Solving systems of nonlinear equations with arbitrary precision

CES Seminar

January 2017

Presented by	Gregor Heiming
Supervisor	Dr. Ralf Hannemann-Tamás Systemverfahrenstechnik AVT, RWTH Aachen University

Contents

1 Introduction

When solving partial differential equations (PDEs) numerically, the convergence of solution methods highly depends on the stability of the method. Typically, for smooth problems, linear stability suffices for convergence [4]. Hyperbolic PDEs such as the wave equation, however, often have discontinuous solutions. Hyperbolic PDEs are usually solved using the method of lines. In the method of lines, first, the spatial components are discretized either by finite differences or a finite element approach. In a second step the resulting system of ordinary differential equations (ODEs) is solved, where the time step size is restricted by the CFL coefficient [12].

For hyperbolic PDEs with non-smooth solutions, the spatial discretization is chosen such that stability is given when solving the ODE system with the Forward Euler method for time discretization [4]. Since usually higher order time discretization schemes are desired, so-called strong stability preserving (SSP) Runge-Kutta methods were developed. It is assumed that the Forward Euler time discretization is strongly stable under a certain norm, to develop a higher order time discretization scheme which keeps the strong stability for the same norm.

In order to optimize SSP Runge-Kutta methods, i.e. to maximize their CFL coefficient, systems of nonlinear equations need to be solved [12]. Since these methods have to maintain stability, the optimized coefficients shall be as precise as possible: Using arbitrary precision floating point numbers seems to be useful. As for many efficient optimization methods, the Jacobian of the system is required, we want to be able to compute derivatives without losing precision. This would be the case for finite differences. Here, automatic differentiation (AD) is the right way.

In this work, it is practically shown, that existing libraries for arbitrary precision and automatic differentiation can be combined for efficiently solving systems of nonlinear equations.

2 Basics

This section shall refresh some basic equations and methods, that are required for understanding the section about solution methods for nonlinear equation systems below.

2.1 Arbitrary Precision

Usually in computer codes, the precision of floating point numbers is restricted by the word length of the system. The frequently used double precision type has a maximum precision of $1.11 \cdot 10^{-16}$.

Arbitrary precision or also called multiple precision datatypes, as the name already tells, can have arbitrary precision. The actual restriction of the precision is the available memory of the machine. As an implementation of arbitrary precision floating point numbers, the GNU library MPFR [3] is used. Since this library requires to use its functions instead of the usual C++ operators, the wrapper MPFR C++ by Holoborodko [6] is useful.

2.2 Automatic Differentiation

Automatic differentiation (AD), also known as algorithmic or computational differentiation is used to obtain derivatives of computer programs, assuming that the code is continuously differentiable.

There are mainly two different ways of achieving this, namely tangent and adjoint mode [9]. In this work, we will only consider tangent mode. The basic idea of automatic differentiation is the usage of the chain rule of differentiation. The derivatives of basic operations such as addition or multiplication and those of frequently used function such as sin and cos are known. By splitting up the code into basic functions, differentiating these basic functions and applying the chain rule, the derivative of any code can be obtained. This is shown with an example below.

Example $y = \cos(x_1 x_2)$. Find $\frac{\partial y}{\partial x_1}$

$$
\begin{aligned}
v_1 &= x_1 & v_1' &= 1 \\
v_2 &= x_2 & v_2' &= 0 \\
v_3 &= v_1 v_2 & v_3' &= v_1 v_2' + v_1' v_2 \\
v_4 &= \cos(v_3) & v_4' &= -\sin(v_3) v_3' \\
y &= v_4 & y' &= v_4' \\
& & &= -\sin(x_1 x_2) x_2
\end{aligned}
$$

The function is shown as a graph in figure 1.

