Improving Newton's method for Initialization of Modelica models

Ylikiiskilä, Johan; Åkesson, Johan; Führer, Claus

2011

Improving Newton's method for Initialization of Modelica models

Johan Ylikiiskilä[†]
Johan Åkesson[*†], Claus Führer[**]

[*] Departement of Automatic Control, Lund University

[**] Departement of Numerical Analysis, Lund University

[†] Modelon AB
Ideon Science Park
SE-22370, Lund, Sweden
E-mail: info@modelon.se

[*/**] Lund University
Sölvegatan 18
SE-22100, Lund, Sweden
E-mail:claus@maths.lth.se

Abstract

Initializing a model written in Modelica translates to finding consistent initial values to the underlying DAE. Adding initial equations and conditions creates a system of non-linear equations that can be solved for the initial configuration. This paper reports an implementation of Newton's method to solve the non-linear initialization system. This implementation also uses a regularization method to deal with singular Jacobians as well as sparse solvers to exploit the sparsity structure of the Jacobian. The implementation is based on the open-source projects JModelica.org and Assimulo, KINSOL from the SUNDIALS suite and SuperLU.

Keywords: initialization; Newton's method; regularization; JModelica.org; Assimulo; KINSOL; SuperLU

1 Introduction

The initialization of a Modelica model is equivalent to finding consistant intial values to the underlying DAE:

$$\mathbf{F}\left(\dot{\mathbf{x}}, \mathbf{x}, \mathbf{w}, t\right) = 0 \tag{1}$$

Here $\mathbf{x} \in \mathbb{R}^{n_x}$ are the states and $\dot{\mathbf{x}} \in \mathbb{R}^{n_x}$ their time derivatives. $\mathbf{w} \in \mathbb{R}^{n_w}$ are the algebraic variables and t is the time.

In JModelica.org, initialization is performed by creating a system of, often non-linear, equations called the initialization system:

$$\mathbf{F}_0\left(\dot{\mathbf{x}}, \mathbf{x}, \mathbf{w}, t\right) = 0 \tag{2}$$

The system $\mathbf{F}_0$ consists of the equations describing the derivatives and algebraic variables in (1), and in addition, $\mathbf{F}_0$ also contains information such as initial equations and fixed start values. How $\mathbf{F}_0$ is formed is explained in Section 2. The non-linear system of equations (2) can be solved in a multitude of ways. This paper focuses on one of the most common, Newton's method.

To simplify notation the three vectors solved for, $\dot{\mathbf{x}}$, $\mathbf{x}$ and $\mathbf{w}$, are grouped together by the notation $\mathbf{u} = [\dot{\mathbf{x}}; \mathbf{x}; \mathbf{w}]$ with $\mathbf{u}_0^{(k)}$ being the values of $\dot{\mathbf{x}}$, $\mathbf{x}$ and $\mathbf{w}$ at time $t = 0$ and iteration k.

Being initialized by an initial guess $\mathbf{u}_0$, Newton's method is basically an iteration over the following three steps:

1. Calculate a direction $\mathbf{u}_0^{(0)}$ by solving

$$\mathbf{J}\left(\mathbf{u}_0^{(k)}\right)\Delta\mathbf{u} = -\mathbf{F}_0\left(\mathbf{u}_0^{(k)}\right) \tag{3}$$

   where $\mathbf{J}\left(\mathbf{u}_0^{(k)}\right)$ and $\mathbf{F}_0\left(\mathbf{u}_0^{(k)}\right)$ are the Jacobian and the residual calculated at the current iterate k.

2. Update $\mathbf{u}_0^{(k+1)}$:

$$\mathbf{u}_0^{(k+1)} = \mathbf{u}_0^{(k)} + \mu\Delta\mathbf{u} \tag{4}$$

where $0 < \mu \leq 1$ is a parameter that is used to increase the convergence radius, for example using linesearch [5]. If $\mu = 1$ then the method is Newton's classical method.

