Installation and usage instructions for the CVode Wrapper

Joep Vanlier
WH 3.2a
(+31) 40 247 5573
j.vanlier@tue.nl

October 22, 2012

Disclaimer

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

1 About this package

This package provides a parser and compilation environment to be able to generate compiled MEX files for the simulation of Ordinary Differential Equation (ODE) models. It was written in order to speed up large scale analyses of models based on ODE models.

2 Installation

Depending on what purpose you wish to generate MEX files for, you will need to acquire different packages. Currently the package offers support for three different compilers, each with their own (dis)advantages. During our benchmarks, fastest simulations were obtained using Microsoft Visual C++ 2008 (MSVC) and GCC/Cygwin. If the aim is to run the MEX files on a Linux machine, GCC is the desired compiler. Steps marked with [G] correspond to installation for use with GCC, steps marked with [M] MSVC and steps marked with [L] LCCWin32. Steps with the prefix [A] are required for all compilers.
If at any time during installation you are doubting what you are doing all this for, look at figure 1.

Step 1. Extracting the package

The first required step is to extract the parser (*parser.zip*) in a directory. The easiest place to do this is in your modeling directory. Remember to check extract with full path info. The directory parser shall henceforth be referred to as $PARSER$.

Step 2. Obtaining the required dependencies

[L] For use with LCCWin32 no additional dependencies are required since it is shipped with MATLAB.

[M] To be able to compile with Microsoft Visual C++ 2008, you will need to download this from the Microsoft website. At the time of this writing the link was: `http://www.microsoft.com/express/Downloads/` but this may be subject to change. The required (free) version is Microsoft Visual C++ 2008 Express Edition. Note that in 64-bit versions of MATLAB this compiler is no longer supported. When working on a 64-bit system, check the MathWorks website on how to set up MSVC as a MATLAB compiler and install all the appropriate SDKs.

[G] To be able to compile using GCC, two packages are required namely gnumex and cygwin. Gnumex can be downloaded from `http://sourceforge.net/projects/gnumex/` and Cygwin

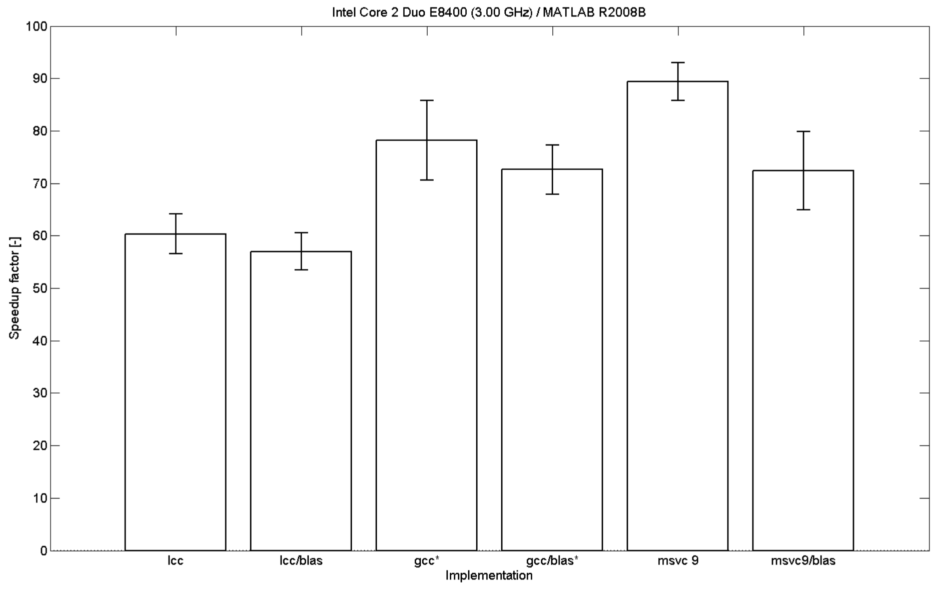

Figure 1: Speed up relative to MATLAB ode15s

from: `http://ptolemy.eecs.berkeley.edu/ptolemyII/ptII4.0/cygwin.htm`.
Note that you require *cygwinDevel.exe* which ships with GCC version 3.2 (since any newer version crashes MATLAB).

Step 3. Installing the dependencies

[A] Write down the root directory of where MATLAB is installed (e.g. c:\MATLAB2008b). This directory shall henceforth be referred to as $MATLABROOT$.

[L] LCCWin32 requires no extra dependencies.

[M] Run *vcsetup.exe*, and follow the installation instructions. The root directory of the Visual Studio (e.g. C:\Program Files\Microsoft Visual Studio 9.0) shall be referred to as $VSTUDIO$. Also locate where .NET is installed on your machine (e.g. C:\WINDOWS\Microsoft.NET), this directory shall be referred to as $DOTNET$.

[G] Important is that during Cygwin installation you check all options under Devel for installation. Note that gcc and make are the absolute minimum requirements to be able to build MEX files. The directory where you install cygwin shall henceforth be referred to as $CYGWIN$. For GCC you also need to install gnumex. Extracting the zip file in a subdirectory named gnumex in $PARSER$. Subsequently start MATLAB and go to the folder where you extracted the gnumex files. Type gnumex and hit return. A window should appear. To be able to get this toolbox to work, you need to set the Cygwin root directory to $CYGWIN$. Select C\C++ as language for compilation and optimisation level -O3. Select generate mex dll and environment \linking type Cygwin. Then set the mex options file to create to $GNUMEX$\mexopts.bat. This path shall henceforth be referred to as $MEXOPTS$. Click make options file. This might take a while. After completion, click Exit. Verify that the file was created typing ls mexopts.bat.