For automatic differentiation by operator overloading, a datatype is used, which in general consists of two numbers, where one represents the function value while the other one represents the tangent-linear derivative. For the AD datatype, all the basic functions

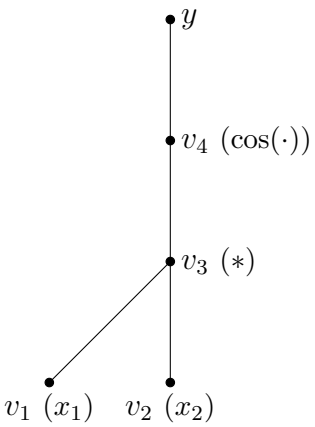

Figure 1: Graph visualization of function $y = \cos(x_1 x_2)$

and operators are overloaded, such that not only the value but also the derivative is computed. In this work, the community edition of the AD library dco/c++ [7] is used.

2.3 Convergence

In this section, some terms regarding the convergence of solution methods are defined, which are then used in the descriptions of the methods below. In general, the convergence order specifies, how fast a method approximates the solution.

2.3.1 Convergence order

Convergence order $p > 1$ means that the norm of the iteration step decreases with given power p. Mathematically speaking: There is an $M < \infty$, s.t.

$$\|x^{(k+1)} - x^*\| \leq M\|x^{(k)} - x^*\|^p \quad \forall k \geq k_0, \tag{1}$$

where k denotes the iteration step. Any norm $\|\cdot\|$ of $\mathbb{R}^n$ can be used. In the following, we will only consider the Eclidean norm, i.e.,

$$\|x\| = \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}. \tag{2}$$

Superlinear convergence A method is said to have superlinear convergence if there is a sequence α_k converging to zero [8] with

$$\|x^{(k+1)} - x^*\| \leq \alpha_k\|x^{(k)} - x^*\|, \;\; k \geq 0. \tag{3}$$

3 Method

In this section, methods for solving systems of nonlinear equations are presented. Before looking at the methods in detail, some general definition shall be made and the notation used in this work is shortly shown.

Let $F : \mathbb{R}^n \to \mathbb{R}^n$ be a sufficiently smooth map, with

$$\begin{aligned} F(x) &= (f_1(x), f_2(x), \ldots, f_n(x))^{\mathrm{T}}, \text{ and} \\ x &= (x_1, x_2, \ldots, x_n)^{\mathrm{T}}. \end{aligned} \tag{4}$$

Then, the problem that is to be solved reads:

$$\boxed{\text{Find a solution } x^* \text{ such that } F(x^*) = 0.}$$

We will use iterative methods for solving problems of this kind. Vector entries are denoted by subscripts, for iteration steps we will use superscripts. For example, the i-th entry of the vector x in iteration step k will be denoted as $x_i^{(k)}$.

3.1 Newton's method

The most established method for solving systems of nonlinear equations is Newton's method [1]. The method is defined as follows:

$$x^{(k+1)} = x^{(k)} - \left(\nabla F(x^{(k)})\right)^{-1} F(x^{(k)}) \tag{5}$$

Usually, Newton's method converges locally with convergence order 2. However, there are several cases where the method fails. One example is a point, where the Jacobian ∇F is singular, so that the linear system in (5) has either no solution at all, or the solution is non-unique.

For large systems, Newton's method is expensive, since in each step the Jacobian needs to be computed and inverted.

In Figure 2, the method is shown at an example function in one dimension. It can be seen that $x^{(3)}$ corresponds to the solution $x^* = 0$.

3.2 Powell's hybrid method

Powell's hybrid method [10, 11] is a rather cheap method for solving systems of nonlinear equations. Similar to Quasi-Newton methods, the Jacobian is approximated and updated in each step without having to determine the local derivatives. The basic scheme is the same as in Newton's method.

$$x^{(k+1)} = x^{(k)} + p^{(k)} \tag{6}$$

Here, the step $p^{(k)}$ is determined from a combination of the Newton step, see equation (5), and the direction of steepest descent.

The implementation of Powell's hybrid method that was implemented during this work starts with the exact Jacobian in the first iteration step. Therefore, the Jacobian is evaluated exactly once and the function itself is evaluated once in each step.

The method converges superlinearly.