3. Check for convergence, and if the stopping criteria are fulfilled, return the result.

This paper will, as the title suggests, improve Newton's method to suit it better to initializing models written in Modelica. The improvement is focused on the first of the three steps constituting Newton's method, the solving of equation (3). Two issues are treated:

- An initial guess sometimes results in a singular Jacobian, so that the linear equation system no longer has a unique solution or a solution at all. In these cases a special regularization procedure has to be applied before the linear system can be numerically solved by e.g. LU factorization. This will be discussed in this paper.
- For large Modelica models the matrix $\mathbf{J}\left(\mathbf{u}_0^{(k)}\right)$ is sparse, a justification for this will be given later. In this case it is interesting to look at representing the matrix $\mathbf{J}\left(\mathbf{u}_0^{(k)}\right)$ in a sparse format and using a sparse linear solver such as SuperLU [12]. This paper will discuss whether, or when, such an implementation is advantageous or not.

2 JModelica.org and initialization

The initialization problem is generated in JModelica.org upon compilation. The system to be solved at initialization is (1) with additional initial equations supplied by the user. The functions associated with the initialization system, such as $\mathbf{F}_0$ and its Jacobian, are supplied by the JMI interface [20].

JModelica.org sets up the DAE system in its index-1 form, a form in which differential variables, x and algebraic variables w can be clearly distinguished. The system contains equations describing all derivatives and algebraic variables. It will then have $n_x + n_w$ equations resulting in an underdetermined system. Thus n_x additional equations are needed [16].

The assumption of (1) being of index 1 can be justified by saying that if a DAE of higher index is encountered, an algorithm such as the one described in [17] is applied to reduce the problem back to a DAE of index 1.

The additional n_x equations can be supplied by the user as fixed start values and initial equations. Adding this information to the System (1), the initilization System (2) is obtained. This is done by adding all equations defined as initial equations as well as an equation of the kind (5) for each variable x_i with a modifier such as `(start = x_0, fixed = True)`.

$$0 = x_i - x_0 \tag{5}$$

If the user has supplied enough additional data, the System (2) can be generated. If, however the user supplies too much information the system becomes overdetermined and the compiler will give an error message. If, on the other hand, not enough information is supplied the system becomes underdetermined. In this case the compiler will try to add information, such as setting some variables to `fixed = true`, making the system well defined. This is accomplished by applying an algorithm to compute a maximal matching between variables and equations. For this purpose, an implementation of the Hopcroft Karp matching algorithm, [11], is employed. If unmatched variables are detected, the corresponding `fixed` attributes are set to `true`, and thereby balancing the system.

3 Implementation

3.1 Overview

The implementation of the algorithm reported spans multiple packages, written in two different programming languages: Python and C. A third language, Cython [4], is used so packages written in the two different languages can communicate with each other. The algorithm basicly consists of four packages, JModelica.org, Assimulo, KINSOL and an external linear solver (cf. Section 3.2) implementing a regularization method and using SuperLU.

The JModelica.org project is the biggest part and consists of code written in multiple languages, the part of JModelica.org used in this thesis is however entirely coded in Python

Assimulo is a package written in Python using

Cython to interface functionality from the SUNDIALS suite, for example KINSOL [4, 1].

Finally KINSOL and SuperLU are two packages entirely written in C. An overview of how these packages interact is presented in Figure 1.

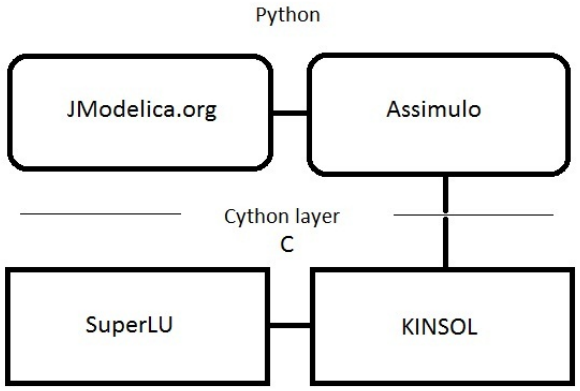

Figure 1: Overview of the packages involved in this paper and how they interact.

Model data, such as evaluation of $\mathbf{F}_0$ and its Jacobian, are obtained by the JMI interface in the JModelica.org part of Figure 1. The data is passed to Assimulo who calls KINSOL using Cython. In KINSOL the data is used to create the system (3) which is solved by the external linear solver, called SuperLU in Figure 1.