Step 4. Setting up the toolbox

Note that the following steps have to be (partially) reperformed whenever you wish to switch compiler!

[A] Open MATLAB and type "cd $PARSER$\CVode"' to go to the directory where the toolbox is extracted. Here you type edit chooseCompiler. Set compiler, to the compiler you wish to use (1=LCCWIN32, 2=MSVC, 3=GCC). Depending on which compiler you choose, there are various options to set. Subsequently we need to set various paths, so the toolbox can find all required files. First directory is the parser directory. Set parserDir to the appropriate path $PARSER$. Note that it is important that you end this line with a backslash.

[L] For LCC-WIN32, you can set optional compiler flags in the block denoted by LCC-Win32. This is not required however.

[M] For MSVC go to the section denoted by Microsoft Visual C++ 2008. Here you need to appropriately set vsroot and netroot to $VSTUDIO$ and $DOTNET$. Again, you can optionally set some compiler flags. To compile with MSVC type "mex -setup" at the MATLAB command prompt, select y and then select the MSVC compiler. On earlier versions of MATLAB, this compiler may not be listed. If that is the case, copy all files starting with msvc90 from $NEWER-MATLABROOT$\bin\win32\mexopts to $MATLABROOT$\bin\win32\mexopts and repeat the procedure.

[G] For GCC, we need to set three directories. Namely the directory where *Cygwin1.dll* can be found, which is usually $CYGWIN$\bin\.

Once all options are set, click save. All should be appropriately set to be able to compile CVode now. Close the MATLAB m-file editor and type setupCVode. Depending on your choice of compiler you will see different output.

3 Setting up the RHS file for the parser

First of all, in order to work with the parser, one needs to adhere to specific rules regarding the MATLAB code. First of all, vectors, matrices, cell arrays and complex numbers are (currently) not supported. Neither are powers. In order to take powers use $pow(a, b)$ for a^b. If one knows that parameter b is an integer, it is preferable to use intPow, since this routine will be faster in the compiled version. The maximum and minimum functions are replaced by *maximum* and *minimum* respectively and only deal with one element inputs. The exp command works as expected, except for the fact that complex numbers are not supported. One thing to be careful of is to avoid integer division. In C, when you divide 2.5/2 for example results in a conversion to an integer. This is why you should write expressions such as this as 2.5/2.0.
To be able to parse a model right hand side file the parser requires knowledge of the model parameters, states, inputs and output variables. This is done by means of a structure called mStruct. mStruct contains four fields of which two are mandatory. The structure fields should be s, p, c and u, referring to states, parameters, constants and inputs. The state, parameter and input fields contain fields named after states, parameters and inputs as referenced in the MATLAB right hand side file. Each of these fields contains a *unique* identifier which denotes its

position in the state/parameter/input vector. Only the state and parameter field are mandatory. The c field contains constants, which are replaced in the M-file by their numeric values.

```
>> mStruct

mStruct =

    s: [1x1 struct]
    u: [1x1 struct]
    p: [1x1 struct]
    c: [1x1 struct]


>> mStruct.s

ans =

          hep_FA: 1
           hep_C: 2
          hep_CE: 3
         hep_GLC: 4
          hep_TG: 5
          ...


>> mStruct.c

ans =

            mwTG: 859.2000
            mvTG: 946.8384
            mwCE: 647.9000
            mvCE: 685.4782
             avg: 6.0221e+023
             ...
```

Each right hand side file should have at least four inputs namely time, the state, the parameters and the inputs. These are also the only vectors that will be available to the right hand side file in the C code. One can specify additional parameters but this is optional and will be ignored when parsing to the C file. Once this structure has been set, the parser can be used to parse the model file. However, there are several options to choose from when it comes to parsing the model to C (see table 1). These options can be set using the command cParserSet. The available choices for the variable solver are shown in table 2.
The stiff solvers in the CVode package are based on Backwards Differentiation Formulas (BDFs). In the case of the dense solver the solution of the linear system at each time step is approximated using Modified Newton iterations where the Jacobian is fixed and often out of date. When using option 3 the linear solver uses an Inexact Newton iteration using the current Jacobian.

Variable Name	**Description**
solver	Specifies the solver to use.
blockSize	Contains an integer value which specifies the size of a single memory block. This option is always mandatory but is only used when only a begin and end time are specified for the solver. It will then allocate an additional chunk of blockSize timesteps once it runs out. The blockSize should ideally be a little larger than the number of time steps the solver produces.
maxConvFail	Maximum number of convergence failures in solving the linear system (in the case of stiff simulation).
minStep	Minimum step size.
maxStep	Maximum number of steps the solver is allowed to take to get to the end time. Once exceeded, results up to that point will be returned.
maxStepSize	Maximum stepsize (use if simulation overshoots points of interest)
debug	Debug mode enabled (0 or 1)? (returns some information when simulation fails).
fJac	Provide CVode with derivatives of the right hand side (0 or 1, avoids internal finite differencing, but does no longer allow the use of if-statements).
aJac	Provide CVode with derivatives of sensitivity equations (0 or 1, does no longer allow the use of if-statements).

