PPODE SUITE
Manual

Pascal A. Pieters

March 2014

Contents

Chapter 1

Introduction

The PPODE SUITE aims to bring ordinary differential equation (ODE) solvers to MATLAB that - in a lot of cases - will increase the performance with respect to the default ODE solvers provided MATLAB . The default MATLAB ODE solvers use the MATLAB language to define the ODE system, which allows access to the wide variety of MATLAB functions and toolboxes. However, since MATLAB is a scripting language, the execution of MATLAB code requires an extra interpretation step at every execution compared to languages like C/C++ and Fortran , which require a translation step only once (Figure 1.1).

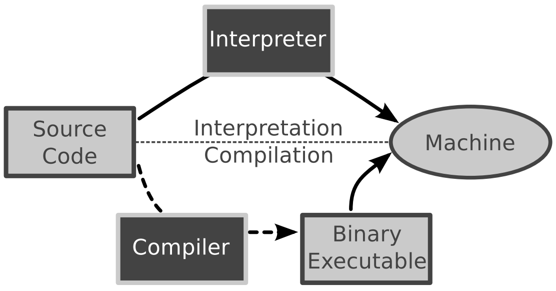

Figure 1.1: Schematic overview of the difference between translation/compilation and interpretation. Dotted arrows represent translation steps, whereas solid arrows represent execution steps. Interpretation generates has a delay every execution, compilation only requires a (slow) execution step once. This compilation step generates a binary machine-readable file, which can be executed almost directly on the hardware layer of a computer.

Considering the differences between compilation and interpretation, various cases can be identified where compilation would be preferable over interpretation. In general, compilation is superior when the number of executions desired eliminates the overhead of the compilation procedure (e.g. data fitting). This holds as long as no MATLAB specific functions or toolboxes are being used, which is mostly not the case for ODE systems.

Furthermore, there is a more specific problem with the build-in MATLAB solvers, namely that not for every specific situation the most suitable solver is available. For example, large systems of sparsely interlinked differential equations (e.g.

Equation 1.1) become time consuming to solve using MATLAB .

$$\frac{d\mathbf{F}}{dt} = \begin{pmatrix} -\alpha_1 & \alpha_2 & 0 & \cdots & \cdots & 0 \\ \alpha_1 & -\alpha_2 & \alpha_3 & \ddots & \ddots & \vdots \\ 0 & \alpha_2 & -\alpha_3 & \alpha_4 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & \cdots & \cdots & \alpha_{n-1} & -\alpha_n \end{pmatrix} \mathbf{F} \tag{1.1}$$

However, suitable solutions for these problems exist. The fast increase of time it takes to solve the system is caused by an exponentially increasing Jacobian matrix. The Jacobian matrix is used by stiff solvers and defined as shown in Equation 1.2.

$$J_{m,n} = \begin{pmatrix} \frac{\partial F_1}{\partial x_1} & \cdots & \frac{\partial F_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial F_m}{\partial x_1} & \cdots & \frac{\partial F_m}{\partial x_n} \end{pmatrix} \tag{1.2}$$

Where F represents the system of ODEs. In the case of an ODE system, m and n are both equal to the number of equations (N). This means that the number of elements of the Jacobian matrix increases exponentially, resulting in the following behaviour:

$$i_{Jac} = N^2 \Rightarrow t_{Jac,op} = \Omega(N^2) \tag{1.3}$$

Where i_{Jac} is the number of elements in the Jacobian matrix and $t_{Jac,op}$ the time a typical operation on the Jacobian matrix takes.

In the specific case of Equation 1.1, the Jacobian matrix is a band matrix, which enables specialized matrix handling methods for the Jacobian matrix, which only make $t_{Jac,op}$ increase linearly.

A more general method is the use of a sparse implementation of the Jacobian matrix. This allows elements of the Jacobian matrix not in the center band to be non-zero, in contrast to band matrix implementations. This suite was created to enable MATLAB to call this type of ODE solvers, following the need to solve large nucleation-elongation ODE systems. These system make use of a cut-off to limit the number of equations, since the exact system consists of an infinite amount of equations. To facilitate the use of a cut-off, the number of equations was left variable and can vary between function calls, without the need to recompile the code.