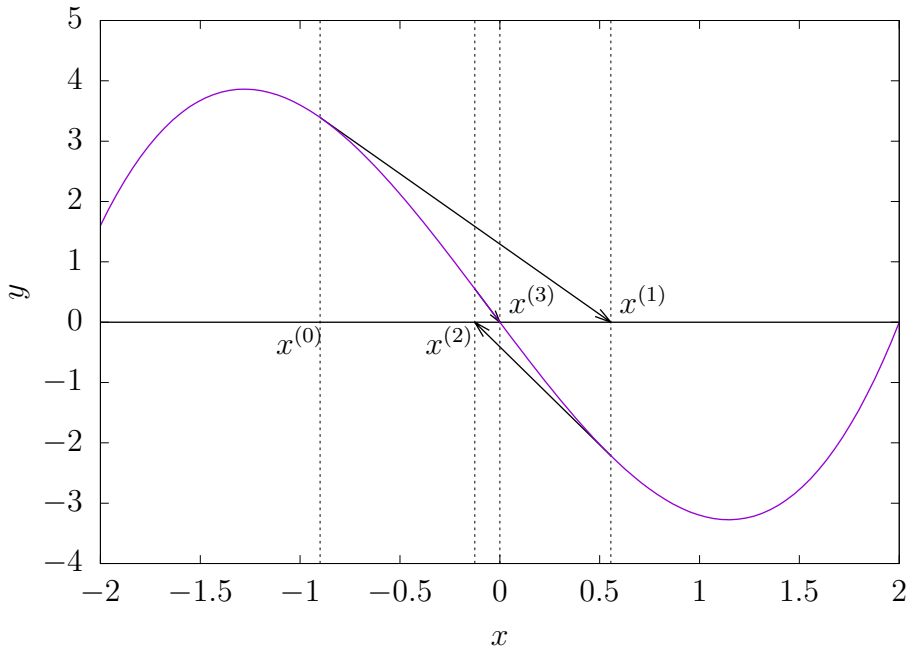

Figure 2: Visualization of Newton's method in 1D

3.3 Fourth order method

Sharma and Arora [13] developed methods of higher order, such as the 4th order Jarratt-like method which is defined in equation (7).

$$y^{(k)} = x^{(k)} - \frac{2}{3}\left(\nabla F(x^{(k)})\right)^{-1} F(x^{(k)}) \tag{7a}$$

$$\begin{aligned} x^{(k+1)} = x^{(k)} - \Bigg[&\frac{23}{8} I - \left(\nabla F(x^{(k)})\right)^{-1} \nabla F(y^{(k)}) \\ &\times \left(3I - \frac{9}{8}\left(\nabla F(x^{(k)})\right)^{-1} \nabla F(y^{(k)})\right)\Bigg] \left(\nabla F(x^{(k)})\right)^{-1} \nabla F(x^{(k)}) \end{aligned} \tag{7b}$$

This method has two function evaluations, two Jacobian computations and one Jacobian inversion per step. However, it is still quite efficient due to the high convergence order. Similar to Newton's method, it only converges locally.

3.4 Hybrid method

From the previously described methods, a hybrid method can be formed in the sense that the computational rather cheap Powell's method can be used to approach a solution point with low accuracy. As a second step a local method, such as the Jarratt-like fourth order method, is used to solve the system with a high precision.

3.5 Termination

All the solution methods above use the same termination conditions. The first condition is a tolerance check: If the norm of the residual falls below the user-provided tolerance value, the method terminates successfully. A similar check is performed with the step size. If the step size falls below a user-provided value, this means that the method is stuck in a local minimum and is therefore stopped. The other termination conditions are maximal numbers of Jacobian and function evaluations respectively and a maximal number of iterations.

4 Implementation

During this work, the previously described solution methods were implemented in C++. Since special features, such as automatic differentiation, were used, external libraries were included.

One requirement for the code was the possible use of different datatypes. We want to support arbitrary precision datatypes, but the code should also work with the standard datatypes float and double. Therefore, the implemented functions are templated.

In the following, the included libraries are briefly described:

MPFR C++ This multiple precision library [6] is the wrapper for the GNU library MPFR, where basic operations are overloaded. Thanks to MPFR C++, compared to GNU MPFR, the used datatype can be exchanged easily.