Table 1: Parser options

Solver	**Type**	**Name**
1	Stiff	Dense Solver
2	Stiff	Dense Solver with precompiled LAPACK/BLAS (slow for small systems but may be better for large matrices)*
3	Stiff	Scaled preconditioned GMRES*
4	Stiff	Scaled preconditioned Bi-CGStab solver*
5	Stiff	Preconditioned TFQMR iterative solver*
10	Non-Stiff	Adams-Moulton solver Order 1-12

Table 2: Available solvers. *No longer supported

4 An example

To explain how the parser works, we work out a simple example model consisting of a single input, two states and an output. Note that this example assumes that you have managed to compile the library with the setupCVode command and have set your MEX compiler to the appropriate compiler as described.
Here k_1, k_2 and k_3 denote rate constants. Furthermore c_1 is a known constant with value 4. Writing down the equation for the system is straightforward:

$$\dot{x}_1 = u_1 - k_1 [x_1] + k_2 [x_2] \quad (1)$$

$$\dot{x}_1 = k_1 [x_1] - \left(k_2 + \frac{k_3}{c_1} \right) [x_2] \quad (2)$$

The simplest method to implement this would be the following (create this file in the modeling directory (MATLAB_models)): *mymodel1.m*

```
function dx = mymodel1(t, x, p, u)

dx(1) = u(1) - p(1) * x(1) + p(2) * x(2);
dx(2) = p(1) * x(1) - ((p(3)/4) + p(2)) * x(2);

dx = dx(:);
```

Note how the constant is hardcoded in the equations. Additionally, it is important to remember to specify every element of the output vector (in this case dx). **Not specifying an output means that the variable remains uninitialised in the compiled code, leading to undesired model behaviour!** Now, let's proceed. First simulate the model in MATLAB, using MATLAB's native ODE solver ode15s. For now, we shall use an initial condition of $[x_1, x_2] = [3, 5]$ and parameter values $[k_1, k_2, k_3] = [2, 3, 4]$ and as an input a constant value of 1.

```
[ t_MATLAB, y_MATLAB ] = ode15s( @mymodel1, [0:.1:10], [3,5], [], [2, 3, 4], [1] );
```

If one were to use a right hand side such as this, one would have to create the structure required by the parser as follows.

```
    mStruct.s.x1 = 1;
    mStruct.s.x2 = 2;
    mStruct.p.k1 = 1;
    mStruct.p.k2 = 2;
    mStruct.p.k3 = 3;
    mStruct.u.u1 = 1;
```

Go to the directory one level up from the directory parser and add the required paths using addPaths. Next, invoke the lines:

```
    convertToC( mStruct, 'MATLAB_models\mymodel1.m' );
    compileC( 'odeC' );
```

Make sure the output filename (in this case odeC) does not exist in the directory yet (it will not overwrite it if it is there). If it is, delete it. After running a MEX file MATLAB keeps the file in use, resulting in the operating system refusing to delete it. When this happens type *clear mexname* (in this case odeC) in the workspace before trying to delete the file. Only compile MEX files after you have made sure the file does not exist.

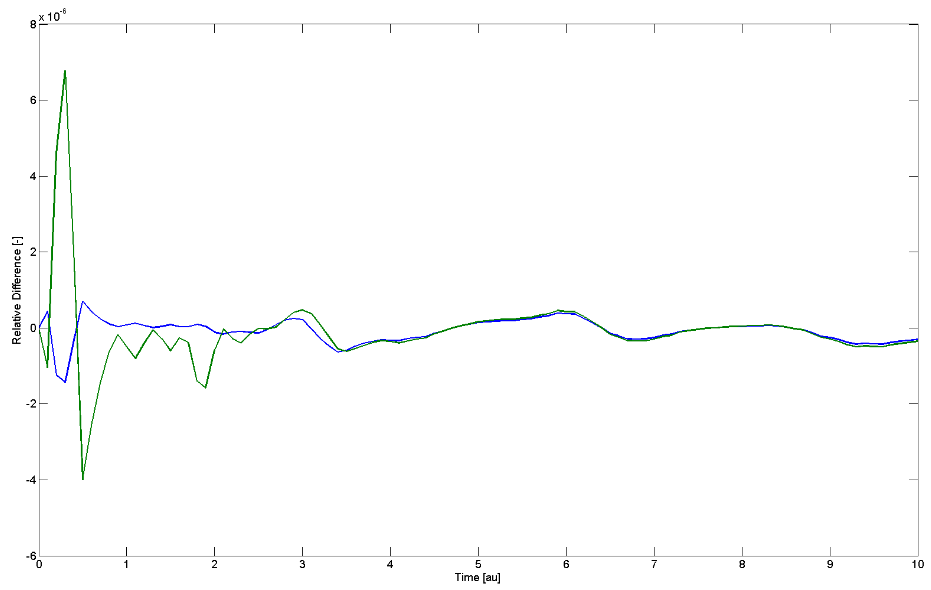

Figure 2: Comparison between Ode15s and CVode Dense solver

To compare the results between the two models, we can make a quick file.

```
options = odeset( 'RelTol', 1e-6, 'AbsTol', 1e-8 );
[t y_MATLAB] = ode15s( @mymodel1, [0:.1:10], [3,5], options, [2, 3, 4], [1] );
[t y_odeC]   = odeC( [0:.1:10], [3,5], [2, 3, 4], [1], [ 1e-6 1e-8 10 ] );

plot( t, ( y_MATLAB - y_odeC.' ) ./ y_MATLAB );
```

