinary differential equations, including an a posteriori estimate of the error.

The method is multi-adaptive in the sense that the resolution of the time discretization is chosen individually for each component of the system of ordinary differential equations, based on an estimation of the error.

The method has been successfully implemented in the *Tanganyika* library, available for download. Included are a few example computations made with this library, as well as instructions for downloading and using the package.

Acknowledgements

I wish to thank

- **Claes Johnson**, my advisor, for his continuous encouragement and support in the making of this project;
- **Rickard Lind**, **Mathias Brossard** and **Andreas Brinck** for beta-testing the programs;
- **Jim Tilander** for his expertise help with C++;
- **Greger Cronquist** for some useful hints on the typesetting;
- **Göran Christiansson** for proof-reading the manuscript;
- **Anna** for letting me do this all summer.

Contents

CHAPTER 1

Introduction

Numerical methods for solving initial value problems for ordinary differential equations have been around for a long time and the number of methods is almost as large as the number of equations.

Common methods, such as the ones supplied with Matlab (`ode45()`, `ode23()`, `ode113()`, `ode-whatever()`), are often fast, meaning that they terminate in a short time.

These methods often provide some sort of local error control, where the error is controlled in some way in each integration step. This, however does not mean control of the global error. Although a tolerance is specified, it is not related – otherwise than by some (hopefully) monotonically increasing, and otherwise unknown, function – to the global error of the solution. The program is thus not concerned with the actual value of the error, leaving the user unaware of the quality of the computed solution.

> *In fact, it was wrong.*
> Bill Clinton (1998).

Using such a classical numerical solver usually means solving the problem at a number of different tolerance levels for the local error, and comparisons between these solutions. Error control is thus (perhaps) obtained manually.

This manual effort should also be taken into account when comparing the efficiencies of different solvers.

1.1 Quantitative Error Control

Using a posteriori estimates of the error, i.e. error estimates based on the computed solution, it is possible to accurately control the size of the global error.

Finite elements present a general framework for solving differential equations, such as e.g. initial value problems for ordinary differential equations, considered in this report. Depending on the choice of basis functions, normally piecewise polynomials of different kinds, the result is a new step method for solving the initial value problem. These methods include $\mathrm{cG}(1)$, $\mathrm{cG}(2)$,$\ldots$, $\mathrm{dG}(0)$, $\mathrm{dG}(1)$,$\ldots$.

Efficieny is obtained by *adaptivity*, putting the computational effort where it is most needed. For initial value problems this usually means adjusting the size of the timestep, thus choosing the timestep to be small where the solution is especially sensitive to errors in the numerical method.

Proper a posteriori error control requires knowledge of the stability of the problem. Stability properties are in general obtained by solving a so-called dual problem. Thus, error control requires some extra effort from the solver, which in some cases is comparable to the effort of solving the problem itself.

Work on quantitative error-control during the last ten years (see references [1]-[10]) has resulted not only in extensive theoretical results, but also in working implementations of the methods, such as e.g. CARDS (solver of initial value problems for ordinary differential equations – see [8]) and FEMLAB (solver of partial differential equations).

The current approach to quantitative error control was originated with the article by Johnson ([9]) in 1988, discussing error estimation for the $\mathrm{dG}(0)$ and $\mathrm{dG}(1)$ methods. Error estimation for these methods are further discussed by Estep in [5]. The $\mathrm{cG}(q)$ method, which is the basis for the multi-adaptive method presented in this report, is discussed at length in [7]. A more classical approach to error analysis can be found in [11].

A comprehensive and major article on adaptive methods for differential equations is [3]. A general and non-technical discussion on error control and adaptivity is [6].

1.2 Multi-Adaptivity

It is desirable, in short, that in things which do not primarily concern others, individuality should assert itself.
John Stuart Mill, On Liberty (1909).

If we view a system of ODE:s as the representation of a mechanical system and notice that different parts, components, of such a system may behave very differently – some parts oscillating very rapidly and others slowly, perhaps undergoing even uniform motion – we realize that different components of an ODE-system may be differently sensitive to the resolution of the discretization. There is obviously a need for multi-adaptivity, *allowing individual components of an ODE-system to use individual timesteps*.

Normally, the same timestep is used for all components of an ODE-system. The novelty of multi-adaptivity is thus allowing individual adaption of the timesteps for the different components.

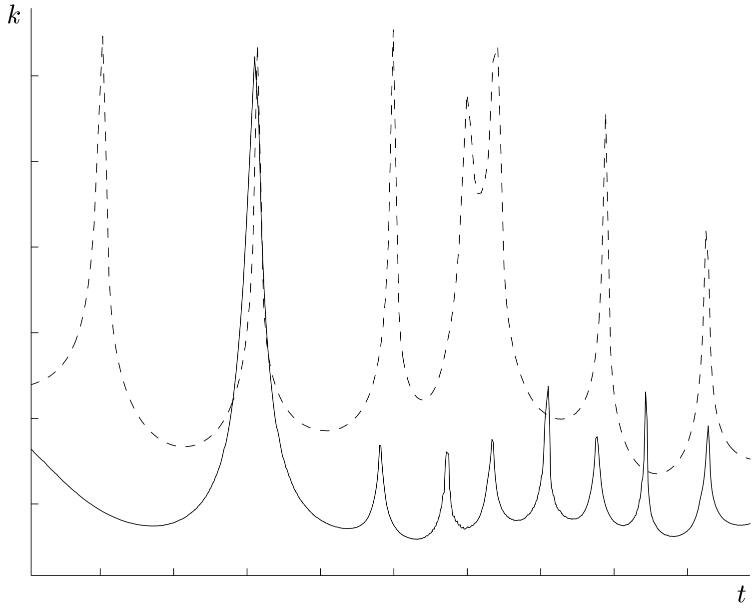


Figure 1.1: These are the actual timesteps used for an example computation on a simple two-dimensional system.

CHAPTER 2

The Method – Multi-Adaptive Galerkin

This chapter describes the multi-adaptive method, complete with an a posteriori error estimate.

The basis for the multi-adaptive method is a generalization of the continuous Galerkin method, $\mathrm{cG}(q)$, described in e.g. [4].

2.1 Equation

The equation to be solved is

$$\begin{cases} \frac{du}{dt}(t) &= f(u,t),\ t \in (0,T], \\ u(0) &= u_0, \end{cases} \tag{2.1}$$

where $f = (f_1, \ldots, f_N)$ is some function[1]depending on the solution $u = (u_1, \ldots, u_N)$ and t, which may represent time.

[1]In order to guarantee the existence of a unique solution, it may be good to know that f is Lipschitz continuous.

2.2 Finite Element Formulation

The weak (variational) formulation of equation (2.1) reads

Find $u = u(t)$ such that $u(0) = u_0$ and

$$\int_0^T (\dot{u}, v) = \int_0^T (f, v) \quad \text{for all test functions } v, \tag{2.2}$$

where $(\cdot,\cdot)$ denotes the usual l^2-inner product.

To define the multi-adaptive $\mathrm{cG}(q)$ method, we introduce the *trial space*, V_k^N, and the *test space*, W_k^N, of functions on $[0, T]$, where

$$\begin{aligned} V_k^N &= \{v : v_i \in \mathcal{P}^{q_i}(I_{ij}), j = 1, \ldots, M_i,\ v_i \text{ is continuous},\ i = 1, \ldots, N\}; \\ W_k^N &= \{v : v_i \in \mathcal{P}^{q_i-1}(I_{ij}), j = 1, \ldots, M_i, i = 1, \ldots, N\}. \end{aligned}$$

Thus $v \in V_k^N$ means that all its components v_i are continuous and piecewise polynomial on the intervals $\{I_{ij}\}_{j=1}^{M_i}$, and $v \in W_k^N$ means that all its components v_i are in general discontinuous and piecewise polynomial (of one degree less) on the same intervals as the corresponding trial function.

The multi-adaptive $\mathrm{cG}(q)$ method is then

Find $U \in V_k^N$ such that $U(0) = u_0$ and

$$\int_0^T (\dot{U}, v) = \int_0^T (f, v) \quad \forall v \in W_k^N. \tag{2.3}$$

The discontinuity of the test functions means we may rewrite this as

Find $\{\xi_{ijk}\}_{k=0}^{q_i}$ such that

$$\int_{I_{ij}} \dot{U}_i v = \int_{I_{ij}} f_i v \quad \forall v \in \mathcal{P}^{q_i-1}(I_{ij}),\ \ j = 1, \ldots, M_i, i = 1, \ldots, N, \tag{2.4}$$

$U(0) = u_0$ and U is continuous,

where the $\{\xi_{ijk}\}$ are the parameters determining the piecewise polynomials $\{U_i\}$. Note that there are $(q+1)$ parameters determining a polynomial of degree q, so the index k is from zero to q.

Finding the parameters $\{\xi_{ijk}\}$ in agreement with eq. (2.4) yields the desired solution. What remains is to find the proper discretization, $\{I_{ij}\}$, i.e. the *timesteps* $\{k_{ij}\}$. To choose the timesteps, we need an error estimate, which will be the basis for adaptivity. By means of this error estimate, the discretization will be chosen in a way to give a resulting final error smaller than the specified tolerance.

