Speeding up simulations of ODE models in Matlab using CVode and MEX files

Natal van Riel
Eindhoven University of Technology
GEM-Z 3.109
+31 40 247 5506
n.a.w.v.riel@tue.nl

original document February 17, 2012 updated September 28, 2012

Significant computation speed-ups can be attained by compiling Matlab code. The CVode package developed in the Systems Biology / Computational Biology group at Eindhoven University of Technology 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 created to speed up large scale analyses of models based on ODE models, especially for applications in computational systems biology. The numerical integrators from the SUNDIALS CVode package (Lawrence Livermore National Laboratory, Livermore, CA) are used. Applications include parameter sensitivity analysis, parameter estimation, identifiability analysis [1] and model-based experiment design [2]. Model evaluation time can be reduced by up to two orders of magnitude [1].
Before the CVode package is introduced, first, some information is provided about compiling and working with precompiled programs in Matlab.

1. Precompiled programs in Matlab using MEX files

A MEX-file (Matlab Executable) provides an interface between MATLAB and subroutines written in C, C++ or Fortran. When compiled, MEX files are dynamically loaded and allow non-MATLAB code to be invoked from within MATLAB as if it was a built-in function.
Mex compiles and links one or more C/C++ or Fortran source files into a shared library called a binary MEX-file, executable from within MATLAB. (Note, the act of compiling code is sometimes referred to as 'building'.)

Installation The compiler Lcc-win32 C is default included with the installation of Matlab for Windows platforms (only for 32 bit!). To install and use other compilers, see below.

Configuration First run the following command to setup the compiler

```
mex –setup
```

and push enter to locate installed compilers.
If no external compilers have been installed separately, the only option will be:

```
[1] Lcc-win32 C
```

To get information about the configured mex compiler:

```
myCompiler = mex.getCompilerConfigurations()
```

To check the MEX filename extension for your platform:

```
mexext
```

©NvR

Usage
mex: Compile an executable from C source code.

mex [options ...] file [files ...]

`mex` *`filenames`*T compiles and links one or more C/C++ or Fortran source files specified in `filenames` into a MEX-file.
The first file listed in `filenames` will be the name of the resulting binary MEX-file.

<u>Example</u>
An example is used to discuss the use of MEX files.
The example YPRIME.M is discussed in the Matlab help (mex -help). To locate this m-file and copy it to the current directory:

```
%FULLFILE Build full filename from parts.
cd(fullfile(matlabroot,'extern','examples','mex'))

%COPYFILE(SOURCE) copy SOURCE to the current directory
copyfile(fullfile(matlabroot,'extern','examples','mex','yprime.m'))
```

See `>>help yprime` for some details about the m-file and the model.
First directly in Matlab:

```
tic
yprime(0,[0 1 2 3])
toc
```

Yields
```
Elapsed time is 0.878464 seconds.
```

Build the YPRIME.C example MEX-file

```
copyfile(fullfile(matlabroot,'extern','examples','mex','yprime.c'))

mex -v yprime.c
```

-v runs compiler in Verbose mode, which will print the values for important internal variables.

Execute the MEX-file:

```
tic, yprime(0,[0 1 2 3]), toc
```

Yields
```
Elapsed time is 0.000522 seconds.
```

mex can also build executable files for stand-alone MATLAB engine and MAT-file applications. See the MEX files guide from MathWorks for more details: http://www.mathworks.nl/support/tech-notes/1600/1605.html .

2. The CVode package

The CVode package can generate compiled MEX files for the simulation of ODE models, composed of systems of coupled 1st order ODEs. Such models consist of equations which contain parameters $\underline{p}$, inputs $\underline{u}(t)$ and state variables $\underline{x}(t)$:

$$\dot{\underline{x}} = f(\underline{x}, \underline{p}, \underline{u}) \qquad \text{with initial conditions} \quad \underline{x}(0) = \underline{x}_0 \qquad (1)$$

Biochemical networks are often modeled in this way [1].

This instruction is for use with the LCCWin32 compiler, which is shipped with MATLAB. It was tested for Matlab 2011b (32 bit, x86) on Windows 7 (64 bit, x64)[1]. For other compilers and/or other platforms, see [3].
The software can be used under GNU General Public License. See http://www.gnu.org/copyleft/gpl for terms and conditions.

2.1 Download

The CVode wrapper package is available from http://bmi.bmt.tue.nl/sysbio/software/pua.html. Download ODE_MEX_v6_GNU.zip.