Which shows that the difference between the two methods is rather small (order of 10^{-6} for this specific example, see figure 2). However, for larger models this method of referencing states and variables can become quite error prone. An alternative way to specify the *same* model is shown on the next page.

model2.m

```
function dx = mymodel2(t, x, p, u, myStruct)

p1 = p( myStruct.p.k1 );
p2 = p( myStruct.p.k2 );
p3 = p( myStruct.p.k3 );
x1 = x( myStruct.s.x1 );
x2 = x( myStruct.s.x2 );
u1 = u( myStruct.u.u1 );
c1 = myStruct.c.c1;

dx( myStruct.s.x1 ) = u1 - p1 * x1 + p2 * x2;
dx( myStruct.s.x2 ) = p1 * x1 - addTwo((p3/c1), p2) * x2;

dx = dx(:);
```

addTwo.m

```
function y = addTwo( in1, in2 )

y = in1 + in2;
```

Of course, in this example, the advantage of using this method to specify ones model does not seem to be very advantageous, but when states, parameters and constants have more realistic names this will become more evident. Note that the structure references don't necessarily have to be at the start of the file. Be warned however that currently the framework only supports functions that only have one output variable. To turn this code into a compiled ODE file, we have to take a few steps. An example of how to use the parser is shown on the next page.

```
addpath( '..' );

mStruct.s.x1 = 1;
mStruct.s.x2 = 2;
mStruct.p.k1 = 1;
mStruct.p.k2 = 2;
mStruct.p.k3 = 3;
mStruct.u.u1 = 1;

mStruct.c.c1 = 1;

%% Custom options
odeInput = 'MATLAB_models\\mymodel2.m';
dependencies = { 'MATLAB_models\\addtwo.m' };

options = cParserSet( 'blockSize', 5000 );
convertToC( mStruct, odeInput, dependencies, options );
compileC( 'mymodel2C' );

options = odeset( 'RelTol', 1e-6, 'AbsTol', 1e-8 );

[t y_MATLAB] = ode15s( @mymodel2, [0:.1:10], [3,5], options, ...
                                    [2, 3, 4], [1], mStruct );

[t y_odeC]   = mymodel2C( [0:.1:10], [3,5], [2, 3, 4], ...
                                        [1], [ 1e-6 1e-8 10 ] );

plot( t, ( y_MATLAB - y_odeC.' ) ./ y_MATLAB );
```

Since the parser is located in a subdirectory of the modeling directory (this is for safety purposes), we first add the path of the modeling directory. Subsequently the structure is created. After the structure is set up, we specify the input file name and some dependencies. Note the cell array brackets in the dependency list. These are required! After setting these variables, we set the blockSize of the parser to a different value and subsequently compile, including our new options. As shown, running this model is possible by invoking the command mymodel2 with its respective options. Also note that because of the way the ode m-file is specified now, the m-file version depends on the structure as an additional input now. One final word of warning, it is unwise to use the same filename for the MEX file as for the MATLAB file, because it is poorly defined whether MATLAB will invoke the m-file or the MEX file in that case (apparently which behaviour you get depends on the version of MATLAB).

5 Interpolation

The new parser version also supports interpolation. In many systems biology models, specific inputs are described by datapoints. To meet this demand, interpolators were added to the parser framework. In this example interpolation shall be introduced.
Generate a simple mStruct file:

```
    mStruct.s.x1        = 1;
    mStruct.s.x2        = 2;
    mStruct.p.k1        = 1;
    mStruct.u.u1        = 1;
```

Additionally, we shall define data used for the interpolation data. Note that in order to be able to change the driving function, we define memory for the data to be interpolated in the substructure u. This is done by defining a list of elements for the time as well as the value vector. Since u1 is already defined we start at element 2 and then allocate memory for 6 time points (6 times 2 elements). Note that it is required that this memory does not contain gaps. Splines are also supported, but they require 2 data elements per time point, and also a vector of second derivatives. Note that when defining the second linear function, we start at element 14, since the first linear function already requires multiple elements. It is important to make sure that elements do not overlap!

```
    % Note that you need to allocate elements for both time as well as the
    % actual data.
    nelem  = 6;
    mStruct.u.time1     = 2:2+6-1;
    mStruct.u.linear1   = 2+6:2+2*6-1;

    mStruct.u.time2     = 14:14+6-1;
    mStruct.u.linear2   = 14+6:14+2*6-1;
```

Now let's look at the ODE m-file:

```
function dx = mymodel3(t, x, p, u, myStruct)

p1          = p( myStruct.p.k1 );
testThingy  = u( myStruct.u.u1 );

dx( myStruct.s.x1 ) = interpolate( &u(myStruct.u.time1), &u(myStruct.u.linear1), 6, t, 1 );
  dx( myStruct.s.x2 ) = interpolate( &u(myStruct.u.time2), &u(myStruct.u.linear2), 6, t, 1 );

dx = dx(:);
```