3.2 KINSOL

The non-linear solver implemented in the initialization algorithm reported is based on KINSOL from the SUNDIALS suite [5]. Although this report focuses on regularization and sparse solvers, some necessary theory on KINSOL has to be reviewed to allow discussion of the implementation of regularization and SuperLU.

KINSOL is a solver of systems of non-linear equations which implements a modified Newton method where the Jacobian is only evaluated when the solution progresses to slow or a certain number of iterations is exceeded [5]. This is to speed up the solution of the nonlinear system since Jacobian evaluations are expensive. The Jacobian can either be calculated by finite differences or have to be supplied as a function by the user.

A regularization method and SuperLU are implemented in KINSOL as an external linear solver. An external linear solver is called by KINSOL to solve (3) and must implement a set of functions. The two functions that are of interest in the implementation discussed here are the *setup* and *solve* functions.

- The *setup* function is called whenever KINSOL needs to (re)evaluate the Jacobian. LU factorization is preferably performed in this function.
- The *solve* function uses the data from the last call to *setup* to solve the linear system.

4 Regularization

When in a step, say in the k^{th} step and the Jacobian is singular, the linear system (3) has no solution or its solution is not unique. Thus a different algorithm for determining the Newton increment Δu has to be used.

We require that Δu is a descent direction and a solution of the following regularized normal equations

$$\begin{aligned}\left(\mathbf{J}\left(\mathbf{u}_0^{(k)}\right)^T\mathbf{J}\left(\mathbf{u}_0^{(k)}\right)+\lambda_k\mathbf{I}\right)\Delta\mathbf{u}\,(h)\\ =-\mathbf{J}\left(\mathbf{u}_0^{(k)}\right)^T\mathbf{F}_0\left(\mathbf{u}_0^{(k)}\right)\end{aligned}\tag{6}$$

with $\lambda_k > 0$.

Here, the matrix $\left(\mathbf{J}\left(\mathbf{u}_0^{(k)}\right)^T\mathbf{J}\left(\mathbf{u}_0^{(k)}\right)+\lambda_k\mathbf{I}\right)$ is positive definite with eigenvalues in $\left[\lambda_k, \lambda_k+\left\|\mathbf{J}\left(\mathbf{u}_0^{(k)}\right)\right\|_2\right] \subset \mathbb{R}$, [18].

We select λ_k in accordance to a strategy used, when implementing the Levenberg-Marquardt method (LM) for solving an overdetermined nonlinear equations systems [7, Ch. 10], by setting

$$\lambda_k := \min\left(1, \left\|\mathbf{J}\left(\mathbf{u}_k\right)^T\mathbf{F}_0\left(\mathbf{u}_k\right)\right\|\right)\tag{7}$$

Note, in the DAE initialization process this regularization technique is required in a single, exeptional step only, while the overall process remains classical Newton iteration, based on solving regular linear systems.

4.1 Implementation

As mentioned in Section 3, regularization is implemented in an external linear solver to KINSOL. This is done so that when the LU factorization in

the *setup* function fails due to the Jacobian being singular, the regularization algorithm is called.

When the regularization is called, the regularization parameter λ_k is given by (7). Secondly, the regularized matrix $\left(\mathbf{J}\left(\mathbf{u}_0^{(k)}\right)^T\mathbf{J}\left(\mathbf{u}_0^{(k)}\right)+\lambda_k\mathbf{I}\right)$ is calculated and stored as the problem Jacobian. A flag is also set telling the linear solver that the problem is currently regularized.

When the *solve* function is called it will continue as usual if the regularization flag is not set. If the flag is set however, a new right hand side corresponding to the right hand side of (6) is calculated and solved for instead of the ordinary $-\mathbf{F}_0\left(\mathbf{u}_0^{(k)}\right)$.