Next to these specific "sparse Jacobian" solvers, a wide variety of other solvers - all written in the Fortran programming language - is available through this suite, for an overview see Chapter 4. The suite consist of the following tools:

PPODE_init builds the ODE solver libraries and in principle only has to be executed once. See Chapter 2.

PPODE_addPaths loads paths of the PPODE SUITE into the MATLAB path variable. To access the PPODE SUITE , this function has to be executed. See Chapter 2.

PPODE_parser parses a MATLAB ODE function to Fortran . The use of this function eliminates the need for knowledge about the Fortran programming language. See Chapter 3.

PPODE_build builds the Fortran function generated by PPODE_parser. Of course, one can also manually provide a Fortran function. See Chapter 3.

Chapter 2

Installation

The PPODE SUITE package depends on:

MATLAB Tested on MATLAB version R2013b (8.2) 64-bit.

gfortran and GCC Tested on version 4.8.1 64-bit.

2.1 Linux and other Unix variants

1. Meet the software requirements by installing MATLAB and the GCC package. MATLAB download and installation instructions can be found on the MathWorks website. The gfortran/GCC package can be obtained from the ⋆nix distribution repository through the distribution package manager. Using Ubuntu for example:
   ```
   $ sudo apt-get install gfortran gcc
   ```

2.2 Windows

Currently, running the PPODE SUITE on Windows is not supported nor tested.

1. Running the PPODE SUITE on Windows would require installing a combination of GCC/Cygwin and gnumex (*http://gnumex.sourceforge.net/*) or the Intel Visual Fortran Composer. To check which versions are supported by MATLAB , check *http://www.mathworks.nl/support/compilers/R2013b/* substituting R2013b with the right version number.

2.3 General

2. Download and extract the PPODE SUITE package.
3. Open matlab and navigate to the extracted PPODE SUITE folder. Add the *PPODE* paths to the matlab path variable.
   ```
   >> PPODE_addPaths
   ```

4. Now the libraries of the different solvers can be build. In order to do so, execute the following command;
   ```
   >> PPODE_init
   ```
   The options 'Debug' can be used to build the libraries with debugging symbols.
   ```
   >> PPODE_init('Debug', 1)
   ```
   For more information, type "`help PPODE_init`" in the MATLAB command window.

Chapter 3

Usage

The ODE solvers need an ODE system to be provided as a Fortran subroutine (function), this can be done either manually or using the parser provided by the PPODE SUITE . Both options are described in the next two sections.

3.1 Manual ODE Function

3.1.1 Introduction

The ODE function of the problem should be written in Fortran 95. Here are some main Fortran peculiarities to consider when writing Fortran code.

Line Formatting The maximum line width is 72 characters. The first character is used to indicate whether the line is a comment line. The second to fifth character are used to indicate labels. The 6th character is used to indicate the continuation of the previous line.

Listing 3.1: Syntax Example

```
c        1         2         3         4         5         6         7
c23456789012345678901234567890123456789012345678901234567890123456789012

! Comments should be introduced by either a 'c' or a '!'.
      if (answer .gt. 42) go to 4242
 4242 ydot(s) = y(1) * (kp * y(s - 1) - gp * y(s)) + gm * y(s + 1) +
     + km * y(s)
```

For Loops For loops are written using the `do` statement. They should be written in the form `do ⟨label⟩ ⟨var⟩=⟨start⟩, ⟨stop⟩[, ⟨step⟩]`. The label should refer to a `continue` statement at the end of the loop.

Listing 3.2: Do-Loop

```
      a = 0
      do 42 i=1, 20
        a = a + 1
   42 continue
! a has the value 20 here.
```

Case Sensitivity The Fortran language is not case sensitive.

Vectors Vectors indexing starts at 0, just like in MATLAB .

3.1.2 Template

The Fortran subroutine that defines the ODE system should have the following arguments:

neq *input* Number of equations.

t *input* The current time point.

y *input* The current value of all states. The length of this vector is equal to `neq`.

np *input* Number of parameters.

p *input* Vector of the values of all parameters.

ydot *output* This is a vector of length `neq` to which all derivatives of the states should be written.