Again, the structure is similar as before. Here p1 and u1 are just a dummy parameter and input element. State 1 and 2 are determined by using a linear interpolation of the data supplied. The interpolation command takes five inputs. A handle to the vector where the time and data are stored, the number of time points, the current time (first input argument of the ODE) and the type of interpolation used. Here 0 denotes stepwise function, 1 a linear function and 2 refers to a spline function.
Subsequently, we set up the regular parser options as before.

```
%% Custom options
odeInput = 'MATLAB_models\\mymodel3.m';
dependencies = {};

options = cParserSet( 'blockSize', 5000 );
convertToC( mStruct, odeInput, dependencies, options );
compileC( 'mymodel3C' );

options = odeset( 'RelTol', 1e-6, 'AbsTol', 1e-8 );
```

Now the model is ready to be simulated. All that's left to do is to define the inputs. Again, note that time and data points are concatenated in the input vector. Note that alternatively one could choose to use parameters for the interpolation data vector. This can be useful in for example an input sensitivity analysis.

```
% Time points for the driving inputs
t = 0:2:10;

% Values at the time points for the driving inputs
u1 = [4,5,6,7,6,9];
u2 = [1,2,3,4,8,9];

% Time points for the simulation
dt      = .01;
tend    = 15;
tPoints = [0:dt:tend];
x0      = [3,5];

% Inputs
inputs  = [1, t, u1, t, u2 ];
```

Finally, we can compute the solution of the ODE (in this case a simple integration of the linear interpolation) and verify our results (see figure 3).

```
[ tz y_odeC ]   = mymodel3C( tPoints, x0, [1], inputs, [ 1e-6 1e-8 10 ] );

% Manually integrate
u1S = interp1( t, u1, tPoints, 'linear' );
u2S = interp1( t, u2, tPoints, 'linear' );
for z = 1 : length( tPoints )
    u1I(z) = dt * trapz( u1S(1:z) ) + x0(1);
    u2I(z) = dt * trapz( u2S(1:z) ) + x0(2);
end

plot( tz, y_odeC.' );
hold on;
plot( tPoints, u1I, 'b.-' );
plot( tPoints, u2I, 'g.-' );
```

```
legend( 'Integrator State 1', 'Integrator State 2', ...
 'Manual State 1', 'Manual State 2' );
```

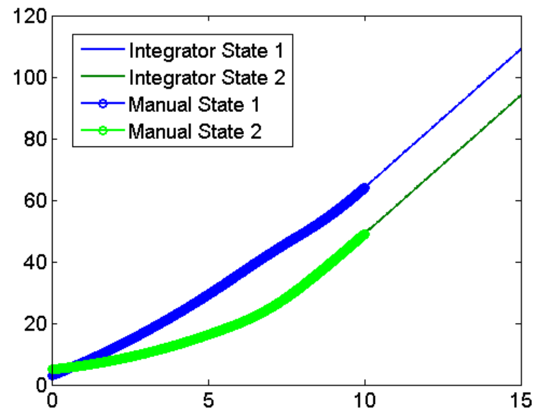

Figure 3: Comparison between manual integration (using the trapezoid rule) and integration using the ODE solver and linear interpolation

6 If statements

Multiline if statements are also supported by the parser (version 5 and up), but one should be careful to stick to strict syntax rules when using them. First of all, the complete conditional (which can be composed of multiple conditions) must be specified on a single line, while every conditional needs to be bracketed separately. Furthermore, brackets are required around the complete conditional statement as well.
An syntactically correct example of an elaborate parsable if statement can be seen in the following ODE file:

```
function dx = mymodel4(t, x, p, u, myStruct)

    dx( myStruct.s.x1 ) = 0;
    dx( myStruct.s.x2 ) = 0;

    if ( ( t > 5 ) && ( t < 10 ) )
        dx( myStruct.s.x2 ) = 1;
    else if ( t > 7 )
            dx( myStruct.s.x1 ) = 1;
            dx( myStruct.s.x2 ) = 2;
        else
            dx( myStruct.s.x2 ) = 0;
        end
    end

    dx = dx(:);
```

7 Sensitivity Analysis

In many cases, it may be desirable to compute the local sensitivity of the model with respect to the parameters and initial conditions. The sensitivity of the solution can be calculated by applying the chain rule of differentiation to the ODE's, resulting in the forward sensitivity equations.

$$\frac{\delta}{\delta t}\left(\frac{\delta y(t)}{\delta p_i}\right) = \frac{\delta f}{\delta y}\frac{\delta y(t)}{\delta p_i} + \frac{\delta f}{\delta p_i} \tag{3}$$

In this sense, the initial condition can be treated as another parameter. Given an initial condition these equations can then be integrated along with the solution of the system. The initial conditions are zero for the initial states and one for the initial conditions (since $\frac{\delta x(0)}{\delta p_i} = 0$ and $\frac{\delta x(0)}{\delta x_0} = 1$). This method is also available in the package, but requires additional parameters in the compiled file.
To include the integration of the forward sensitivity equations, include an additional flag set to one and an additional output used to store the sensitivities.

```
[ t y_odeC, S ]   = mymodel2C( time, initCond, params, input, tols, 1 );
```