With the regularization parameter calculated as described, there is still a problem of the Jacobian being singular at the solution. The strategy chosen is only valid for overdetermined systems if the Jacobian is regular at the solution [7, Ch. 10]. To see what effects this presents on the DAE initialization, the algorithm has been tested on the following problem (8).

$$\begin{aligned} 0 &= x^2 \\ 0 &= y^2 \end{aligned} \tag{8}$$

Problem (8) has the solution $x = y = 0$ where its Jacobian (9) is singular.

$$\begin{bmatrix} 2x & 0 \\ 0 & 2y \end{bmatrix} \tag{9}$$

The algorithm converges, albeit slowly (29 iteration when starting at $x = y = 1.0$), to the solution $x = y = 0.0018$. The stopping criteria attained in this case is the norm of the residual being smaller than a given tolerance ϵ, in this case set to $6\cdot10^{-6}$.

Hence the problem of a singular Jacobian at the solution slows down the algorithm but it does not cause it to crash, as long as the tolerance is not set too small. There are methods discussed in [10] handling this problem which may be included in a later implementation.

4.2 A simple example

To test if regularization indeed works, a simply constructed system with poorly chosen initial values is initialized. The example contains two states x, and y as well as an algebraic variable w and is written as follows:

```
model SingularTest
  Real x ;
  Real y (start = 1, fixed = true);
  Real w (start = 2, fixed = true);
equation
  der(x) = x^2 - y;
  der(y) = x^2 + z^2;
  0 = w - x^2 -y;
end SingularTest;
```

In the initialization problem, the consistent values of the two states y and x, their derivatives and the algebraic variable w are solved for. The sought DAE equations are the three equations in the `equation` block. Added to these are the two equations corresponding to fixed start values. Five variables and five equations make up the well defined initialization system (10).

$$\begin{aligned} 0 &= x^2 - y - \dot{x} \\ 0 &= x^2 - z^2 - \dot{y} \\ 0 &= w - x^2y - y \\ 0 &= y - 1 \\ 0 &= w - 2 \end{aligned} \tag{10}$$

At the initial guess given in the Modelica code (the variable x is without a start guess and is given the default guess zero), the system (10) has the Jacobian (11):

$$\begin{pmatrix} 1 & 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 & 4 \\ 0 & 0 & 0 & -1 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{11}$$

(11) is singular and so is $\mathbf{J}^T\mathbf{J}$, $\mathbf{J}^T\mathbf{J}+h^2\mathbf{I}$ is however regular.

Trying to initialize this model in JModelica.org yields the regularization algorithm to be called followed by the Jacobian becoming regular and Newton's method proceeding as usual. Hence the regularization implemented succeeds in handling the singular Jacobian.

5 Sparse solvers

Another aspect taken into account when solving (3) is the structural properties if the Jacobian. When solving large systems, the solution of the system (3) can become very costly and slow due to the size of the Jacobian. But although the Jacobian is big in size it is not necessarily dense. A matrix, and the corresponding linear system, is

said to be sparse if there are many zero entries and only a few entries different from zero and this is something that can be exploited.

Another approach to exploiting model structure commonly employed in Modelica tools is based on the Block Lower Triangular (BLT) transformation, in which the system of equations is decomposed into a sequence of smaller equation systems, see e.g. [8, 9].

5.1 Sparse Jacobians in Modelica models

In the case of a Jacobian, the number of non-zero entries in row i corresponds to the number of variables upon which the function i in the system (2) is dependent. Since each function in a Modelica model of size n normally depends on about five to ten variables, the number of non-zero entries will grow linearly while the size of the Jacobian will grow quadraticaly for each added function. In addition, many equations in (2) come from initial values set by a `fixed = true` which only depend on one variable. The hypothesis stated here is that the Jacobian will get more sparse as the Modelica model itself gets bigger.

5.2 SuperLU

Since the Jacobians are sparse, it is interesting to look at sparse solvers for solving (3). The solver investigated in this paper is SuperLU, a fast LU-factorization algorithm optimizing memory usage [12]. The SuperLU solver will also be coupled with regularization to be able to handle singular Jacobians.

SuperLU is implemented, similar to the regularization method, as an external linear solver. Since JModelica.org has support for sparse Jacobians through the JMI interface [20], the implementation is similar to the dense case. The function calculating the sparse Jacobian is wrapped in Cython [4] and passed to KINSOL instead of the dense Jacobian. In this case the Jacobian given by the JMI interface is given in coordinate or triplet format, each non zero element is stored as the value along with the row and column number. The format required by SuperLU is *Compressed Column format* or *Harwell-Boeing format* where the columns are stored in one array, their row numbers in one array and the index of when the column changes in a third array [13]. This requires the Jacobian to be reformated before passing to SuperLU, which is performed by `scipy`. The computational effort for this transformation grows linearly with the size of the problem [6].