2.2.1 Details

The parameters $\{\xi_{ijk}\}$ may e.g. be the nodal values for a subdivision of the intervals into q_i subintervals. For an interval I_{ij}, let the nodal points of an equipartition of this interval be $\{t_{ijk}\}_{k=0}^{q_i}$. The corresponding nodal (Lagrange) basis functions, $\{\lambda_{ijk} : \mathbf{R} \to \mathbf{R}\}$, are then defined on I_{ij} for $k = 0, \dots, q_i$, by

$$\lambda_{ijk}(t) = \frac{(t - t_{ij0}) \cdots (t - t_{ij,k-1})(t - t_{ij,k+1}) \cdots (t - t_{ijq_i})}{(t_{ijk} - t_{ij0}) \cdots (t_{ijk} - t_{ij,k-1})(t_{ijk} - t_{ij,k+1}) \cdots (t_{ijk} - t_{ijq_i})}. \tag{2.5}$$

On the interval I_{ij}, U_i may then be written (uniquely) as

$$U_i = \sum_{k=1}^{q_i} \xi_{ijk} \lambda_{ijk}, \tag{2.6}$$

for some values $\{\xi_{ijk}\}$.

Inserting this into eq. (2.4), computing a few integrals (simple but tedious) and solving the resulting system of linear algebraic equations, yields

$$\left\{ \begin{array}{rcl} \xi_{ij1} & = & \xi_{ij0} + \int_{ij} w_{q_i 1}(\tau_{ij}(t)) f_i(U,t)dt, \\ \xi_{ij2} & = & \xi_{ij0} + \int_{ij} w_{q_i 2}(\tau_{ij}(t)) f_i(U,t)dt, \\ \vdots & & \\ \xi_{ijq_i} & = & \xi_{ij0} + \int_{ij} w_{q_i q_i}(\tau_{ij}(t)) f_i(U,t)dt, \end{array} \right. \tag{2.7}$$

where $\tau_{ij}(t) = \frac{t - t_{i,j-1}}{t_{ij} - t_{i,j-1}}$ and the $\{w_{qk}\}_{k=1}^{q}$ are polynomial weight functions. These are given in table 2.1 for $q = 1, 2, 3$.

$w_{11}(\tau) = 1$		
$w_{21}(\tau) = \frac{1}{4}(5 - 6\tau)$	$w_{22}(\tau) = 1$	
$w_{31}(\tau) = \frac{1}{27}(37 - 96\tau + 60\tau^2)$	$w_{32}(\tau) = \frac{1}{27}(26 + 24\tau - 60\tau^2)$	$w_{33}(\tau) = 1$

Table 2.1: Weight functions for the cG(q) integrals, $q = 1, 2, 3$.

2.2.2 Even more Flexibility

Note that we could have allowed each component to be piecewise polynomial without beforehand fixing the deqree of the polynomial on the whole of the discretization. We could thus have allowed the polynomial degree to change from one interval to the next. The method would then be even p-adaptive, choosing the (in some sense) best degree of the polynomials for every single interval I_{ij}.

For simplicity, though, the polynomial degrees have been chosen to be $\{q_i\}$ rather than $\{q_{ij}\}$. The difference would be an extra index j on q.

2.3 Error Estimation

The error estimate is obtained starting the same way as in references [1], [4] and [10].

To estimate the error at final time T in the l^2-norm, the *dual* problem of eq. (2.1) is introduced. The dual problem is

$$\left\{ \begin{array}{rcl} -\frac{d\varphi}{dt}(t) & = & J^*(u,U,t)\varphi(t),\ t \in [0,T), \\ \varphi(T) & = & e(T)/||e(T)||, \end{array} \right. \tag{2.8}$$

where $e = U - u$ is the error, $||\cdot||$ is the l^2-norm and J^* is defined as

$$J^*(u,U,\cdot) = \left(\int_0^1 \frac{\partial f}{\partial u}(su + (1-s)U,\cdot)ds \right)^*, \tag{2.9}$$

i.e. J^* is the transpose (or more generally, the adjoint) of the Jacobian of f at a mean value of u and U.

Note now that by the chain rule,

$$\begin{array}{rcl} -J(u,U,\cdot)(U-u) & = & \int_0^1 \frac{\partial f}{\partial u}(su+(1-s)U,\cdot)ds(u-U) \\ & = & \int_0^1 \frac{\partial f}{\partial s}(su+(1-s)U,\cdot)ds \\ & = & f(u,\cdot) - f(U,\cdot). \end{array} \tag{2.10}$$

We may thus write

$$
\begin{array}{rcl}
||e(T)|| & = & (e(T), e(T))/||e(T)|| \\
& = & (e(T), \varphi(T)) \\
& = & (e(T), \varphi(T)) - (e(0), \varphi(0)) + \int_0^T (e, -\dot{\varphi} - J^*(u, U, \cdot)\varphi) \\
& = & [(e(t), \varphi(t))]_0^T - \int_0^T (e, \dot{\varphi}) - \int_0^T (e, J^*(u, U, \cdot)\varphi) \\
& = & \int_0^T (\dot{e}, \varphi) - \int_0^T (J(u, U, \cdot)e, \varphi) \\
& = & \int_0^T (\dot{e} - J(u, U, \cdot)e, \varphi) \\
& = & \int_0^T (\dot{U} - f(u, \cdot) - J(u, U, \cdot)(U - u), \varphi) \\
& = & \int_0^T (\dot{U} - f(U, \cdot), \varphi) \\
& = & -\int_0^T (R, \varphi),
\end{array}
\tag{2.11}
$$

where R is the residual, i.e.

$$
R = f(U, \cdot) - \dot{U}. \tag{2.12}
$$

Using the finite element formulation for $\overline{\varphi} \in W_k^N$, we continue to get

$$
\begin{array}{rcl}
||e(T)|| & = & -\int_0^T (R, \varphi - \overline{\varphi}) \\
& = & -\sum_{i=1}^N \int_0^T R_i(\varphi_i - \overline{\varphi}_i) \\
& = & -\sum_{i=1}^N \sum_{j=1}^{M_i} \int_{I_{ij}} R_i(\varphi_i - \overline{\varphi}_i) \\
& \leq & \sum_{i=1}^N \sum_{j=1}^{M_i} \sup_{I_{ij}} |R_i| \int_{I_{ij}} |\varphi_i - \overline{\varphi}_i| \\
& \leq & \sum_{i=1}^N \sum_{j=1}^{M_i} C_{q_i} k_{ij}^{q_i} \sup_{I_{ij}} |R_i| \int_{I_{ij}} |\varphi_i^{(q_i)}|,
\end{array}
\tag{2.13}
$$

where the $\{C_{q_i}\}$ are constants.

2.3.1 The Constant C_q

Choosing the test function $\overline{\varphi}$ as the $(q-1)$:th-order Taylor-expansion of φ around $(t_{ij} + t_{i,j-1})/2$ on I_{ij} yields

$$
C_q = \frac{1}{q! 2^{q-1}}. \tag{2.14}
$$

The proof is simple. Noting that, with

$$
\overline{f}_{q-1}(x) = f(x_0) + f'(x_0)(x - x_0) + \ldots + \frac{1}{(q-1)!} f^{(q-1)}(x_0)(x - x_0)^{(q-1)} \tag{2.15}
$$

we have

$$|f(x) - \overline{f}_{q-1}(x)| = |\frac{1}{(q-1)!}\int_{x_0}^{x} f^{(q)}(y)(y-x_0)^{(q-1)}dy| \tag{2.16}$$

and thus, with $x_0 = (a+b)/2$,

$$\begin{array}{rcl} \int_a^b |f - \overline{f}_{q-1}| & = & \frac{1}{(q-1)!}\int_a^b |\int_{x_0}^x f^{(q)}(y)(y-x_0)^{(q-1)}dy|dx \\ & \leq & \frac{1}{(q-1)!}\left(\int_a^b |x-x_0|^{q-1}dx\right)\left(\int_a^b |f^{(q)}(x)|dx\right) \\ & = & \frac{1}{q!}\left(|a-x_0|^q + |b-x_0|^q\right)\int_a^b |f^{(q)}| \\ & = & \frac{2(b-a)^q}{q!2^q}\int_a^b |f^{(q)}| \\ & = & \frac{(b-a)^q}{q!2^{q-1}}\int_a^b |f^{(q)}|. \end{array} \tag{2.17}$$

Another useful estimate (see the section on adaptivity below) is

$$\int_a^b |f - \overline{f}_{q-1}| \leq \tilde{C}_q (b-a)^{q+1} \sup_{I_{ij}} |f^{(q)}|, \tag{2.18}$$

where

$$\tilde{C}_q = \frac{1}{(q+1)!2^q}, \tag{2.19}$$

which is obtained as above, choosing $\overline{f}_{q-1}$ to be the $(q-1)$:th-order Taylor expansion around the midpoint.

2.3.2 A Correction of the Error Estimate

The method to be used is not, because of the difficulty involved with solving eq. (2.7), the true multi-adaptive $\mathrm{cG}(q)$ method, as will be described further in chapter 3.

Not solving the equations properly will introduce the *discrete residual*, which should be zero if the discrete equations, i.e. (2.7), were solved properly. The following analysis will result in an extra term in the error estimate (2.13), including the discrete residual together with its proper stability factor, accounting for accumulation of errors due to a non-zero discrete residual.