Sometimes it is useful to integrate an ODE in parts, in such an event you need to propagate the sensitivities as initial conditions for the sensitivity to the next part of the simulation if you wish to calculate sensitivity information. You can do this by passing a sensitivity vector as an additional argument. *Please note that the values of the supplied initial sensitivity vector will be changed by the MEX file!*

```
[ t y_odeC, S1 ]   = mymodel2C( time, initCond, params, input, tols, 1 );
[ t y_odeC, S2 ]   = mymodel2C( time, initCond, params, input, tols, 1, S1(:,end) );
S   = [S1(:,1:end-1) S2];
```

To show the sensitivity analysis capabilities, you can try the following two simple examples. The first is a comparison with a numerically computed Jacobian.

```
addpath( '..' );
mStruct.s.x1 = 1;
mStruct.s.x2 = 2;
mStruct.p.k1 = 1;
mStruct.p.k2 = 2;
mStruct.p.k3 = 3;
mStruct.u.u1 = 1;
mStruct.c.c1 = 1;

%% Custom options
odeInput = 'MATLAB_models\\mymodel2.m';
dependencies = { 'MATLAB_models\\addtwo.m' };

options = cParserSet( 'blockSize', 5000 );
convertToC( mStruct, odeInput, dependencies, options );
compileC( 'mymodel2C' );

time            = [0:.1:10];
initCond        = [3,5];
params          = [2,3,4];
tols            = [ 1e-6 1e-8 10 ];
input           = 1;

% Stack the parameters and initial condition in a parameter vector
pars            = [ params, initCond ];

% Compute the sensitivities using the forward sensitivity calculations
[ t y_odeC, S ]   = mymodel2C( time, initCond, params, input, tols, 1 );

% Make a routine that computes the model and concatenates the outputs
% (since lsqnonlin only uses vector residuals)
resids = @( pars ) ( reshape( mymodel2C( time, pars(4 : 5), ...
                        pars( 1 : 3 ), input, tols ).', ...
                        length( time ) * 2, 1 ) );

% Set the options of the optimiser in such a way that it does not
% perform any fitting. We are simply exploiting it to calculate a
% numerical Jacobian w.r.t. the parameters and initial conditions
options = optimset( 'MaxIter', 0, 'MaxFunEvals', 0 );
[x,resnorm,residual,exitflag,output,lambda,jacobian] = ...
                        lsqnonlin( resids, pars, [], [], options );

% Reshape the Jacobian
jacobian = reshape( full( jacobian ), length( time ), 10 );

% Compare the two!
figure;
subplot( 1, 3, 1 );
```

```
plot( jacobian );
title( 'Numerical Jacobian' );
subplot( 1, 3, 2 );
plot( S.' );
title( 'Forward sensitivity result' );
subplot( 1, 3, 3 );
plot( jacobian - S.' );
title( 'Difference' );
```

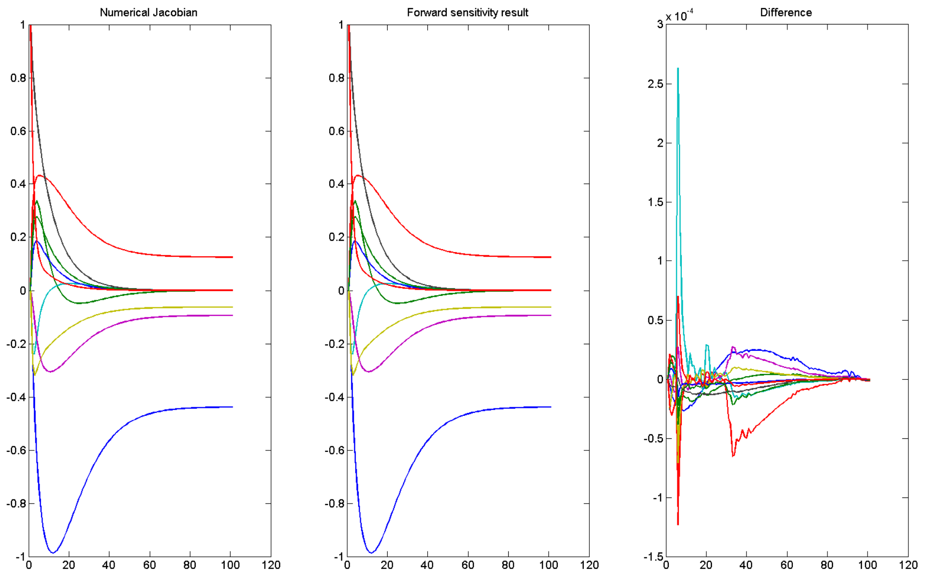

Figure 4: Comparison between computation of the sensitivities by means of finite differencing and solving the forward sensitivity equations