In the external linear solver, the methods used for LU factorization are called in the *setup* and the solving routines are called in the *solve* function. Regularization is also implemented in the same fashion as in the dense solver but with sparse matrices.

6 Results

6.1 Regularization

As mentioned briefly in the end of Section 4.2, the regularization algorithm succeeds with the constructed example presented there. A model of a distillation column, `jmodelica.examples.distillation`, from the JModelica.org distribution, is a model with a singular Jacobian at the initial guess supplied in the Modelica file. When solving the initialization problem with KINSOL coupled with an ordinary linear solver, the solver fails, stating that the Jacobian could not be LU-factorized. When a regularized linear solver, like the ones described in Sections 4.1 and 5.2, are used however, one regularized step is taken and KINSOL then converges to the solution without having to perform another regularization step.

6.2 SuperLU

To test the effiency of the initialization algorithm with SuperLU, several Modelica models have been initialized with the sparse and the dense initialization algorithm. The initialization has been timed multiple times and a mean value is calculated. The mean values and medians of the times are later compared to decide which algorithm is faster. The tests have been performed on a Intel Core 2 Duo T5870 processor under 32 bit Windows 7 Professional.

To test if the initialization algorithm is faster using SuperLU instead of a dense linear solver two series of non-linear systems have been compared. From [2] the problem series *Broyden* and *Moraeux* are problems concerning constrained optimization but can be seen as a non linear root finding problem. A simple script `AtoM.py` is implemented to translate the models, supplied in the

AMPL format .mod, to Modelica for later treatment by JModelica.org. These problem series offer similar systems of different size. In Figures 2 and 3 the speedups of both the Broyden and Moreaux problems are plotted against the number of variables of the systems. In Figure 2 the speedup of the total time is plotted while Figure 3 plots the speedup of the total time except the time spent evaluating system functions and Jacobians. Here speedup means the time measured with dense solver divided by time measured with sparse solver, ei.e. how many times faster the sparse solver is than the dense solver. It should be noticed that computation of Jacobians in JModelica.org used for the benchmarks is slow, due to limitations in the CppAD package, [3], with regards to sparse Jacobians. Therefore, we focus on comparison of the time spent in KINSOL in the cases of sparse versus dense linear solvers.

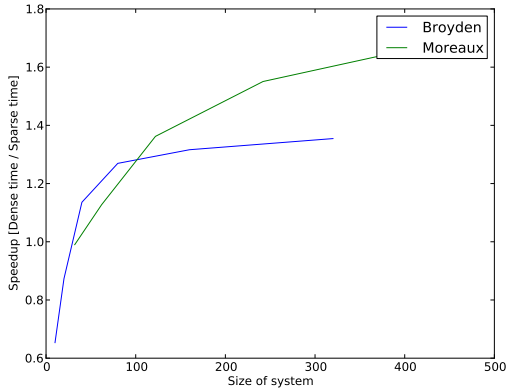

Figure 2: Speedup of the total times for the Broyden and Moreaux problems.

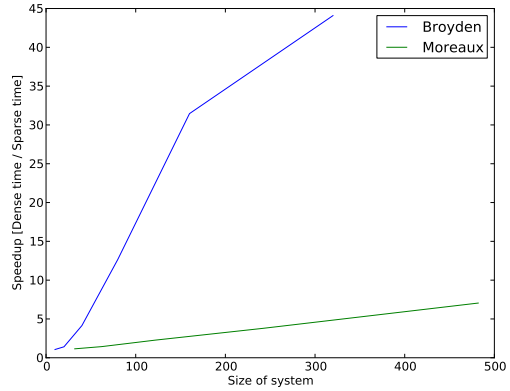

Figure 3: Speedup of the Broyden and Moreaux problems not counting time in fevals and jevals.

In Tables 1 and 2 the data from the test runs on the Broyden and Moreaux preoblems are presented. The data consists of the time spent in total is presented (tot) along the time spent in KINSOL and linear solver without evaluations of Jacobians and system functions (KIN), the time spent on evaluating the residual and Jacobian (Evals) and the number of non-linear iterations required (iters). The data is scaled by the total time of the dense solver to simplify comparison.