Defining the discrete residual to be

$$\overline{R}_{ij} = \overline{R}_i(t) = \int_{I_{ij}} f_i - (\xi_{ijq} - \xi_{ij0}),\ j = 1, \ldots, M_i,\ i = 1, \ldots, N, \tag{2.20}$$

we get for $\overline{\varphi}_i \in \mathcal{P}^{q_i-1}(I_{ij})$ and some $\eta_{ij} \in I_{ij}$,

$$\int_{I_{ij}} R_i\overline{\varphi}_i = \overline{\varphi}_i(\eta_{ij}) \int_{I_{ij}} (\dot{U}_i - f_i) = -\overline{\varphi}_i(\eta_{ij})\overline{R}_{ij}. \tag{2.21}$$

Thus, $\int_{I_{ij}} R_i\overline{\varphi}_i$ differs from zero and we get an additional term in our error estimate, continuing from eq (2.11):

$$\begin{array}{rcl} ||e(T)|| & = & -\int_0^T (R,\varphi) \\ & = & -\sum_{i=1}^N \int_0^T R_i\varphi_i \\ & = & -\sum_{i=1}^N \sum_{j=1}^{M_i} \left[\int_{I_{ij}} R_i\varphi_i - \int_{I_{ij}} R_i\overline{\varphi}_i - \overline{\varphi}_i(\eta_{ij})\overline{R}_{ij}\right] \\ & = & -\sum_{i=1}^N \sum_{j=1}^{M_i} \left[\int_{I_{ij}} R_i(\varphi_i - \overline{\varphi}_i) - \overline{\varphi}_i(\eta_{ij})\overline{R}_{ij}\right] \\ & \leq & \sum_{i=1}^N \sum_{j=1}^{M_i} \left[C_{q_i} k_{ij}^{q_i} \sup_{I_{ij}} |R_i| \int_{I_{ij}} |\varphi_i^{(q_i)}| + |\overline{R}_{ij}| \sup_{I_{ij}} |\overline{\varphi}_i|\right] \\ & \approx & \sum_{i=1}^N \sum_{j=1}^{M_i} \left[C_{q_i} k_{ij}^{q_i} \sup_{I_{ij}} |R_i| \int_{I_{ij}} |\varphi_i^{(q_i)}| + |\overline{R}_{ij}| \sup_{I_{ij}} |\varphi_i|\right], \end{array} \tag{2.22}$$

if we choose $\overline{\varphi}$ close to φ.

2.3.3 Other Error Contributions

Other error contributions that are not dealt with here are quadrature errors and numerical errors due the finite precision arithmetic.

2.4 Adaptivity

Introducing the *stability function*, defined by

$$s_i(t) = s_{ij} = \sup_{I_{ij}} |\varphi_i^{(q_i)}|,\ t \in I_{ij},\ j = 1, \ldots, M_i,\ i = 1, \ldots, N \tag{2.23}$$

and the *stability factor*, defined by

$$S_i(T) = \int_0^T |\varphi_i^{(q_i)}|, \; i = 1, \ldots, N, \tag{2.24}$$

the error estimate (2.13) may be written in two alternative ways as

$$\begin{array}{lll} ||e(T)|| & \leq & \sum_{i=1}^{N} \sum_{j=1}^{M_i} \tilde{C}_{q_i} s_{ij} k_{ij}^{q_i+1} \sup_{I_{ij}} |R_i|, \\ ||e(T)|| & \leq & \sum_{i=1}^{N} C_{q_i} S_i \sup_{(0,T)} (k_i^{q_i} |R_i|). \end{array} \tag{2.25}$$

The stability properties are obtained by numerical approximation (by the multi-adaptive $\mathrm{cG}(q)$ method) of the solution of the dual problem.

Notice that the error contribution from the non-zero discrete residual is not included in these expressions, since I have chosen to base the adaptivity on the Galerkin discretizational error alone. However, the contribution from the non-zero discrete residual is of course included in the computation of the error estimate and thus, indirectly, also in the adaptive procedure.

Adaptivity is then based on the expression

$$||e(T)|| \leq \text{error estimate} = \text{TOL}, \tag{2.26}$$

where TOL is a given tolerance for the error of the solution at time $t = T$.

The discretization is now chosen by *equidistribution* of the error, both onto the different components and onto the different intervals, i.e.

$$\tilde{C}_{q_i} s_{ij} k_{ij}^{q_i+1} \sup_{I_{ij}} |R_i| = \frac{\text{TOL}}{NMi}. \tag{2.27}$$

Alternatively, we may whish to do

$$C_{q_i} S_i \sup_{(0,T)} (k_i^{q_i} |R_i|) = \frac{\text{TOL}}{N}. \tag{2.28}$$

Knowing thus the residuals and the stability functions (or factors) we may choose the proper timesteps. This is done in a way that is iterative in two respects. Firstly, the timestep for an interval is chosen based on the residual in the previous interval. Secondly, the $\{M_i\}$, are not known until the end of the computation. The values $\{M_i\}$ are then a more or less clever guess based on a previous computation. Of course, having computed the solution, we don't have to guess these values to compute an error estimate.

2.4.1 Moderating the Choice of Timesteps

Choosing timesteps as described in the previous section without any extra moderation may cause problems. If the residual in one interval is small, the timestep of the next interval will be large. A large timestep will (often) result in a large residual, which in turn in the same way means the timestep of the next interval will be small. There is thus a chance the timestep will oscillate if it is only based on the residual of the last interval. What needs to be done is to make sure the timesteps (and thus also the residuals) don't differ too much between adjacent intervals. This may be done in a lot of different ways, e.g. by choosing the (harmonic) mean of the previous timestep and the value of the new timestep, as based on the residual. (The Tanganyika library uses a somewhat more sophisticated moderation of the timesteps.)

2.4.2 Choosing Data for the Dual Problem

According to eq. (2.8), we need to know the true error in order to solve the dual problem. If we indeed knew the true error, we would not have to bother with any of this, and since the true error is unknown, we have to make a clever guess. We now discover another benefit of multi-adaptivity – it makes it easier for us to estimate the data for the dual problem! Since we equidistribute the error onto the different components, an estimation of the proper data for the dual problem should be $\pm 1/\sqrt{N}$, N being the dimension, for the different components. The signs for the different components may be obtained by solving at different tolerance levels.

Since, however, we don't know the stability properties of the problem until the computation is done, we cannot expect the errors of an initial computation to be fully equidistributed onto the different components. Hence, we cannot expect $\pm 1/\sqrt{N}$ to always work as data for the components of the dual problem. Again, proper data is obtained by e.g. solving at different tolerance levels.

CHAPTER 3

The Implementation – Tanganyika

This section describes the actual implementation of the method described in the previous section.

3.1 Individual Stepping

The individual stepping is done according to eq. (2.7). This requires knowledge about U, including the values of all other components. These values are evaluated by interpolation (or *extra*polation), according to the order of the method, of the nearest known values of the other components. The solution of the integral equation is done iteratively for every component.

The order of the stepping follows one simple principle;

the last component steps first.

It is the fact that the equations are not solved simultaneously that results in non-zero discrete residuals.

3.1.1 Organization, Book-Keeping

Doing the stepping individually rather than stepping all components together requires some book-keeping, keeping track of the positions of all components and which one is to step next.

The individual stepping is done according to figure 3.1 below. The implementation pretty much follows this scetch.

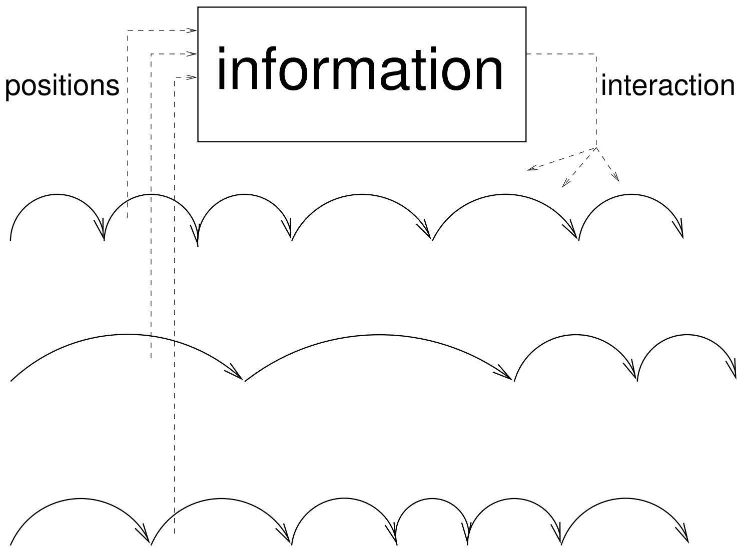


Figure 3.1: This is how the individual stepping is done. The different components tell/send their respective positions and in turn they get their interactions with (forces from) the other components. Thus, just as in nature itself, progress is made by the exchange of information, small pieces of information (gravitons or perhaps *femions*).

3.2 Quadrature

The integrals of eq. (2.7) are evaluated by Gaussian (Gauss-Legendre) quadrature.

Since the order of the weight functions for the integrals of a $\mathrm{cG}(q)$ method are $(q-1)$, we expect the total order of the integrands to be of order $q+(q-1) =$

$2q - 1$ (and even more if f is of quadratic or higher order). It would thus be wise to use quadrature that is exact at least for polynomials of order $2q - 1$, which is exactly the case for Gaussian quadrature with q nodal points.

Thus, midpoint quadrature for $\mathrm{cG}(1)$, two-point Gaussian quadrature for $\mathrm{cG}(2)$ and so on.

3.3 The Program

The method has been implemented as a library, called *Tanganyika*. To use the library functions, all one needs to do is to

```
#include <tanganyika.h>
```

in one's C/C++ program. For more details, refer to the *Tanganyika User Manual* included in Appendix B. For *even more* details (all!) download the source code – see chapter 6.

3.3.1 Language

The language of the Tanganyika library is C++, although its interface is pure C. An object-oriented programming language such as C++ is obviously well-suited for such a program like the Tanganyika library, viewing the different objects as classes; Solution, Component, Element, etc.

3.3.2 Modularity

A nice feature of the C++ programming language is the use of class derivation and inheritance, enabling a modular implementation of the different methods. Implemented in the current version (1.0) of the library are $\mathrm{cG}(1)$, $\mathrm{cG}(2)$ and $\mathrm{cG}(3)$, but the implementation of another method, such as e.g. $\mathrm{dG}(0)$, would require only the implementation of a new subclass, specifying only what differs from the already existing methods. (This would in reality mean perhaps 50 lines of code.)