As can be seen in figure 4 the results agree well. The second example is the computation of the sensitivities in two steps rather than one (this can be useful when for example computing a step function).

```
addpath( '..' );
mStruct.s.x1 = 1;
mStruct.s.x2 = 2;
mStruct.p.k1 = 1;
mStruct.p.k2 = 2;
mStruct.p.k3 = 3;
mStruct.u.u1 = 1;
mStruct.c.c1 = 1;

%% Custom options
odeInput = 'MATLAB_models\\mymodel2.m';
dependencies = { 'MATLAB_models\\addtwo.m' };

options = cParserSet( 'blockSize', 5000 );
convertToC( mStruct, odeInput, dependencies, options );
compileC( 'mymodel2C' );

options = odeset( 'RelTol', 1e-6, 'AbsTol', 1e-8 );

time1           = [0:.1:5];
time2           = [5:.1:10];
initCond        = [3,5];
params          = [2,3,4];
tols            = [ 1e-6 1e-8 10 ];
input           = 1;

% Compute sensitivity in one step
[ t y_odeC, S ]   = mymodel2C( [time1 time2(2:end)], ...
                               initCond, params, input, tols, 1 );

% Compute sensitivity in two steps
[ t y_odeC, S1 ]   = mymodel2C( [time1], initCond, params, ...
                                           input, tols, 1 );
% Propagate the initial conditions for y and the
% sensitivities from the previous simulation
[ t y_odeC, S2 ]   = mymodel2C( [time2], y_odeC(:, end), params, ...
                                       input, tols, 1, S1(:,end) );

figure;
subplot( 1, 3, 1 );
plot( S.' );
title( 'Sensitivities Single Step' );
subplot( 1, 3, 2 );
plot( [S1(:,1:end-1) S2].' );
title( 'Sensitivities Two Step' );
subplot( 1, 3, 3 );
plot( S.'-[S1(:,1:end-1) S2].' );
title( 'Difference' );
```

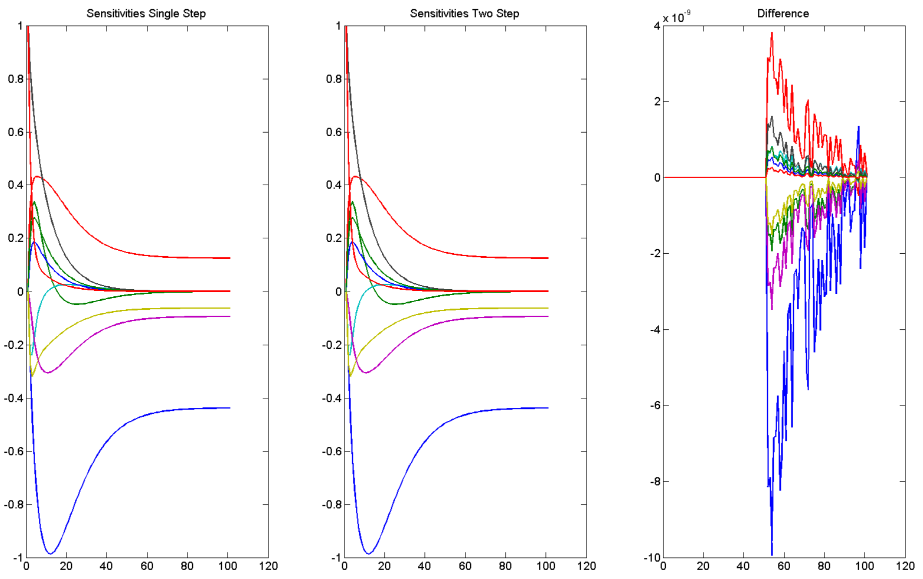

Figure 5: Comparison between computation of the sensitivities in a single or two steps

As can be seen in figure 5, there is a little difference due to roundoff errors but the results agree well.

Computing the solution to the sensitivity equations can be computationally expensive compared to a regular model simulation. Note that there from version 10rev2 there is an option for using the analytical sensitivity equations, which are orders of magnitude faster! However, their computation is not compatible with using if-statements in your m-code. Enabling the computation of the analytic Jacobian can be done by setting the appropriate solver option aJac:

```
options = cParserSet( 'aJac', 1 );
```

See *sens3.m* for an example.

8 Observer Functions

An additional package has been added for computing observer functions for use in data fitting procedures. Though they might seem cumbersome at first, the speed-ups attained usually warrant the initial investment. When dealing with data fitting to observations made on the system, additional parameters are usually required (think of scaling factors, offsets, standard deviations and other parameters associated with data). To incorporate these, new fields can be added in the model structure. These should be incorporated in the subfield o. An example is shown below:

```
mStruct.o.sigma1          = 1;
mStruct.o.sigma2          = 2;
```

If the problem contains unknown initial conditions that should be treated as parameters, an additional field named i is required. This field should contain subfields which hold the indices corresponding to the states to which the initial condition belong. For example, if we were to specify a free initial condition for the state identified by the identifier $x2$, it would look as follows:

```
mStruct.i.x2_0 = mStruct.s.x2;
```