Listing 3.3: ODE Template

```
c-----------------------------------------------------------------------
c
c PPODE ODE function - Model Name
c   Short model description.
c
c DEVELOPED BY:
c
c   Pascal Pieters <p.a.pieters@student.tue.nl>
c
c-----------------------------------------------------------------------
c
c ARGUMENTS:
c
c    neq :in    Number of states/equations.
c      t :in    Current time point.
c      y :in    Vector of the current values of the states.
c     np :in    Number of parameters.
c      p :in    Vector of the values of the parameters.
c   ydot :out   Vector of the numerical derivatives of the states.
c
c PARAMETERS:
c
c   p(1) :in    s   : Parameter description.
c   p(2) :in    kp  : ...
c
c-----------------------------------------------------------------------

      subroutine func (neq, t, y, np, p, ydot)
      integer neq, i, s, np
      double precision t, y, ydot, kp, par
      dimension y(neq), ydot(neq), par(np)

      s = int(p(1))
```

```
kp = p(2)
...
ydot(i) = ...
...

return
end
```

Examples can be found in the "⟨PPODE SUITE Source⟩/examples" folder.

3.2 Parser

PPODE SUITE can translate a MATLAB function to Fortran . The function that executes this procedure is PPODE_translate. The parser is created using a combination of lex and yacc/bison. The parser interprets the the MATLAB code and creates a tree structure out of it. This tree structure is then used to create the Fortran code. This provides more flexibility and better interpretation than more direct forms of translation. An additional benefit is that the structure can be used to determine the Jacobian of the function.

3.2.1 Restrictions and Pitfalls

The MATLAB ODE function should have the following structure:

```
>> ⟨dx⟩ = func( ⟨t⟩, ⟨x⟩, ⟨par⟩, ⟨neq⟩, ⟨np⟩ )
```

Where ⟨t⟩ is the independent variable, ⟨x⟩ the dependent variable(s) and ⟨par⟩ the parameter values. The last two arguments are optional and represent the number of equations (⟨neq⟩) and number of parameters (⟨np⟩).

If the number of equations is not fixed, it should always depend on ⟨neq⟩. If you would for example use γ as a parameter, which would per definition result in $2 \cdot \gamma$ equations, define $\gamma = \frac{neq}{2}$ in your code, instead of passing γ as a parameter.

One of MATLAB 's main benefits is the ease of using vectors and matrices. Not all of this functionality can easily be ported to Fortran , therefore there are some restrictions to consider when using vectors. First of all, matrices are (currently) not supported, only 1-dimensional datatypes are supported, i.e. vectors. Moreover, MATLAB supports dynamic vectors (vectors that can change size during exection). This could be implemented in Fortran , but it was chosen not to since almost all code that is written using dynamic vectors can also be written using static vectors, which in most cases is much faster. So make sure the size of vectors does not change in the ODE function and **all vectors are initialized using the "zeros" function.**

For example, the following code will not be parsed correctly and is slow and a bad coding practice in general:

```
>> dx = []; dx = [dx da]; dx = [dx da];
```

Here, the vector dx is initialized as an empty vector and changes size twice afterwards. Better code, that can be translated to Fortran is the following:

```
>> i = length(da); dx = zeros(2*i, 1); dx(1:i) = da; dx((i+1):(2*i)) = da;
```

Furtermore, bear in mind that MATLAB has a lot of specific functions, that will are not implemented in Fortran or the parser and will therefore not work.

3.3 Building the MEX Function

The MATLAB function `PPODE_build`, included in the PPODE SUITE , can be used to build the ODE Fortran file against the right solver libraries. First of all, make sure the PPODE SUITE paths are added to the MATLAB path variable.

```
>> PPODE_addPaths
```

And the libraries are build.

```
>> PPODE_init
```

Now the function `PPODE_build` can be used. Extensive help can of course be acquired using "`help PPODE_build`". The simplest usage of the function is the following:

```
>> PPODE_build('odeproblem.F', 'odeproblem_stiffsolver')
```

This command will generate a MEX file named 'odeproblem_stiffsolver' of the problem defined by 'odeproblem.F', using the default (stiff) solver. The correct file extension is automatically added to the MEX file, so do not supply an extension for the second function argument.