2.2 Installation

Installation Extract the parser zip-file in a folder. The easiest place to do this is in your modeling directory ('MATLAB'). Remember to check extract with full path info. The directory parser shall henceforth be referred to as $PARSER$.

Configuration Open MATLAB and go to the modeling directory.
Type "cd $PARSER$\CVode" to go to the directory where the toolbox is extracted. For the LCCWin32 compiler, the default configuration can be used. Type

```
setupCVode
```

Go back to the modeling directory and add the toolbox directories to the search path by typing:

```
addPaths
```

Verify by typing "path". Type "savepath" if one wants to keep this search path available for future times when running MATLAB. (savepath saves the current MATLAB path in the pathdef.m file.) Otherwise "addPaths" should be run each time before the package can be used.

See [3] if you want to use a different compiler with the CVode package.

2.3 An example

First, an example to explain how the parser works. The example model consists of a single input, two states and an output:

[1] On 32-bit Microsoft Windows platforms, MATLAB provides a C compiler, Lcc. To view Help on using the Lcc compiler, type: `winopen(fullfile(matlabroot, '\sys\lcc\bin\wedit.hlp'))`

$$\dot{x}_1 = u_1 - k_1 x_1 + k_2 x_2$$
$$\dot{x}_2 = k_1 x_1 - \left(k_2 + \frac{k_3}{c_1} \right) x_2 \qquad (2)$$

here k_1, k_2 and k_3 denote rate constants. Furthermore c_1 is a known constant with value 4.
Implement this model as an m-file and save it in the modeling directory: mymodel1.m[2]

```
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(:);
```

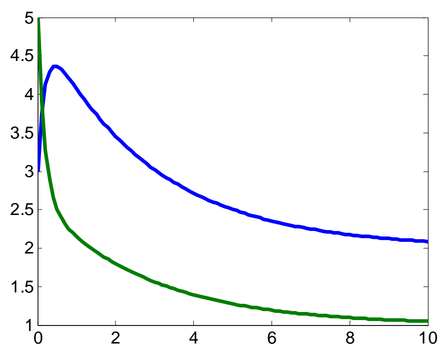
Note how the constant is hardcoded in the equations.
It is important to remember to specify every element of the output vector (in this case dx). **Not specifying an output means that the variable remains uninitialized in the compiled code, leading to undesired model behavior!**

First simulate the model 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:

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

Results in

```
Elapsed time is 0.577352 seconds.
```



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

[2] Or : `copyfile('MATLAB_models\mymodel1.m')`

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.
Create the structure required by the parser for the example 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;
mStruct.c.c1 = 1;
```

```
>> mStruct

mStruct =

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

Once this structure has been set, the parser can be used to parse the model file. Parse the m-file to a C-file:

```
convertToC( mStruct, 'mymodel1.m' );
```

The C-file dxdt.c is created in $PARSERS$\outputC\model.
Compile the C source file into a MEX file:

```
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 command window before trying to delete the file. Only compile MEX files after you have made sure the file does not exist.

The resulting MEX file 'odeC.mexw32' is available in the modeling directory and can be simulated:

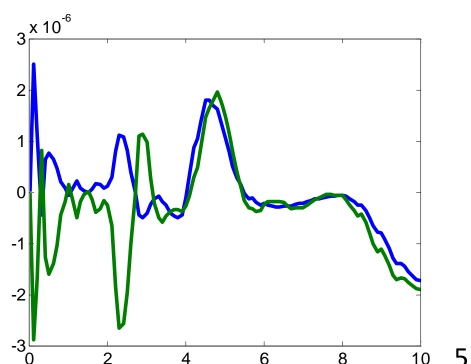
```
tic
[t y_odeC] = odeC([0:.1:10], [3,5], [2,3,4], [1], [1e-6, 1e-8, 10]);
toc
```

Results in

```
Elapsed time is 0.002200 seconds.
```

which is more than 250 times faster than the simulation with the built-in ODE solver.
To compare the results between the two simulations the relative difference is plotted:

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



©NvR

showing that the difference between the two methods is rather small (order of 10^{-6} for this specific example), but the complied version is much faster.

The five input arguments for the MEX file will be discussed in the next section.
For larger models this method of implementing the ODE file and referencing states and variables can become quite error prone. An alternative way to specify the same model is shown in [3].

2.4 Some rules regarding the MATLAB code for the parser

To work with the parser, one needs to adhere to specific rules regarding the MATLAB code to set up the m-file for the ODE model (the right-hand-side file).

- Vectors, matrices, cell arrays and complex numbers are (currently) not supported.
- 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 (min, max) 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.

Each right hand side file should have *at least* <u>four inputs</u> namely <u>time</u>, the <u>state</u>, the <u>parameter</u> and the <u>input</u> vectors. 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.