CHAPTER 4

Results

In this chapter I present the results from a few computations made with the Tanganyika library.

4.1 A First Simple Example

As a first simple example, consider the following system of equations:

$$\begin{cases} \dot{u_1} &= u_2, \\ \dot{u_2} &= -u_1, \text{ in } (0,T] \\ u(0) &= (0,1). \end{cases} \tag{4.1}$$

The solution is of course $u(t) = (\sin(t), \cos(t))$. The equations are solved by the multi-adaptive $\mathrm{cG}(1)$ method with tolerance $8 \cdot 10^{-4}$ and $T = 50$. (The tolerance was actually chosen to be $.001$. The resulting error estimate was, however, $8 \cdot 10^{-4}$.) The true error is, according to figure 4.1, $6.8 \cdot 10^{-4}$ and the component errors are $5.3 \cdot 10^{-4}$ and $4.2 \cdot 10^{-4}$ respectively.

Note the behaviour of the multi-adaptive method, choosing different timesteps for the two methods. The timesteps are chosen on basis of the residuals and stability functions. These are shown, together with the resulting timesteps, in figure 4.1. Note also the approximate equidistribution of the error.

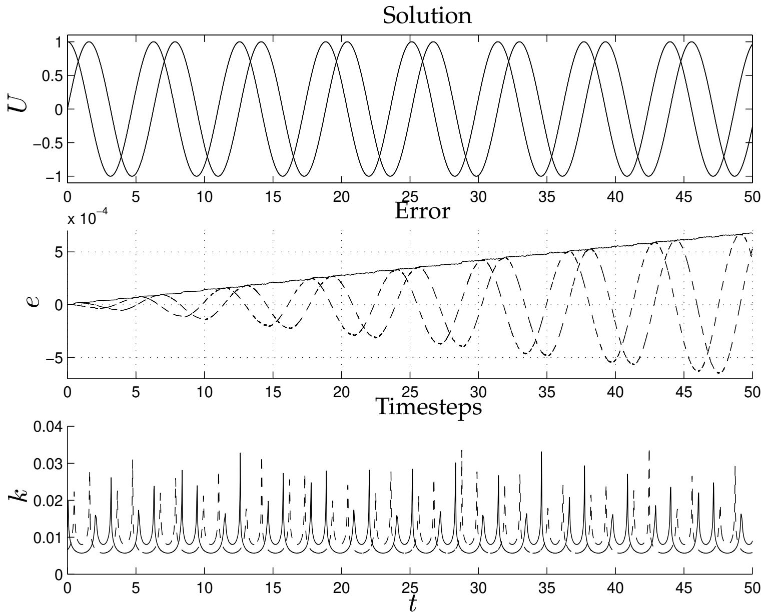


Figure 4.1: The solution of the simple harmonic oscillator problem, the errors and the timesteps respectively.

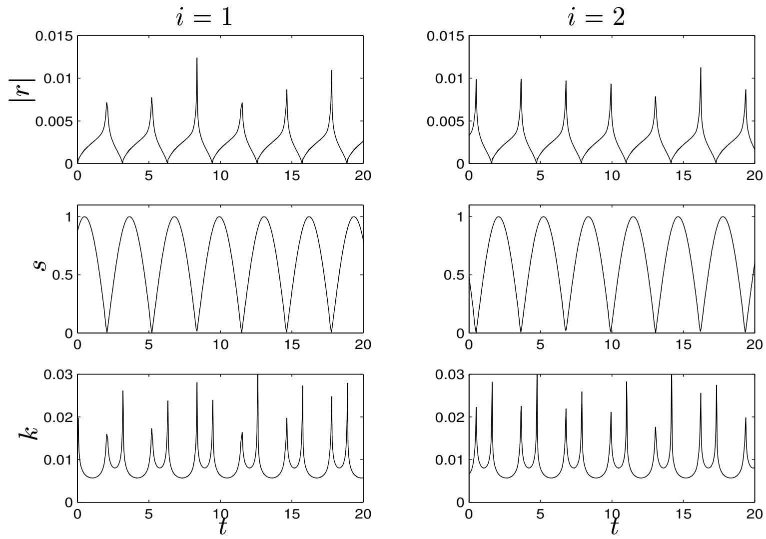


Figure 4.2: Residuals, stability functions and timesteps for the two components of the harmonic oscillator problem, shown for the interval $(0, 20)$.

4.2 Wave Propagation in an Elastic Medium

As a second example, consider wave propagation in an elastic medium, represented by a number of masses connected with springs according to figure 4.3.

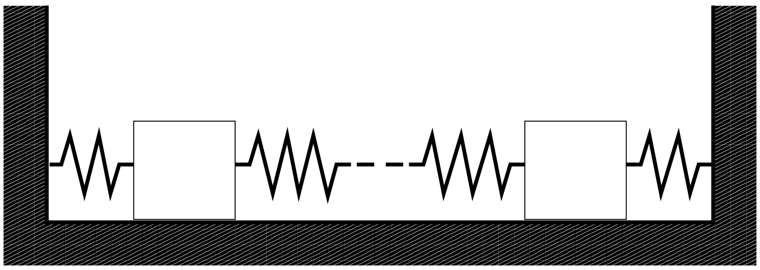


Figure 4.3: A system of N masses and $N + 1$ springs.

The proper equations are easily obtained from Newton's second law of motion.

$$\left\{\begin{array}{rcl} \ddot{x} & = & Ax, \text{ where} \\ A & = & \begin{bmatrix} -2 & 1 & & & & \\ 1 & -2 & 1 & & 0 & \\ & 1 & -2 & 1 & & \\ & & 1 & -2 & \ddots & \\ & 0 & & \ddots & \ddots & 1 \\ & & & & 1 & -2 \end{bmatrix} \end{array}\right. \tag{4.2}$$

This may also be thought of as a FEM space discretization of the wave equation.

With initial conditions corresponding to all but one masses being at rest at $t = 0$, we expect a propagation of the timesteps. At the beginning all but one mass are at rest, so the timesteps for these masses may be large. As the oscillations of a mass increase, the corresponding timesteps should decrease and oscillate. This is also the case according to figure 4.4.

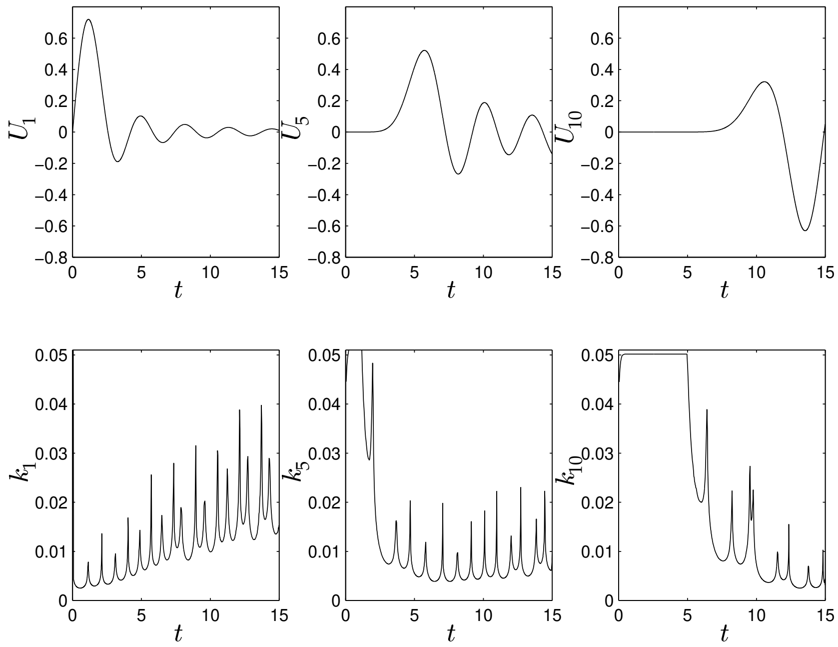


Figure 4.4: Solutions for components 1,5 and 10 of a system consisting of 10 masses and 11 springs, together with their respective timesteps, solved at $\mathrm{TOL} = 5 \cdot 10^{-4}$ with the multi-adaptive $\mathrm{cG}(1)$ method.

4.3 Gravitation

As a third example, consider a system of three bodies (planets) in a somewhat complicated situation where one of the planets is in orbit around a larger one, and a third even smaller planet comes in making sort of a weird sling-shot around the smaller planet.

The forces involved are $1/r^2$ and for a certain choice of initial conditions, the solution is as depicted in figure 4.5 below for $\mathrm{TOL} = .001$, solved with the multi-adaptive $\mathrm{cG}(2)$ method.

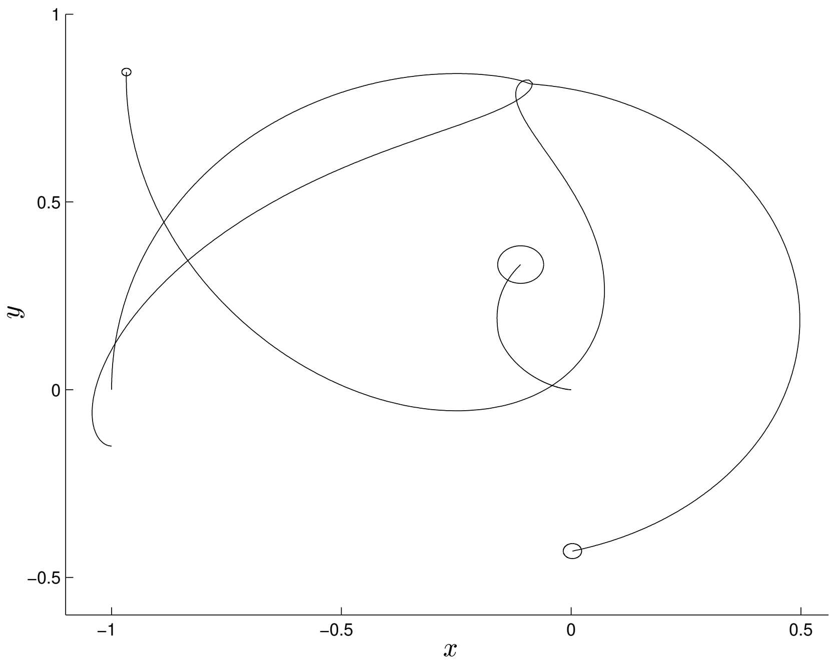


Figure 4.5: Orbits for the three planets. The circles drawn represent the planets at time $t = T$.