The first two mandatory arguments, extra options can be specified. This is done by first giving the option name and then the value. For example, if the problem is not stiff and one would like verbose output, a non-stiff solver should be specified and the verbose mode should be enabled:

```
>> PPODE_build('odeproblem.F', 'odeproblem_stiffsolver', ...
    'Solver', 'Non-Stiff', 'Verbose', 1)
```

Note that both the option name and value are case insensitive.

3.3.1 Sparse Jacobian Matrix

When using the solver that uses a sparse matrix implementation for the Jacobian matrix, the number of non-zero values of the Jacobian matrix should be supplied. There are two options to tackle this problem.

The first and default option needs an analytical Jacobian to be specified. The generated function will evaluate the Jacobian two times to determine the number of non-zero elements. This option can be selected by setting 'INPUTNNZ' to zero.

The second option is to manually provide a number of non-zero elements. This option can be selected by settting 'INPUTNNZ' to one. When this option is set, the generated function will require the first argument to be a 2x1 vector consisting of first the number of equations, and second the number of non-zero elements.

In order to determine the number of non-zero elements of the Jacobian matrix, consider the definition of this matrix:

$$J_{m,n} = \begin{pmatrix} \frac{\partial F_1}{\partial x_1} & \cdots & \frac{\partial F_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial F_m}{\partial x_1} & \cdots & \frac{\partial F_m}{\partial x_n} \end{pmatrix} \tag{3.1}$$

In the case of an ODE system, m and n are both equal to the number of equations. F represents the system of ODEs. The number of nonzero elements can also be determined using trial and error. By just trying some values for `nnz`, the error messages might give away the correlation between `nnz` and the parameter and number of equations (this does not work with LSODES).

3.4 Executing the MEX Function

The MEX function generated by the PPODE build function can be called as follows:

```
>> [⟨t⟩, ⟨y⟩] = ⟨F⟩(⟨neq⟩, ⟨abstol⟩, ⟨reltol⟩, ⟨times⟩, ⟨par⟩, ⟨y0⟩)
```

Where ⟨F⟩ is the name of the MEX function, ⟨neq⟩ is the number of equations, ⟨abstol⟩ and phreltol are the absolute and relative tolerances respectively, ⟨times⟩ is a vector of time points at which output is desired, ⟨par⟩ is a vector of parameter values and ⟨y0⟩ is a vector of the initial values of the states. The function returns the vector ⟨t⟩ with the time points at which the values of the states are calculated. The matrix ⟨y⟩ contains the values of all states at the time points specified by ⟨t⟩.

Chapter 4

Solvers

4.1 Introduction

The solver for the ODE problem can be specified using the 'Solver' option of the `PPODE_build` function:

```
>> PPODE_build(⟨source⟩, ⟨target⟩, 'Solver', ⟨solver⟩)
```

Valid options for ⟨solver⟩ are:

'Stiff' (or 'BDF') The BDF based solver of the LSODE package. *See 4.2.1.*

'Stiff2' (or 'VODE') The BDF based solver of the VODE package. *See* **??**.

'MEBDFSO' (or 'MEBDFSparse') The modified extended BDF based solver using a sparse Jacobian matrix. *See 4.2.2.*

'LSODES' (or 'BDFSparse') The BDF based solver using a sparse Jacobian matrix. *See* **??**.

'Non-Stiff' (or 'Adams-Moulton') The Adams-Moulton based solver of the LSODE package. *See 4.3.1.*

'Non-Stiff2' (or 'VODEAM') The Adams-Moulton based solver of the VODE package. *See* **??**.

'RK23', 'RK45', 'RK78' The Runge-Kutta based solvers of the RKSUITE package. *See 4.3.2.*

'Switching' (or 'LSODA') The solver that switches between the non-stiff Adams-Moulton based solver and the stiff BDF based solver of the LSODE package. *See 4.4.1.*

If you do not know which solver to choose, but you already know which MATLAB solver performs best, table 4.1 might be helpfull.