```
[t y] = ODEMEX(tspan, x0, p, u, options);
```

ODEMEX is the compiled MEX model
tspan: simulation time vector
x0: initial conditions
p: parameters
u: input
options: [RelTol, AbsTol, maxtime]
with RelTol: relative error tolerance
AbsTol: absolute error tolerance
maxtime: the maximum time the solver is allowed to take in seconds.
At each step, the solver estimates the local error `e` in the `i`th component of the solution. This error must be less than or equal to the acceptable error, which is a function of the specified relative tolerance, `RelTol`, and the specified absolute tolerance, `AbsTol`.
RelTol tolerance is a measure of the error relative to the size of each solution component. Roughly, it controls the number of correct digits in all solution components, except those smaller than thresholds `AbsTol(i)`. `AbsTol(i)` is a threshold below which the value of the `i`th solution component is unimportant. The absolute error tolerances determine the accuracy when the solution approaches zero.
The `maxtime` option was added to provide a limit to the simulation time, because specific simulations could take unacceptably long when performing large parameter sweeps. In the example maxtime was set to 10 seconds.

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 step solvers are shown in table 2.

Table 1: Parser options

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 time steps 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).
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.

Table 2: Available solvers[3]

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

The CVode package also supports *interpolation* and *if statements* in the ODE file (version 5 and up). Interpolation is required in case certain model inputs are described by data points, which is quite common in systems biology models [1]. The syntax is described in [3], as well as the syntax to perform *sensitivity analysis*.

[3] 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.

3. Building standalone applications

To compile a MATLAB application into a standalone application or software component the MATLAB Compiler Toolbox should be used

Installation 1) The Compiler Toolbox should have been installed with Matlab. This can be checked by typing `>> help compiler` in the Command Window, or look in the 'toolbox' folder for 'compiler', e.g. C:\Program Files\MATLAB\R2011b\toolbox\compiler .
If the toolbox is missing, it can be added be restarting the Matlab setup. See Appendix A for further details.
2) Install an external compiler. See http://www.mathworks.com/support/compilers for a list of compilers that is supported by your version of Matlab.
Here we will use Microsoft Visual Studio 10.0 Express. Check if Microsoft Visual Studio 10.0 has already been installed on you r computer; typically, the compiler will be installed in C:\Program Files\Microsoft Visual Studio 10.0. Details about installing Microsoft Visual Studio 10.0 Express can be found in Appendix B.

Configuration Before one can use a compiler (with either the mcc or mbuild command) it first needs to be setup. First run the following command to setup the compiler

```
mbuild -setup
```

and push enter to locate installed compilers and choose:

```
[2] Microsoft Visual C++ 2010 Express
```

This configuration step should be repeated after installing a new version of MATLAB Compiler.

Usage
Use the MATLAB Compiler by running the Deployment Tool GUI (deploytool) or executing the mcc command from MATLAB.

deploytool: Open Deployment Tool, GUI for MATLAB Compiler

mcc: Invoke MATLAB to C/C++ Compiler

Example
Using the mcc command to invoke to Matlab compiler