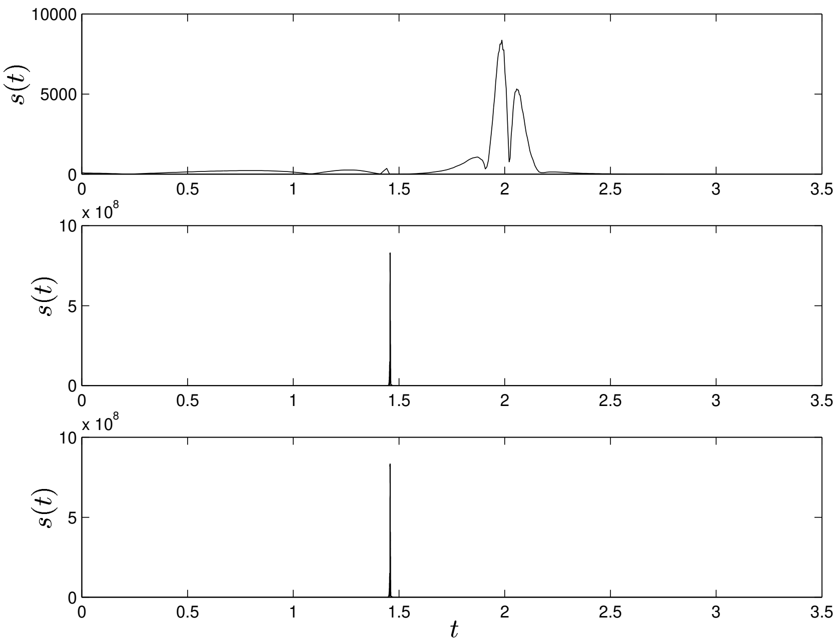


Figure 4.6: Stability functions for the x-components of the three planets.

As one might expect, the three bodies are differently sensitive to the resolution of the discretization. This is also evident in figure 4.7, where are drawn the timesteps for the components corresponding to the x-coordinates of the three planets. (The problem is in two dimensions so there is a total number of 12 components.) In this figure are also the number of timesteps used for the different components. The larger planet, corresponding to components 1,2,7 and 8, obviously doesn't require as many steps as the two smaller ones. The largest number of steps is, according to this figure, needed to resolve the y-velocities of the smallest planet, which is not too strange, considering the main acceleration is in the y-direction at the critical point.

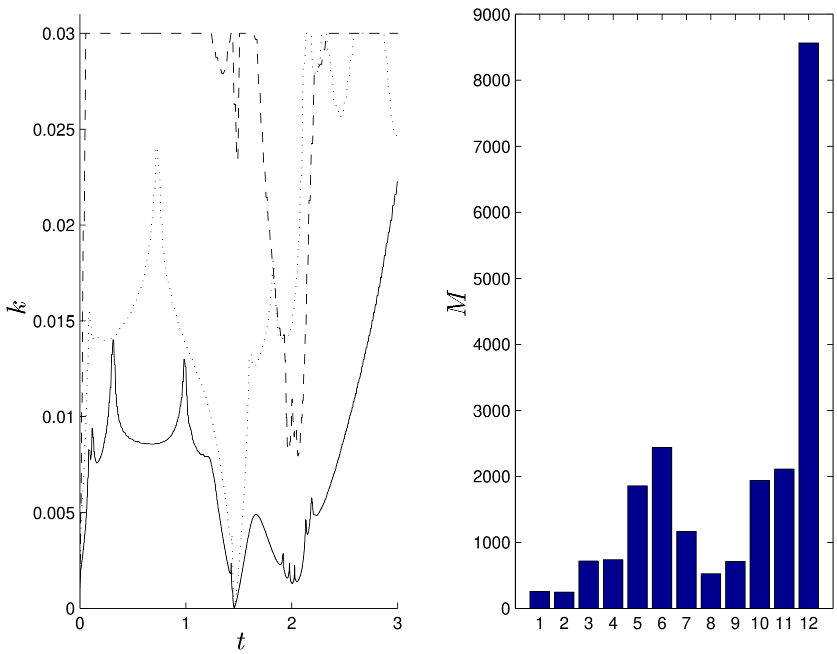


Figure 4.7: Timesteps (left) and the number of timesteps (right) for the 12 different components of the three-body problem.

It is obviously crucial for the timesteps (of the involved components) to be small just when the smallest planet makes the sling-shot. This is realized in the adaptive algorithm by an extremely large value of the stability functions for the involved components, as was shown in figure 4.6.

4.4 The Lorenz System

As a fourth and final example, consider the Lorenz system given by the equations

$$\begin{cases} \dot{x} &= \sigma(y-x), & t \in (0,T], \\ \dot{y} &= rx - y - xz, & t \in (0,T], \\ \dot{z} &= xy - bz, & t \in (0,T], \\ \multicolumn{3}{l}{x(0) = x_0, y(0) = y_0, z(0) = z_0,} \end{cases} \tag{4.3}$$

where $\sigma = 10$, $b = 8/3$ and $r = 28$, and $(x_0, y_0, z_0) = (1, 0, 0)$.

The solution at $\mathrm{TOL} = 2.5 \cdot 10^{-5}$ and $T = 10$ is shown in figure 4.8, together with the timesteps used for the computation. The "chaotic", flipping, behaviour of the Lorenz system is not evident in this figure, since T is too small. The purpose of this example is however not to illustrate certain characteristics of the Lorenz system, but to illustrate the use of multi-adaptivity for the three components.

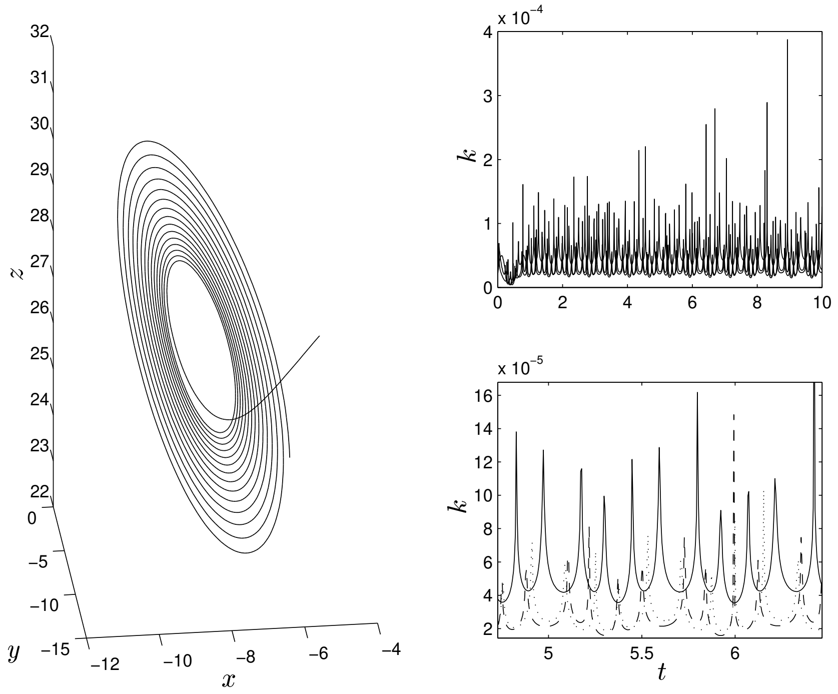


Figure 4.8: At the left is the solution of the Lorenz system, solved with the mulitadaptive $\mathrm{cG}(1)$ method at $\mathrm{TOL} = 2.5 \cdot 10^{-5}$ and with final time $T = 10$. At the right are the timesteps used for the computation.

Below in figure 4.9 is given the behaviour of one of the stability functions.

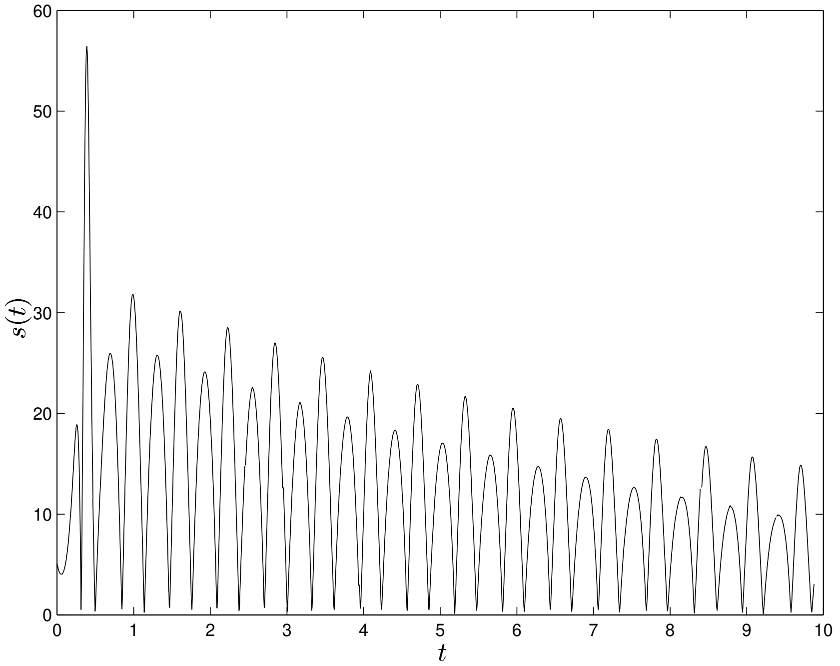


Figure 4.9: The figure shows the stability function $s = s(t)$ for the y component of the Lorenz system. The other two components are similar to this one.