Table 1: Times measured for the Broyden problems.

	Broyden	10	40	80	320
	size	10	40	80	320
tot	Dense	1.0	1.0	1.0	1.0
	Sparse	1.530	0.880	0.787	0.738
KIN	Dense	0.452	0.249	0.244	0.243
	Sparse	0.425	0.060	0.019	0.006
Evals	Dense	0.548	0.751	0.756	0.757
	Sparse	1.105	0.820	0.768	0.732
iters	Dense	17	60	112	137
	Sparse	17	60	112	137

Table 2: Times measured for the Moreaux problems.

	Moreaux	10	40	80	160
	size	32	122	242	482
tot	Dense	1.0	1.0	1.0	1.0
	Sparse	1.010	0.734	0.645	0.584
KIN	Dense	0.666	0.495	0.457	0.435
	Sparse	0.580	0.218	0.121	0.062
Evals	Dense	0.334	0.505	0.543	0.565
	Sparse	0.430	0.516	0.524	0.522
iters	Dense	38	112	123	137
	Sparse	38	112	123	137

The models tested in Tables 1 and 2 are not models originally written in Modelica but rather optimization benchmarks. To test how the initialization algorithm using a sparse solver behaves when used on 'real' Modelica models, the same test performed in Tables 1 and 2 are performed on some models with different sizes in Table 3.

- CSTR: an example from the JModelica.org package describing two continously stirred tank reactors in series.
- DIST: an example from the JModelica.org package already mentioned in Section 6.1.

- CoCy: a Modelica model describing a combined cycle power plant initialized at full load.

Table 3: Times measured for the Modelica models.

	Model	CSTR	Dist	CoCy
	size	15	99	150
tot	Dense	1.0	1.0	1.0
	Sparse	1.112	0.599	0.635
KIN	Dense	0.645	0.552	0.426
	Sparse	0.610	0.115	0.112
Evals	Dense	0.355	0.448	0.574
	Sparse	0.502	0.484	0.523
iters	Dense	10	24	34
	Sparse	10	24	34

6.3 Sparsity of Jacobians

It is also interesting to take a look at the sparsity of the systems to put the results obtained in Section 6.2. In Table 4 the sparsity of the systems, that is the sparsity of the system Jacobian, timed in Table 1 and 2, are presented.

Table 4: Sparsity measured in the percentage of elements differnet from zero in the Jacobian.

Model	10	20	40	80	160	320
Broyden	54.0	31.0	16.5	8.5	4.31	2.17
Moreaux	8.98	4.73	2.43	1.23	0.62	-

In Table 5 the sparsity of the Modelica models timed in Table 3 are presented supporting the hypothesis of bigger models being more sparse..

Table 5: Sparsity measured in the percentage of elements differnet from zero in the Jacobian.

Model	Size	Sparisty
CSTR	15	24.0
DIST	99	2.67
CoCy	150	1.75

7 Conclusions

The regularization method is handling singular Jacobians at initialization. So far, no models, supported by JModelica.org, have caused the initialization algorithm based on regularization to stop due to a singular Jacobian. A problem with a singular Jacobian at the solution is however solved slower, as shown in the end of Section 4.1. Pan and Fan [10] proposes techniques to handle this problem that may be used in a later implementations.

Table 5 imply that larger Modelica models are more sparse than smaller Modelica models, thus supporting the hypothesis stated in section 5.1 of Modelica models getting more sparse as they grow in size.

Regarding sparsity, Figures 2, 3 and Tables 1, 2 and 3 imply that the problems are initialized faster with the sparse version of the initialization algorithm. Due to CppAD slowing down the evaluations of Jacobian, the times spent in KINSOL are compared instead of the total time.

When applied to the Modelica models in Table 3, the sparse version solves the bigger problems (of size $n \approx 100$ or bigger) around 4-5 times faster than the dense version. The bigger benchmarks from the Broyden and Moreaux series show an even bigger speedup, Broyden320 is for example solved 40 times faster. For smaller model, like the model of the two stirred tank reactors, the organizational effort of SuperLU and the model being less sparse, outweighs the advantages and the two methods are equal.