Eigen The Eigen library [5] is a powerful linear algebra library. It is used for vector and matrix operations and allows the use of arbitrary datatypes. It even has support for the arbitrary precision datatype from MPFR C++ [6].

dco/c++/communityedition dco [7] is an AD library developed by the Software Tools in Computational Engineering group at RWTH Aachen University. The advantage of dco compared to other AD libraries, is the use of templated types. This makes the combination of arbitrary precision and automatic differentiation possible.

Jacobian Inversions In the methods above, Jacobian inversions are required. Inverting matrices is usually avoided in numerical analysis. Instead of computing the inverse, we introduce a new variable which is defined as the solution of a system of linear equations where the Jacobian is the system matrix. Looking for example at the Newton's method:

$$x^{(k+1)} = x^{(k)} - \left(\nabla F(x^{(k)})\right)^{-1} F(x^{(k)}) \qquad \text{(5 revisited)}$$

We define the Newton step direction $p^{(k)}$ as the solution of the equation system

$$\nabla F(x^{(k)}) \cdot p^{(k)} = F(x^{(k)}). \tag{8}$$

The resulting definition of Newton's method is then given by

$$x^{(k+1)} = x^{(k)} - p^{(k)}. \tag{9}$$

For solving systems of linear equations, there are efficient direct and iterative methods. Thus, this is cheaper than directly inverting the Jacobians.

Interface The interface of the implemented functions is stated in listing 1. The solvers autonomously call the driver routine for computing the Jacobian. For this, the evaluation function needs to be called with the dco datatype. The evaluation function is therefore passed to the solver as a Functor with arbitrary type, so the type does not need to be set with the function call.

```
template<template<typename> class TFunctor,typename TREAL>
bool powellHybridSolver(const unsigned n,    // Dimension
                  Matrix<TREAL,Dynamic,1> &x, // Solution vector
                  TREAL xtol,                 // Step tolerance
                  TREAL tol,                  // Residual tolerance
                  unsigned maxit,             // max num of iterations
                  unsigned maxfev,            // max num of fctn eval
                  unsigned maxjev,            // max num of Jacobian eval
                  unsigned &nit,              // actual num of iterations
                  unsigned &nfev,             // actual num of fctn eval
                  unsigned &njev)             // actual num of Jac. eval

template<template<typename> class TFunctor, typename TREAL>
bool newtonSolver(const unsigned n,
                  Matrix<TREAL,Dynamic,1> &x,
                  TREAL xtol,
                  TREAL tol,
                  unsigned maxit,
                  unsigned maxfev,
                  unsigned maxjev,
                  unsigned &nit,
                  unsigned &nfev,
                  unsigned &njev)

template<template<typename> class TFunctor, typename TREAL>
bool jlm4Solver(const unsigned n,
                  Matrix<TREAL,Dynamic,1> &x,
                  TREAL xtol,
                  TREAL tol,
                  unsigned maxit,
                  unsigned maxfev,
                  unsigned maxjev,
                  unsigned &nit,
                  unsigned &nfev,
                  unsigned &njev)
```

Listing 1: Interface of the implemented solution methods

5 Results and Validation

In this section, the previously described methods are tested against each other. Here, the main focus shall be on the implementation with arbitrary precision datatype.

For validation and comparison of the methods, we use the following test equation system taken from [2].