4.5 True Error vs. the Error Estimate

In this section, we return to the first simple example, the harmonic oscillator, and compare the true error to the error estimate. Ideally the true error is smaller than and close to the error estimate. Is this the case for the multi-adaptive cG(q) method proposed in this work?

To check the reliability of the solver, the solution of eq. (4.1) was computed with $T = 100$ at a large number of tolerances. The results are given for cG(q), $q = 1, 2, 3$, in figure 4.10.

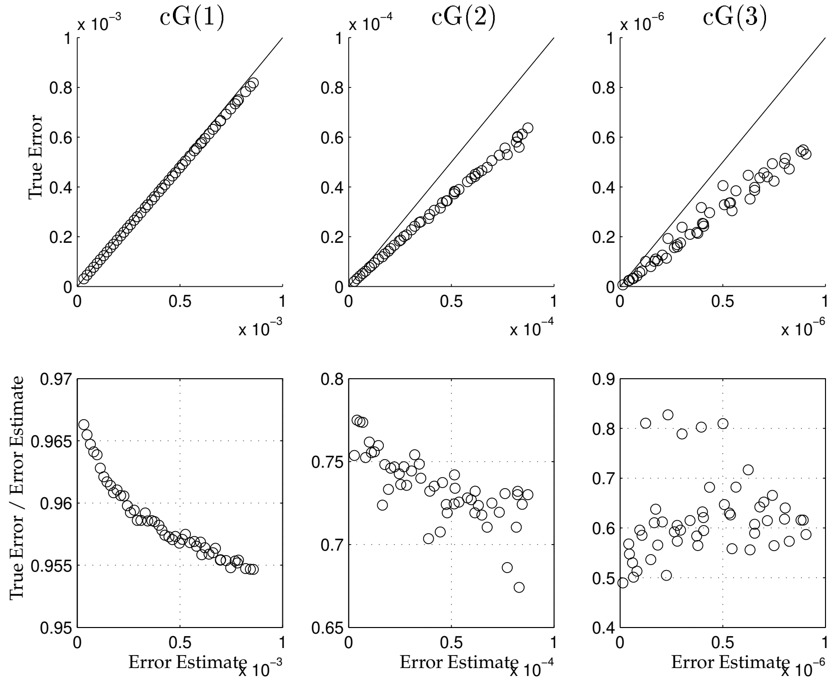


Figure 4.10: True error vs. error estimate for multi-adaptive cG(1), cG(2) and cG(3) respectively. Solid lines indicate the ideal maximum size of the true error.

As can be seen the true error is smaller than and close to the error estimate for the three methods. For this specific problem at these specific tolerance levels, the error for the cG(1) method is mostly discretizational error (arising from the finite element discretization of the error), whereas for the cG(3) method the error is mostly mostly computational (arising from a non-zero discrete residual). For the cG(2) method the situation is somewhere in between. This explains the different variances in error–tolerance correlations for the three methods.

Notice also how sharp the error estimate is, especially for the cG(1) method. Again, this is due to the fact that at this tolerance level, most of the error is the usual finite element discretizational error for the $\mathrm{cG}(1)$ method.

For comparison, the same computations were performed with the often used MATLAB ODE-solver, `ode45()`. As can be expected with a solver lacking global error control, the tolerance is only nominal, in the sense that its correlation to the true error is unknown.

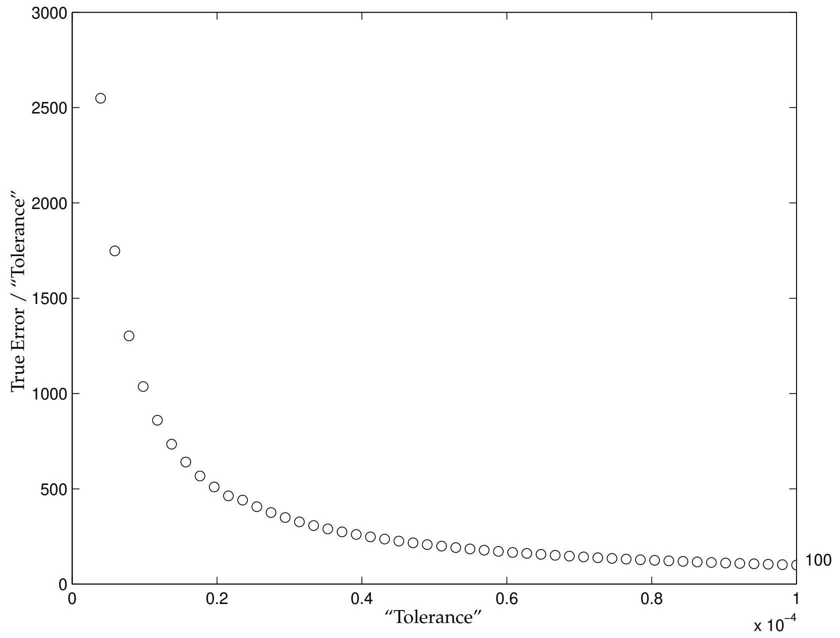


Figure 4.11: True error / tolerance vs. tolerance for MATLAB:s ODE-solver `ode45()`.

The above comparisons between true error and error estimate were made for a simple 2-component linear system. We conclude this section by showing the results for a computation on the following nonlinear problem:

$$\left\{ \begin{array}{rcl} \dot{u}_1 &=& u_1, \\ \dot{u}_2 &=& u_2 + u_1 u_1, \\ \dot{u}_3 &=& u_3 + u_1 u_2, \\ \dot{u}_4 &=& u_4 + u_1 u_3 + u_2 u_2, \\ \dot{u}_5 &=& u_5 + u_1 u_4 + u_2 u_3, \\ u(0) &=& (1, 1, \frac{1}{2}, \frac{1}{2}, \frac{1}{4}). \end{array} \right. \tag{4.4}$$

The solution is (obviously) $u(t) = (e^t, e^{2t}, \frac{1}{2}e^{3t}, \frac{1}{2}e^{4t}, \frac{1}{4}e^{5t})$.

A comparison between true error and error estimate is given in figure 4.12 for the multi-adaptive cG(1)-method. Also for this nonlinear problem, the true error is smaller than and close to the error estimate, as desired.

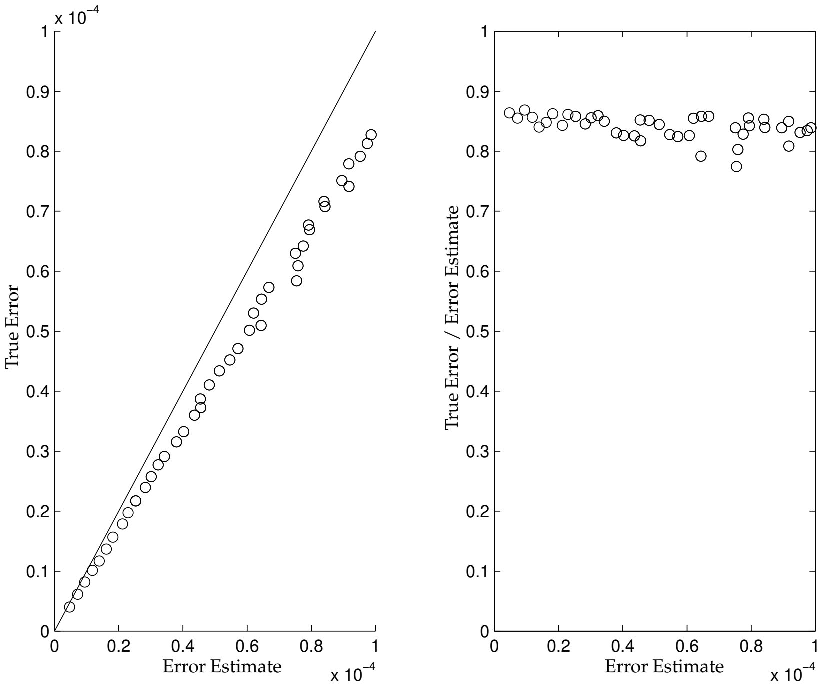


Figure 4.12: True error vs. error estimate for the multi-adaptive cG(1)-method. The solid lines indicate the ideal maximum size of the true error.

Finally, notice that these results were all obtained automatically, the only data specified being the equation (including initial data) and the tolerance. The equations were then solved automatically, including the solution of the dual problem – which was automatically generated by numerical differentiation of the given equation – and error estimation, giving a resulting final error smaller than the given tolerance.

CHAPTER 5

Conclusion

As was shown in the previous section, the correlation between error and error estimate is as desired for the three methods – at least for simple model problems.

Multi-adaptivity is thus a reality and the method is already implemented – in the Tanganyika multi-adaptive ODE-solver library. This library (at least the current version, 1.0) was written primarily with the intention to be a working implementation of the multi-adaptive method, secondarily with the intention to be a general, fast and reliable ODE-solver. Although the current implementation is indeed general and reliable, it is still not fast and effective enough, mainly because of the large amount of work needed to solve the dual problem. This has nothing to do with the multi-adaptivity itself. It is a consequence of the generation and full solution of the dual problem. There *are* cures for this and in future versions, more focus will be on speed and effectivity. The main focus, however, will always be on proper error control.