4.2 Stiff

4.2.1 BDF

The Backward Differential Formulas based method uses the BDF implementation that ODEPACK supplies. The order of these formulae can range between

MATLAB	Equivalent	Probably Also Suitable
ode45	'RK45'	'RK78', 'Non-Stiff', 'Non-Stiff2'
ode23	'RK23'	'RK45', 'Non-Stiff', 'Non-Stiff2'
ode113	'Non-Stiff', 'Non-Stiff2'	'RK45'
ode15s	'Stiff', 'Stiff2'	'Switching' (partially stiff problems), 'MEBDFSO' (large number of states that are not very interdependent)
ode23s	-	'Stiff', 'Stiff2', 'Switching', 'MEBDFSO'
ode23t	-	'Switching', 'Stiff', 'Stiff2', 'MEBDFSO'
ode23tb	-	'Switching', 'Stiff', 'Stiff2', 'RK45', 'RK23', 'MEBDFSO'

Table 4.1: Solver selection helper.

1 and 5 and can be limited by setting the 'MaxOrder' option when building the ODE system.

Credits

Credits for the ODEPACK package obtained from (*http://www.netlib.org/*).

Author	Alan C. Hindmarsh
Institution	Center for Applied Scientific Computing, L-561 Lawrence Livermore National Laboratory Livermore, CA 94551 United States of America

4.2.2 Modified Extended BDF using Sparse Jacobian

The Modified Extended Backward Differential Formulae based method uses the BDF implementation that MEBDFSO supplies. The order of these formulas can range between 1 and 5 and can be limited by setting the 'MaxOrder' option when building the ODE system.

Credits

Credits for the MEBDFSO package obtained from (*http://www.netlib.org/*).

Authors	T.J. Abdulla J.R. Cash
Institution	Department of Mathematics Imperial College London SW7 2AZ England
Contact	t.abdulla@ic.ac.uk j.cash@ic.ac.uk

4.3 Non-Stiff

4.3.1 Adams-Moulton Methods

The Adams-Moulton method based solver uses the Adams-Moulton implementation that ODEPACK supplies. The order of these formulae can range between 1 and 12 and can be limited by setting the 'MaxOrder' option when building the ODE system.

Credits

Credits for the ODEPACK package obtained from (*http://www.netlib.org/*).

Author	Alan C. Hindmarsh
Institution	Center for Applied Scientific Computing, L-561
	Lawrence Livermore National Laboratory
	Livermore, CA 94551
	United States of America

4.3.2 Runge-Kutta Methods

The Runga-Kutta methods based solver uses the RKSUITE package. Three Runge-Kutta pairs are available: 2-3, 4-5 and 7-8. Use higher orders in combination with smaller tolerances.

Credits

Credits for the RKSUITE package obtained from (*http://www.netlib.org/*).

Author	R.W. Brankin
Institution	Numerical Algorithms Group Ltd.
	Wilkinson House
	Jordan Hill Road
	Oxford OX2 8DR
	United Kingdom
Contact	richard@nag.co.uk
	na.brankin@na-net.ornl.gov
Authors	I. Gladwell
	L.F. Shampine
Institution	Department of Mathematics
	Southern Methodist University
	Dallas, Texas 75275
	United States of America
Contact	h5nr1001@vm.cis.smu.edu

4.4 Mixed

4.4.1 Switching between BDF and Adams-Moulton Methods

The switching method uses the LSODA subroutine that the ODEPACK package supplies. This method automatically switches between the BDF based stiff solver (order 1-5) and the Adams-Moulton methods based non-stiff solver (order 1-12).

Credits

Credits for the ODEPACK package obtained from (*http://www.netlib.org/*) and the LSODA subroutine in particular.

Author	Alan C. Hindmarsh
Institution	Center for Applied Scientific Computing, L-561 Lawrence Livermore National Laboratory Livermore, CA 94551 United States of America
Author	Linda R. Petzold
Institution	Univ. of California at Santa Barbara Dept. of Computer Science Santa Barbara, CA 93106 United States of America