$$
\begin{aligned}
F(x) &= (f_1(x), f_2(x), \ldots, f_{15}(x))^{\mathrm{T}}, \text{ with} \\
f_1 &= 2x_1 + x_2 + x_3 + x_4 + x_5 - 6, \\
f_2 &= x_1 + 2x_2 + x_3 + x_4 + x_5 - 6, \\
f_3 &= x_1 + x_2 + 2x_3 + x_4 + x_5 - 6, \\
f_4 &= x_1 + x_2 + x_3 + 2x_4 + x_5 - 6, \\
f_5 &= x_1 x_2 x_3 x_4 x_5 - 1, \\
f_6 &= f_1 + (2 - x_6)x_6 - 2x_7 + 1, \\
f_7 &= f_2 + (3 - x_7)x_7 - x_6 - 2x_8 + 1, \\
f_8 &= f_3 + (3 - x_8)x_8 - x_7 - 2x_9 + 1, \\
f_9 &= f_4 + (3 - x_9)x_9 - x_8 - 2x_{10} + 1, \\
f_{10} &= f_5 + (1 - x_{10})x_{10} - x_9 + 1, \\
f_{11} &= f_6 + 5 - \sigma + (1 - \cos(x_{11})) - \sin(x_{11}), \\
f_{12} &= f_7 + 5 - \sigma + 2(1 - \cos(x_{12})) - \sin(x_{12}), \\
f_{13} &= f_8 + 5 - \sigma + 3(1 - \cos(x_{13})) - \sin(x_{13}), \\
f_{14} &= f_9 + 5 - \sigma + 4(1 - \cos(x_{14})) - \sin(x_{14}), \\
f_{15} &= f_{10} + 5 - \sigma + 5(1 - \cos(x_{15})) - \sin(x_{15}), \\
\text{with } \sigma &= \cos(x_{11}) + \cos(x_{12}) + \cos(x_{13}) + \cos(x_{14}) + \cos(x_{15}).
\end{aligned}
\tag{10}
$$

Figure 3 shows how the residual of the function decreases. Different convergence speeds can be seen, where it is important to mention that this does not show the convergence over the runtime. One iteration of the fast converging Jarratt-like method requires two evaluations of the Jacobian which is quite expensive for large systems.

In figure 4 the development of the iteration step lengths is plotted. In both plots 3 and 4, the y-axis is important to look at because here, the achieved precision can be seen.

6 Conclusion and Outlook

In this work, it was shown that we can combine arbitrary precision with automatic differentiation, and use this for solving systems with nonlinear equations. As mentioned in the introduction, this shall be used to optimize strong stability preserving Runge-Kutta methods. After this proof of concept, the developed code can be extended by further solution methods.