> *The facts all contribute only to setting the problem,*
> *not to its solution.*
> Ludwig Wittgentstein, Tractatus Logico-Philosophicus (1909).

CHAPTER 6

Download

The program is available for download – as is this report – at

    http://www.dd.chalmers.se/~f95logg/Tanganyika/

Included in the package is the *Tanganyika library* containing the actual solver together with *Antananarive*, an X-interface for the library. The program will run under any (not too antique) UNIX system, such as Linux, SunOS, Solaris, ... You will also need GTK, the Gimp ToolKit, for the X-interface. GTK is available for download at

    http://www.gtk.org/

The program is distributed under the GNU General Public License (GPL). (See Appendix A.)

Bibliography

[1] E. BURMAN, *Adaptive Finite Element Methods for Compressible Two-Phase Flow*, Phd thesis, Chalmers University of Technology 1998.

[2] N. ERICSSON, *A Study of Transition to Turbulence for Incompressible Flow using a Spectral Finite Element Method*, Lic thesis, Chalmers University of Technology 1998.

[3] K. ERIKSSON, D. ESTEP, P. HANSBO, C. JOHNSON, *Introduction to Adaptive Methods for Differential Equations*, Acta Numerica (1995), 105-158.

[4] K. ERIKSSON, D. ESTEP, P. HANSBO, C. JOHNSON, *Computational Differential Equations*, Studentlitteratur, 1996.

[5] D. ESTEP, *A Posteriori Error Bounds and Global Error Control for Approximations of Ordinary Differential Equations*, SIAM J. Numer. Anal. vol **32** (1995), 1-48.

[6] D. ESTEP, S. VERDUYN LUNEL, R. WILLIAMS, *Error Estimation for Numerical Differential Equations*, (1995), `http://www.cacr.caltech.edu/publications/techpubs/ [980930]`.

[7] D. ESTEP, D. FRENCH, *Global Error Control for the Continuous Galerkin Finite Element Method for Ordinary Differential Equations*, M^2AN vol **28** (1994), 815-852.

[8] D. ESTEP, R. WILLIAMS, *Accurate Parallel Integration of Large Sparse Systems of Differential Equations*, Math. Models Meth. Appl. Sci. (to appear).

[9] C. JOHNSON, *Error Estimates and Adaptive Error Time-Step Control for a Class of One-Step Methods for Stiff Ordinary Differential Equations*, SIAM J. Numer. Anal. vol **25** (1988) no 4, 908-926.

[10] R. SANDBOGE, *Adaptive Finite Element Methods for Reactive Flow Problems*, Phd thesis, Chalmers University of Technology 1996.

[11] G. DAHLQUIST, *Error Analysis for a Class of Methods for Stiff Nonlinear Initial Value Problems*, Lecture Notes in Mathematics 506, Springer-Verlag (1976).

APPENDIX A

Notation

In this chapter, I explain the notation used in this report.

Unfamiliar expressions should in general be explained when first introduced. Since, however, it is not always clear which expressions are familiar and which are not, I include the following list of notation:

FEM
: the finite element method, which is the basis for the multi-adaptive $\mathrm{cG}(q)$ method proposed in this report

$\mathrm{cG}(q)$
: a Galerkin method with continuous piecewise polynomials of order q

multi-adaptivity
: adaptive error control, where the discretizations are chosen individually for the different components [of and ODE-system]

Tanganyika
: besides being a geographical location in the south of Africa, Tanganyika is the name of the multi-adaptive ODE-solver library, based on this report

Antananarive
: this is the X-Windows interface for the Tanganyika library

dual problem
: an auxiliary problem that has to be solved in order to get an estimation of the error

u
: the solution, in this case of the initial value problem (2.1)

U
: the finite element approximation of the solution u

t
: independent variable, often thought of as the time

T
: the end-value of t

N
: the number of dimensions (components) of the ODE-system

M_i
: the number of intervals for the subpartition of $(0, T]$ for component i

V_k^N
: the trial space for our finite element formulation

W_k^N
: the test space for our finite element formulation

φ
: the solution of the *dual problem*

e
: the error of our approximate solution, i.e. $(U - u)$

J
: the Jacobian of f in eq. (2.1)

R
: the residual, i.e. $(f(U, \cdot) - \dot{U})$

k_{ij}
: the size of the j:th timestep for component i, i.e. the length of the interval I_{ij}

$C_q, \tilde{C}_q$
numerical constants appearing in the error estimates

$\overline{R}$
the discrete residual, i.e. the residual of the discrete equations obtained from the finite element formulation of the continuous problem

s_i
the *stability function* for component i, a function obtained from the solution of the dual problem, describing the local stability properties for component i

S_i
the *stability factor* for component i, a number obtained from the solution of the dual problem, describing the the global stability properties for component i

TOL
the tolerance, i.e. a beforehand specified upper bound for the error of the solution

APPENDIX B

Tanganyika User Manual

User Manual

Tanganyika Library 1.0
Tanganyika X-interface 1.0
(Antananarive)

Anders Logg

Chalmers Finite Element Center
Department of Mathematics
Chalmers University of Technology
Göteborg, Sweden
1998

Jag har en syster i Tanganyika.

B.1 Introduction

The **Tanganyika library** is a multi-adaptive solver of initial value problems for ordinary differential equations. The method used for solving the equations is a variant of the $\mathrm{cG}(q),\ q = 1, 2, 3$, finite element method.

The solver is *adaptive* in the sense that the size of the timesteps is chosen small enough to give an error smaller than the given tolerance, equidistributing the error onto the different intervals. The solver is *multi-adaptive* in the sense that the timesteps are chosen individually for the different components.

For further details on the solver, download the report *A Multi-Adaptive ODE-Solver* from

`http://www.dd.chalmers.se/~f95logg/Tanganyika/`

The Tanganyika X-interface, **Antananarive**, is just that, an X-windows interface for the Tanganyika Library.

B.2 Download

To download the Tanganyika library and X-interface, goto

`http://www.dd.chalmers.se/~f95logg/Tanganyika/`,

click the link named *Download* and follow further instructions on this page.

You will then receive the whole package, containing everything you need – almost. In addition you must also have **GTK**, the Gimp ToolKit, installed on your system. (GTK is used by the X-interface for drawing the buttons.) If you just want to use the library and if you can do without the buttons, you don't need GTK. However, if you do want the X-interface and you don't have GTK, download GTK from

`http://www.gtk.org/`

and install it according to the instructions.

B.3 Installation

If you haven't realized that until now, you should be on a Unix system (Linux, SunOS, Solaris,...). For the following instructions, it is assumed that commands are typed to a shell (`/bin/bash`, `/bin/sh`, `/bin/tcsh`, ...), i.e. probably in an xterm. Commands are written as

```
>> command
```

Note that you shouldn't type the '>> '! Oh well, you probably know all this but just in case you're one of our sysadmins at `dd.chalmers.se`... `;`)

1. **Unpacking.** The first thing you need to do is to unpack the Tanganyika source code. To do this, type

   ```
   >> unzip tanganyika-1.0.zip
   ```

   or the corresponding command for uncompression if you choose to download another format.

   This will create a directory (with a couple of sub-directories) named

   ```
   Tanganyika-1.0/
   ```

2. **Configuring.** Edit the file `defs` in the `Tanganyika-1.0` library for the variables to match your system. It should probably look something like this:

   ```
   CC           =g++
   LINK         =g++

   INCLUDE_PATH =-I/usr/include -I/usr/include/g++
   LIBRARY_PATH =-L/usr/lib
   ```

   as it does on my Linux 2.0 or

   ```
   CC           =g++
   LINK         =g++

   INCLUDE_PATH    =-I/opt/gnu/include
   LIBRARY_PATH    =-L/opt/gnu/lib -R/opt/gnu/lib
   ```

   as it does on Sun Solaris 2.6 at `dd.chalmers.se`.

3. **Compiling.** Compile the library and the X-interface by typing

   ```
   >> make
   ```

in the `Tanganyika-1.0/` directory. The library and the X-interface will now compile. (If not, something went wrong and hopefully you know how to deal with it.)

This will also generate the file `.antananariverc` in your home directory.

4. **Running the demo.** Check if you managed to compile the library correctly by typing

```
>> ./demo
```

in the `Tanganyika-1.0/bin/` directory. This should result in some text output ending with something like

```
Message: Computing error estimate...
Message: ...done!
Message: Error estimate: 7.526e-04 <= TOL = 1.000e-03

Message: Error estimate small enough, so I'm done.
Message: Saving...
Message: ...done!
```

and data stored in the file `tst.data`, together with a MATLAB `.m` file.

You may also want to run the X-interface by typing

```
>> ./antananarive
```

in the same directory.

5. **Completing the Installation.** Complete the installation by putting the generated files wherever you want them. You may want to do the following (assuming the current directory is the `Tanganyika-1.0/` directory):

   - Place the X-interface. Type e.g.

     ```
     >> cp bin/antananarive /usr/local/bin
     ```

     or

     ```
     >> cp bin/antananarive /usr/bin
     ```

- Place the library header file. Type e.g.

  ```
  >> cp include/tanganyika.h /usr/include
  ```

- Place the library. Type e.g.

  ```
  >> cp lib/libtanganyika.a /usr/lib
  ```

Notice that you probably cannot do this otherwise than as superuser (root)!

B.4 The Tanganyika Library

This is a tutorial for the Tanganyika library, version 1.0.

B.4.1 What it does

This library provides functions for solving initial value problems for systems of ordinary differential equations. The method used is a multi-adaptive finite element method, which is described in detail in the report *A Multi-Adaptive ODE-solver*.