In the benchmarks presented here, the time for evaluating Jacobians outweights the time spent in KINSOL, especially if SuperLU is employed. This is due to the fact that the package used for generation of Jacobians has weak support for computation of sparse deriviatives. This deficiency will be adressed in future versions of JModelica.org.

In conclusion, the sparse version of the initialization algorithm is advantageous when applied to bigger models. For smaller models however, the two version performs equally. However, the slow evaluation of sparse Jacobians make the dense solver a better choice for smaller models.

References

[1] The assimulo homepage. `http://www.jmodelica.org/page/199`.

[2] The coconut benchmark: Library 3 constraint satisfaction test problems. `http://www.mat.univie.ac.at/~neum/glopt/coconut/Benchmark/Library3_new_v1.html`.

[3] The cppad webpage. `http://www.coin-or.org/CppAD/`.

[4] Stefan Behnel, Robert Bradshaw, Dag Sverre Seljebotn, and other contributors. Cython c-extensions for python - homepage. `http://www.cython.org/`.

[5] Aaron M. Collier, Alan C. Hindmarshand Radu Serban, and Carol S. Woodward. *User Documentation for kinsol v2.6.0*. Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, May 2009.

[6] The SciPy community. *SciPy Reference Guide*, 0.9.0.dev6665 edition, October 2010.

[7] J.E. Dennis and R.B. Schnabel. *Numerical methods for unconstrained optimization and nonlinear equations*. Prentice Hall, 1983.

[8] Hilding Elmqvist. *A Structured Model Language for Large Continuous Systems*. PhD thesis, Department of Automatic Control, Lund University, Sweden, may 1978. TFRT-1015.

[9] Jan Eriksson. A note on the decomposition of systems of sparse non-linear equations. *BIT Numerical Mathematics*, 16(4):462–465, 1976. 10.1007/BF01932730.

[10] Jinyan Fan and Jianyu Pan. A note on the levenberg-marquardt parameter. *Applied Mathematics and Computation*, 207:351–359, 2009.

[11] John E. Hopcroft and Richard M. Karp. An $^{2/5}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4):225–231, 1973.

[12] X. S. L. W. Demmel James W. Demmel, John R. Gilbert, Stanley C. Eisenstat, John R. Gilbert, Xiaoye S. Li, and Joseph W. H. Liu. A supernodal approach to sparse partial pivoting. *SIAM J. MATRIX ANAL. APPL*, 20(3):720–755, 1999.

[13] Xiaoye S. Li James W. Demmel, John R. Gilbert. *SuperLU Users Guide*.

[14] K. Levenberg. A method for the solution of certain nonlinear problems in least squares. *Quart. Appl. Math.*, 2:164–166, 1944.

[15] D.W. Marquardt. An algorithm for least-squares estimation of nonlinear inequalities. *SIAM J. Appl. Math.*, 11:431–441, 1963.

[16] S.E Mattson, H. Elmqvist, M. Otter, and H. Olsson. Initialization of hybrid differential-algebraic equations in modelica 2.0. In *Second International Modelica Conferencem, Proceedings*, pages 9–15. The Modelica Association, March 2002.

[17] Sven Erik Mattsson and Gustaf Söderlind. Index reduction in differential-algebraic equations using dummy derivatives. *SIAM J. Sci. Comput*, 14(3):677–692, May 1993.

[18] Arnold Neumaier. Solving ill-conditioned and singular linear systems: A tutorial on regularization. *SIAM Review*, 40(3):636–666, September 1998.

[19] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer, 2006.

[20] Johan Åkesson, Karl-Erik Årzén, Magnus Gäfvert, Tove Bergdahl, and Hubertus Tummescheit. Modeling and optimization with optimica and jmodelica.org - languages and tools for solving large-scale dynamic optimization problem. *Computers and Chemical Engineering*, 34(11):1737–1749, nov 2010.

Acknowledgements

The authors would like to acknowledge the kind assistance from Francesco Casella in providing the Combined Cycle benchmark model used in the paper. This work was partially funded by the Swedish funding agency Vinnova under the grant program "Forska and Väx".