Note that these identifiers can not be used in the observer function expressions. If you would like to refer to the initial condition of a state inside the observer function, please use the regular state notation (where you reference them by time) which is discussed below.
Observations are encoded in a cell array of observer functions. These observer functions are encoded as strings, each corresponding to a different measurement. A single measurement can consist of either one or multiple time points but is limited to vectors. Observer functions require a slightly different syntax than is common in MATLAB. Observations use the identifiers defined in the model structure (for example $sigma1$ as defined earlier). This also places an additional requirement on the use of variables, namely that no double sub fields may exist in the model structure, as they would result in non-unique identifiers. States are referenced by their desired time points. For example, if we would like $x1$ at time point 2, 4 and 6, we would specify $x1([2, 4, 6])$. Note that both the [and] and the (and) brackets are mandatory (even when just considering a single time point) and indicate that we are specifying time points. Standard MATLAB vector notation such as $[2 : 2 : 6]$ is also supported. The parser will simply evaluate what is between the brackets in a prior parsing step. The final step entails the use of data. Since data is usually referenced by indices rather than time, different brackets are used for this purpose. Since cell arrays are not supported in the observer functions we decided to use the { and } brackets for this purpose (example: $data(\{1 : 4\})$). Again, please note that all brackets are mandatory.
An example observer function may look something like this:

```
obs{1} = 'k1*u1*x1([0:.1:1]) * (1+sigma1)';
obs{2} = 'k2*(x1([0,1,2,3,4]) + x2([0,1,2,3,4]) - data({1,2,3,4,5})) / (1+sigma2)';
obs{3} = 'k3*x2([0:.1:5])./x1([0:.1:5])/k2';
```

The observer functions can subsequently be compiled as:

```
[ timePts, parIndices, obsIndices, icIndices ] = ...
observerFunctions( 'test', obs, mStruct, 1 )
```

The observer generating function has the following input arguments:

- Output filename
- Observer structure
- Model structure
- Flag whether Jacobian should be computed

Note that for large systems, constructing this Jacobian can be time consuming. The output variables are:

- A vector of time points which should be used for simulation
- A vector of indices corresponding to model parameters (which should be supplied to the solver)
- A vector of indices corresponding to observational parameters
- A vector of indices used to link initial conditions to their respective parameter indices

An example of a model simulation using these structures is given below:

```
y_odeC = mymodel2D( timePts, [3 pars(icIndices(mStruct.i.x2_0))], ...
pars(parIndices), input, tols );
```

In this example the first initial condition is given a numerical value of 3, while the second initial condition is free (as defined before). Note how the parameters belonging to the model simulation are selected from the full parameter vector *pars* using the *parIndices* structure returned from the objective function generator. The rest of the syntax is as usual. The observer function can subsequently be called as:

```
data = [2,3,7,4,5];
[ observ ] = test( y_odeC, pars, data, input );
```

As shown, the observation function uses the model output, all the parameters and the data as inputs.

8.1 Observer sensitivity

One advantage of using the observer function add-on is that sensitivities with respect to the observer functions can be computed. Especially for sparse Jacobians and stiff systems this can result in speed-ups and increased stability of non-linear least squares algorithms. Note however that if this is desired, both the observer and model functions may both not contain if statements or interpolation functions. The syntax of this is very similar to the syntax mentioned before, except that now the sensitivity matrix obtained from the model simulation has to be passed along to the observer function.

```
[ t y_odeC, S1 ] = mymodel2D( timePts, ...
[3 params(icIndices(mStruct.i.x2_0))], params(parIndices), input, tols, 1 );

[ observ, sens ] = test( y_odeC, params, data, input, S1 );
```

To see this example in action, see the MATLAB example file *obsTest.m*. To see how this can be used in parameter estimation see the file *fitExample.m*.

9 A few hints regarding performance

There are three ways to obtain output from your compiled ODE file.

- *Equilibration mode*: If you supply a single value in the time parameter, the model will run the simulation from $t = 0$ to your given time point. It will then return the solution at that time point. This is the fastest method for simulating such an equilibration.
- *Time series mode*: The second method is to invoke the ODE file with a series of time points, which results in the solution at each of those time points. This way of invoking the model is good for sensitivity analysis and parameter optimisation purposes, where the solution at specific time points is required.
- *Dynamic mode*: The last method is to invoke the solver by supplying a begin and end time. This is the slowest method, since it dynamically allocates memory as it solves the system. The advantage of this mode is however, that you get each and every point the solver computed. This can be advantageous if your model exhibits a sudden transient somewhere in the behaviour that you might miss if you simply compute a fixed set of time points. Hence, this is actually the preferred mode for visual inspection. The blockSize setting in the ODE solver options refers to the number of time points it allocates memory for each time it crosses the memory limit. Best value for this is slightly over the total memory required such that no reallocation is required.

Note: If you call the ODE file with two outputs, the result will be a time vector and a matrix containing the solution. A single output will result in omission of the time vector in the output.

10 A few hints regarding model parameterisation

Regarding optimisation, it is imperative that you use strict tolerances. This importance stems from the fact that the optimiser uses very slight differences in parameters to estimate the gradient at a certain point. If the tolerances are sloppy, these estimates will be poor and the numerical derivatives unreliable. This manifests itself in the way that the optimiser stops early, before reaching an adequate minimum. For the experiments so far, the optimal tolerances tended to be about 3 to 4 orders of magnitude *lower* than the ones set in ode15s. This can be verified empirically by performing a few optimisations with ode15s and then with the C solver from different initial values. It is not likely that these two methods will converge to the same minimum but when the numerical Jacobian is bad, the optimiser will stop early, with the differences being obvious and very large. If the problem is numerically challenging, computing the sensitivity equations and using these to form the appropriate Jacobian is more desirable than finite differencing (also see the MATLAB help). Additionally, combined with the use of observer functions, this can result in speed-ups in non-linear least squares fitting.

However, if you run into problems, my email address is at the top of this document.

Happy modeling and good luck!

- Joep.