B.4.2 How to use it

In your C(++) program, include the library by doing

```
#include <tanganyika.h>
```

What will then be included is the following:

```
#ifndef TANGANYIKA_H
#define TANGANYIKA_H

#define TAN_METHOD_CG1 1
#define TAN_METHOD_CG2 2
#define TAN_METHOD_CG3 3

#define TAN_OUTPUT_DEVNULL 0
#define TAN_OUTPUT_COUT    1
#define TAN_OUTPUT_CERR    2
#define TAN_OUTPUT_COM     3

#define TAN_FORWARD_PROBLEM 1
#define TAN_DUAL_PROBLEM    2
#define TAN_ERROR_ESTIMATE  3
```

```
bool InitializeSolution (double  *dInitialData,
                         double   dStartTime,
                         double   dEndTime,
                         double   dTolerance,
                         double (*fFunction)
                         (double *U, double t, int iIndex),
                         int     *iMethods,
                         int      iSizeOfSystem,
                         int      iMessageOutput,
                         void   (*Progress) (double dProgress),
                         bool     bErrorEstimation);
void ClearSolution();
bool Solve ();
bool Save(const char *cFileName);

#endif
```

What the different functions do should be quite clear from their names. Below follows a description of the different functions.

B.4.3 `InitializeSolution()`

Use this function to tell the library what to solve. The data passed to this function are described below.

1. **`dInitialData`** should be a valid pointer to a block of doubles, specifying the initial data for the problem, i.e. e.g.

   ```
   double *dInitialData = new double[2];

   dInitialData[0] = 0.0;
   dInitialData[1] = 1.0;
   ```

2. **`dStartTime`** should be a double specifying the time, t, at the beginning of the solution. (You probably want to pass `0` for this argument.)

3. **`dEndTime`** should be a double specifying the time, t, at the end of the solution, such as e.g. `10`.

4. **`dTolerance`** should be a double specifying the tolerance for the l^2-norm of the error at the end of the solution. The library will try to solve the equations with an error that is smaller than this tolerance.

5. **`fFunction`** should be a pointer to a function specifying the equations, being

$$\dot{u}(t) = f(u,t),\ u(t) \in \mathbf{R}^N, \tag{B.1}$$

i.e. e.g.

```
double f(double *U, double t, int iIndex)
{
   switch(iIndex){
   case 0:
     return ( U[1] );
   case 1:
     return ( sqrt(U[1]) + U[0] );
   default:
     return 0.0;
   }
}
```

In this example, the name of the function (that must be declared with the parameter list as above) is `f`, so the reference that should be passed to `InitializeSolution()` is simply the name of the function, i.e. `f`.

6. **`iMethods`** should be a valid pointer to a block of ints, specifying the methods to be used for the different components. Valid values are

   `TAN_METHOD_CG1`,
   `TAN_METHOD_CG2` and
   `TAN_METHOD_CG3`.

7. **`iSizeOfSystem`** should be an integer specifying the size of the system, i.e. the number of equations.

8. **`iMessageOutput`** should be an integer specifying the desired type of output from the library during solution. Valid values are

   `TAN_OUTPUT_DEVNULL`,
   `TAN_OUTPUT_COUT`,
   `TAN_OUTPUT_CERR`,
   `TAN_OUTPUT_COM`.

These will set the adress of output from the program.
`TAN_OUTPUT_DEVNULL` means no output will be written.
`TAN_OUTPUT_COUT` means output will be to standard output.
`TAN_OUTPUT_CERR` means output will be to standard error.
`TAN_OUTPUT_COM` means output will be to standard output, in a special format that may be interpreted by e.g. the Tanganyika X-interface. With this output set, the current status of the program will be written to standard output as

```
STAT iStatus,
```

where `iStatus` is one of
`TAN_FORWARD_PROBLEM`,
`TAN_DUAL_PROBLEM` or
`TAN_ERROR_ESTIMATE`,
indicating what is going on.

9. **`Progress`** should be a pointer to a function that will be passed the progress of the computation, the progress being a number between 0 and 1. This might be useful for updating e.g. progress bars. (The Tanganyika X-interface does not use this for updating the progress bars. Instead the progress is parsed from the output.)

   The function should be declared as

```
void FunctionName(double dProgress)
{
    // Code goes here
}
```

10. **`bErrorEstimation`** should be `true` or `false`, telling whether or not an error estimate should be computed. If `false`, no dual problem will be solved and the given tolerance will only be nominal in the sense that it won't (necessarily) be close to the true error. However, a smaller nominal tolerance will (probably) mean a smaller error.

`InitializeSolution()` will return `true` upon successful initialization of the solution and `false` if something went wrong, i.e. if e.g. the data passed was illegal. (A negative tolerance or whatever.)

B.4.4 `ClearSolution()`

Call this function to free all memory used by the library.

B.4.5 `Solve()`

Call this function to solve the equations, after having done `InitializeSolution()`. The return value will be `true` or `false` depending on whether or not the computation was successful.

B.4.6 `Save()`

Call this function to save data from the computation in MATLAB format. This will generate two files. One `filename.m` file and one `filename.data` file, where `filename` is the filepath specified by `cFileName`. The first of these is called from MATLAB by typing the name of the file (excluding the `.m` extension), which will load the data stored in the second one into the proper variables.

B.5 The Tanganyika X-interface, Antananarive

This is a tutorial for Antananarive, the Tanganyika X-interface, version 1.0.

B.5.1 Introduction - What is this program anyway?

This program is an interface for the Tanganyika multi-adaptive ODE-solver library. All it does is to call the library functions to generate a program from given user data. This program is then compiled using `g++` or whichever compiler you prefer. (This may be changed in the "settings..." menu or in the `.antananariverc` file in your home directory.)

The compiled program will output data to files (file path specified in the "options..." menu) in Matlab format. Two files will be generated; one `.data` and one `.m`. Data from the solution is stored (ASCII) in the first of these files. Typing the filename in Matlab will call the `.m`-file, reading all data properly from the `.data`-file.

The compiled program may be run either from this program (the Tanganyika X-interface, version 1.0) or manually from a shell. If you run the compiled program from this program, you get the benefit of parsed output as messages and progress bars. (The Tanganyika X-interface will read output from the generated program at standard output.)

B.5.2 Using the program - Step by Step

All you have to do is to press

open - (edit) - (save) - make - solve

in that order. Below follows a more detailed description.

1. Open a `.xt`-file, specifying the system of ordinary differential equations, by pressing the "Open..." button and then choosing a `.xt`-file. (The suffix is not really important so there may be xt-files without the `.xt`-suffix.)

2. Edit the equations by pressing the "edit" button. The contents of the opened file will then be editable in the text window. Of course you don't have to edit the file if you don't wanna change anything, but remember to save the file before moving on to compiling the program, as the program will be generated from the file and not from the contents of the text window. For information on the syntax, see the section below, "`.xt`-file syntax".

3. Save the changes if you made any by pressing the "save..." button and then typing/choosing a file name. Note that the `.xt`-suffix will not be added automatically.

4. Generate the program and compile it by pressing the "make" button. A `.C`-file will then be generated in the working directory specified in the "settings..." menu. This file is then compiled and the output program will be `filename.bin`, also in the working directory.

5. Solve the equations by pressing the "solve" button. This will run the generated program and parse its output to update the progress bars and typing status of the solution. By the way, the upper of the two progress bars is for the forward solution (the solution of the equations you specified) and the one below is for the solution of the dual (backward) problem that is solved to estimate the error of the solution.

B.5.3 Settings

Working directory
This is where the `.C` and `.bin` files will be generated.

Compiler
This is the name of the (C++) compiler present in your system.

CFLAGS
These flags are passed to the compiler, specifying e.g. code optimizations.

INC_PATH
This is where the compiler will look for include files.

LIB_PATH
This is where the compiler will look for libraries (`lib*`-files).

B.5.4 Options

Start Time
Well,..., this is the start time, the value of t at the beginning.

End Time
And this would then be T, the value of t at the end of the solution.

Tolerance
This is the value of the tolerance for the l^2-norm error of the solution at $t = T$.

Output filename
This is the file in the current working directory where the generated program will store the solution.

B.5.5 `.xt`-file syntax

- You have to specify four things:

The size of the system:	`N = ?`
Initial data:	`U[i] = ?`
Equations:	`F[i] = ?`
Methods:	`M[i] = ?`

- All data must end with a semicolon (`;`).
- `%` at the beginning of a line means a comment, i.e. this line will not be interpreted.
- Indices begin with `0`!
- Specification of equations must be C syntax. You may thus not write

  ```
  F[5] = U[2] * U[1]^2 + sqrt(abs(U[3]));
  ```

  Instead you must write

  ```
  F[5] = U[2] * pow(U[1],2) + sqrt(fabs(U[3]));
  ```

- Methods are specified as integers `1`, `2` or `3` for

  cG(1): continuous first-order Galerkin,
  cG(2): continuous second-order Galerkin and
  cG(3): continuous third-order Galerkin,

  respectively.

Here follows a simple example (for a simple harmonic oscillator):

```
%
% This is an example
%

% size of system
N = 2;

% initial data
U[0] = 0;
U[1] = 1;

% equations
F[0] = U[1];
F[1] = -U[0];

% methods
M[0] = 1;
M[1] = 2;
```

B.5.6 Download, Updates, Further Information

This program is available for download at

```
http://www.dd.chalmers.se/~f95logg/Tanganyika/
```

together with the Tanganyika library. At this site is also available in postscript format the report *A Multi-Adaptive ODE-Solver*.

B.6 GNU General Public License

These programs (both the Tanganyika library 1.0 and the Tanganyika X-interface) are distributed under the GNU General Public license, GPL, included below.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

fin