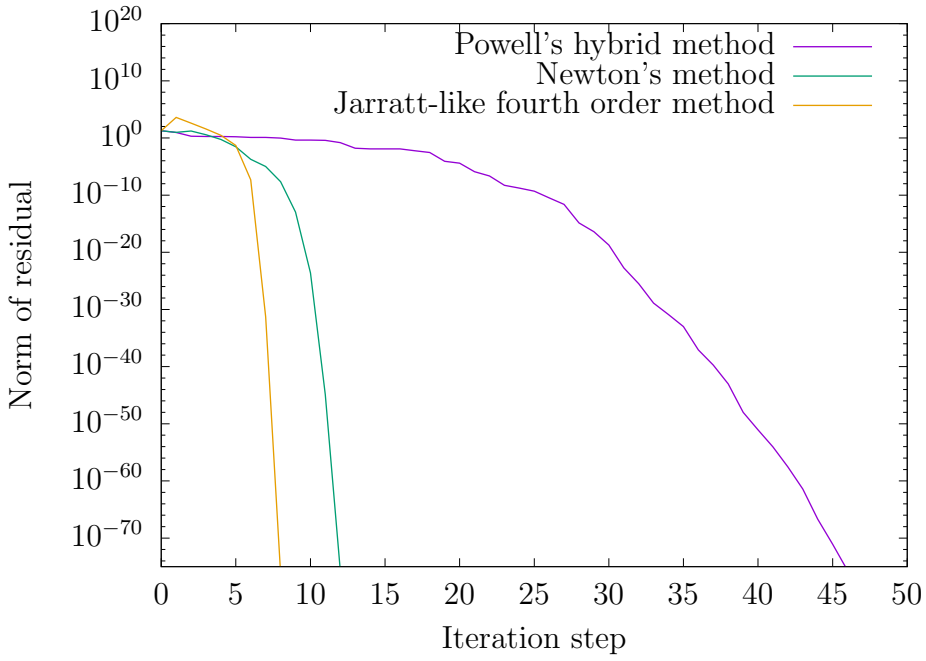

Figure 3: Comparison of the solution methods: Development of the residual

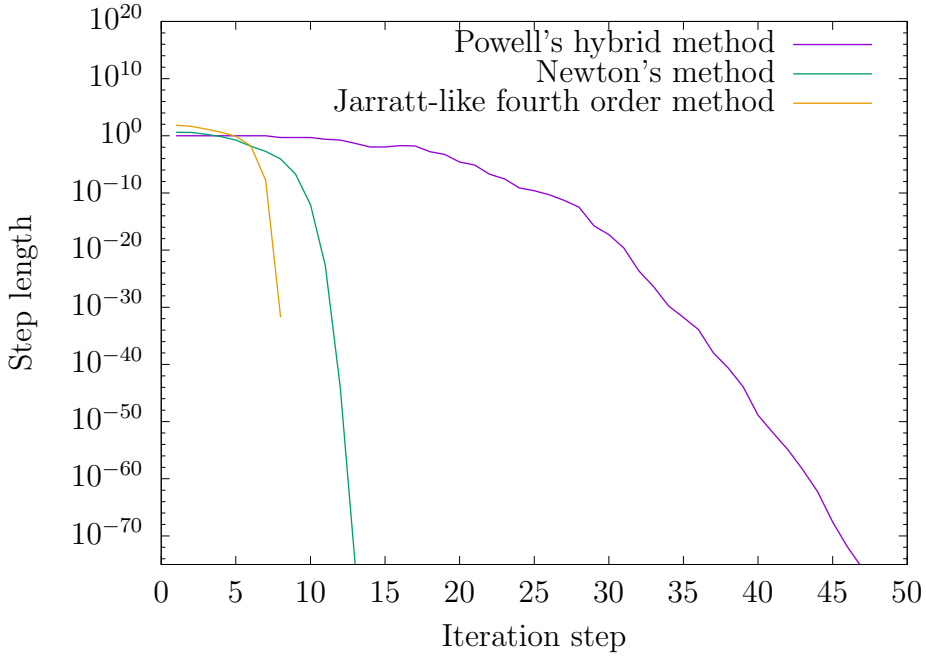

Figure 4: Comparison of the solution methods: Development of the step length

References

[1] Wolfgang Dahmen and Arnold Reusken. *Numerik für Ingenieure und Naturwissenschaftler*. Springer-Verlag, 2006.

[2] JE Dennis, Jr, Jose Mario Martinez, and Xiaodong Zhang. Triangular decomposition methods for solving reducible nonlinear systems of equations. *SIAM Journal on Optimization*, 4(2):358–382, 1994.

[3] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Transactions on Mathematical Software (TOMS)*, 33(2):13, 2007.

[4] Sigal Gottlieb. On high order strong stability preserving Runge–Kutta and multi step time discretizations. *Journal of Scientific Computing*, 25(1):105–128, 2005.

[5] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[6] Pavel Holoborodko. MPFR C++. http://www.holoborodko.com/pavel/mpfr/, 2008–2012.

[7] Klaus Leppkes, Johannes Lotz, and Uwe Naumann. dco/c++/communityedition - Algorithmic Differentiation by Operator Overloading in C++, 2014–2016.

[8] Jorge J Moré and John Arthur Trangenstein. On the global convergence of Broyden's method. *Mathematics of Computation*, 30(135):523–540, 1976.

[9] Uwe Naumann. *The art of differentiating computer programs: an introduction to algorithmic differentiation*, volume 24. Siam, 2012.

[10] Michael JD Powell. A FORTRAN subroutine for solving systems of nonlinear algebraic equations. Technical report, Atomic Energy Research Establishment, Harwell (England), 1968.

[11] Michael JD Powell. A hybrid method for nonlinear equations. *Numerical methods for nonlinear algebraic equations*, 7:87–114, 1970.

[12] Steven Ruuth. Global optimization of explicit strong-stability-preserving Runge-Kutta methods. *Mathematics of Computation*, 75(253):183–207, 2006.

[13] Janak Raj Sharma and Himani Arora. Efficient Jarratt-like methods for solving systems of nonlinear equations. *Calcolo*, 51(1):193–210, 2014.