```
mcc -m yprime.m
```

The executable generated using mcc can be run from MATLAB command window using the ! operator e.g !yprime.exe.

Note: mbuild can also be used to compile an executable from C source code. However, in recent releases, mbuild's functionality has been replaced by mcc and deploytool. mbuild is likely to be obsoleted in coming releases.

See Matlab documentation for more information: http://www.mathworks.nl/help/toolbox/compiler/

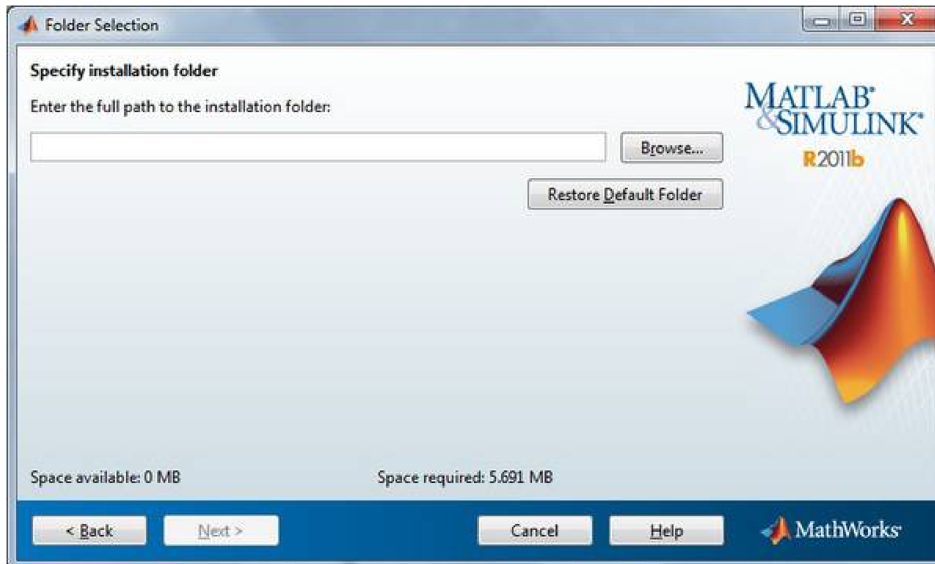
Acknowledgement
The CVode wrapper was developed by Joep Vanlier.

References

1. Vanlier J, Tiemann CA, Hilbers PA, van Riel NA. (2012) An integrated strategy for prediction uncertainty analysis, Bioinformatics, in press
2. Vanlier J, Tiemann CA, Hilbers PA, van Riel NA. A Bayesian approach to targeted experiment design, 2012, submitted
3. Vanlier J. (2010) Installation and usage instructions for the CVode Wrapper, December 17, 2010

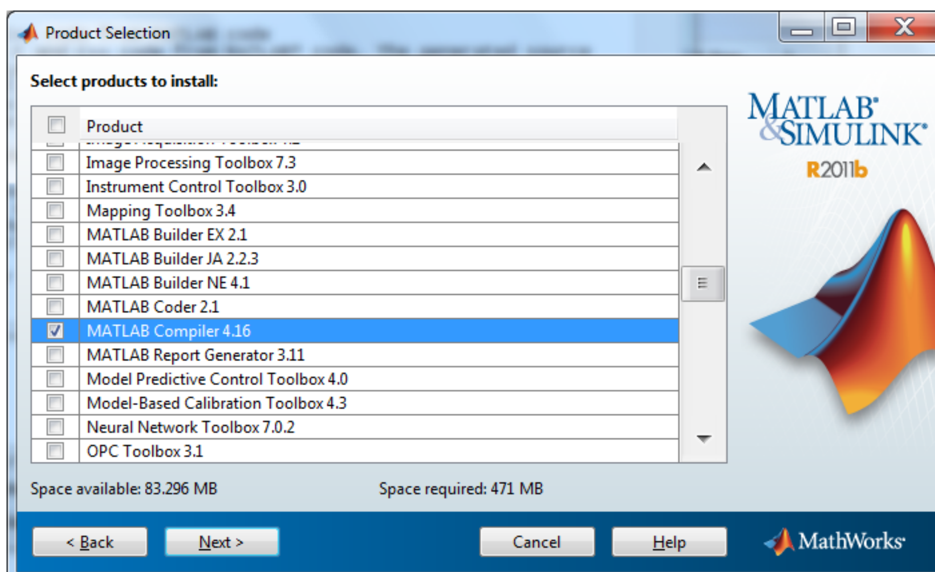
APPRENDIX A Adding Matlab toolboxes at a later stage

Go to http://w3.tue.nl/en/services/dienst_ict/services/services_wins/campussoftware/matlab/ (for TU/e students and employees only), select your version of Matlab and choose Custom installation.



Enter the correct installation path (typically C:\Program Files\MATLAB\R2011b) because otherwise a completely new version of Matlab will be installed. (It is recommended to use the Browse-button in order to avoid typing errors.)

Select MATLAB Compiler from the product list:



Start Matlab and type `>> help compiler` in the Command Window to verify that the Compiler Toolbox has been added.

APPENDIX B Installing the Microsoft Visual C++ compiler

The Microsoft Visual C++ 2010 Express compiler is available thru a web setup (http://www.microsoft.com/visualstudio/en-us/products/2010-editions/visual-cpp-express) or a via an ISO image file (http://www.microsoft.com/visualstudio/en-us/products/2010-editions/express-iso).