Generating code with `exmex`; a crash course

Anders Lennartsson
Department of Mechanics
Royal Institute of Technology
S-100 44 Stockholm
Sweden

November 4, 1998

Abstract

A short description of the prerequisites involved in succesful code generation of systems of ordinary differential equations and some other expressions is made. It is followed by a brief manual for `exmex`, the code generating procedure for exporting equations from Maple to MATLAB.

1 Introduction

Systems of ordinary differential equations, ODEs, is the result of physical modeling in many fields, chemistry, physiology, and biology are some examples other than mechanics. Preparing the equations of motion for efficient numerical evaluation is a vital part of the analysis process, especially if extensive numerical analysis is about to be performed. `exmex` was designed to generate C code of ordinary differential equations and provide useful tools to aid in the analysis of the dynamical systems governed by these equations. The code generated by `exmex` is designed to be linked as external functions to MATLAB and called as any other functions from within a MATLAB workspace. The name `exmex` was derived as an acronym to *EXport Matlab EXternal.*

The choice of MATLAB was purely one of convenience, both commercial and free software exist that have more or less the functionality of MATLAB, examples are $\mathrm{MATRIX_X}$ and SCILAB. Many of these can also link compiled code for direct access from the main program. For instance, succesful experiments have been made with linking slightly modified code generated by `exmex` to SCILAB, and tests have been made with exporting code from the computer algebra software MuPAD.

2 Methods and concepts

The process of transforming ordinary differential equations to computer code consists of the steps shown in figure 1.

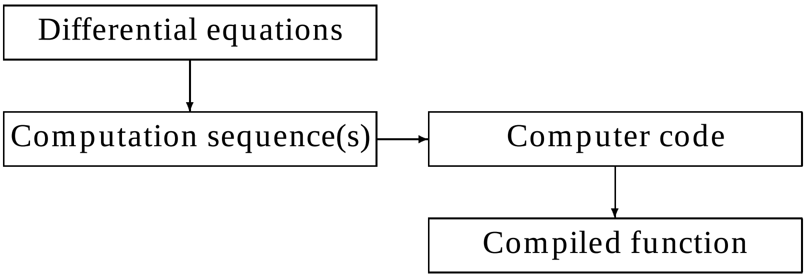

Figure 1: Systems of ordinary differential equations are translated into compiled functions by the steps shown in the figure. The computation sequences are determined based on what the equations will be used for, and what options the function will respond to.

We will not dwell on concepts and methods to understand all details of the process, but a few are important to understand before we turn to the actual use of `exmex`.

2.1 Mass matrices

Reformulating higher order systems on first order form creates a block structure of the mass matrix where the number of blocks equals the order of the original ODEs. The structure is seen in the following matrix representation of the dynamic equations and the kinematic differential equations for a mechanical system

$$\begin{bmatrix} \mathcal{M}_\beta & 0 \\ 0 & W^{\beta\tau} \end{bmatrix} \begin{bmatrix} \dot{u}_\beta \\ \dot{q} \end{bmatrix} = \begin{bmatrix} R_\beta \\ u_\beta \end{bmatrix}. \tag{1}$$

A typical case where the second block is explicitly given, is the use of $\dot{q} = u$ as kinematic differential equations, which normally renders the first block matrix to have a full, or nearly full, structure.

Well designed algorithms for solving equations (1) take advantage of the block structure. The `exmex`-package supports systems converted from second order in a general way by accepting two *separate* systems of ODEs as arguments. These may be independently specified on implicit or explicit form.

2.2 Subexpressions

Even if Maple's optimization algorithm can locate and remove excessive subexpressions, it can be useful to interactively introduce intermediate variables during the derivation process. Sometimes an expression is formulated that should be introduced at a large number of positions in the equations being derived. This can happen at all stages during the derivations, in the specification of geometry, velocity, acceleration, or forces. Normally, the earlier it is specified the more it spreads. The function of the optimization algorithm is to do reverse engineering and find all the identical expressions again. If the equations are expanded, much of the structure of the specific expression may be lost which can inhibit the algorithm from finding many of the expressions. Usually there is an increase in complexity when expansion is performed but it can make the equations less complex.

An example where an intermediate variable is worthwhile to introduce manually is to represent the magnitude of relative velocity between a particle and a surrounding medium. Such expressions are used for example when modeling fluid drag on particles. Consider a particle with position $q_1\boldsymbol{n}_1 + q_2\boldsymbol{n}_2 + L\boldsymbol{n}_3$ relative to a reference point,

```
> r1:=Evector(q1,q2,L,FN):
```

With the identity transformation as kinematic differential equations, the velocity $\boldsymbol{v}_1$ relative to the reference frame `FN` is

```
> v1:=subs(kde,diffFrameTime(r1,FN));
```

$$v1 := [[u1,\ u2,\ 0],\ FN]$$

Now, if the particle is suspended in a fluid moving with a steady velocity $\boldsymbol{v}_m = -U\boldsymbol{n}_1$

```
> vm:=Evector(-U,0,0,FN):
```

the velocity of the particle relative to the fluid is the difference

```
> v1 &-- vm;
```

$$[[\mathit{u1}+U,\ \mathit{u2},\ 0],\ \mathit{FN}]$$

Introduction of an intermediate variable to represent the value of the magnitude of the velocity difference can be done by keeping a definition of the new variable in a set or list. The intermediate variable is not a Maple variable in the sense that it holds a value, it is only used as a symbol on the left hand side of the equation that defines it, and as a symbol in the positions where it replaces the subexpression. If we choose `vd` as the name of the new variable the definition can be stored as a member of the list `sx`,

```
> sx:=[vd=Emag(v1 &-- vm)]:
```

The new variable is then used wherever the difference should appear, for example in the dynamic pressure $p = \rho v_d^2/2$.

2.3 Derivatives of the dynamical system

Numerical analysis of dynamical systems benefit from a number of derivatives of the base system, the jacobian, and the first order variational equations are two of them. Functionality for these purposes have been developed and is included in `exmex`. Code for evaluation of the jacobian is optionally derived and generated for normal systems as well as variational systems where the base system is extended with the variational equations.

2.4 External functions

Differential equations with discontinous functions as forces, or perhaps even time dependent functions from real time measurements of physical elements is plausible. To process such systems it is sometimes useful with functions not explicitly defined in Maple, but available as a function to include when compiling the generated code. External functions are supported in systems processed by `exmex`, and one or two options are used to define them and their properties. One argument specifies all files to include while compiling the generated source code. These files are assumed to hold functions written in C and used by function calls in the equations.

2.4.1 Example

Suppose we have a system where a force is defined as a function of configuration,

$$f_1 = \begin{cases} 0 & q1 > 0 \\ -kq_1 & q1 < 0 \end{cases} \tag{2}$$

Such a function is difficult to define in Maple but more easily done in C code,

```
double f1(double q1, double k)
{
  if (q1>0.0)
    return 0.0;
  else
    return -k*q1;
}
```

The indeterminate form `f1(q1,k)` is used in the Maple expressions and when generating the code for a derivative evaluator one includes an option that specifies a file where `f1` is defined.

2.4.2 Derivatives of external functions

If code for the jacobian or variational equations are generated, the derivatives of the external functions with respect to the variables must also be available as external functions. Differentiations are performed with respect to the variables of the systems, not the parameters. Derivatives may be named arbitrary, in which case a specification of all names must be given as an argument to `exmex`. Otherwise derivatives should follow the pattern where new functions are introduced by concatenating `d`, the function name, `d`, and the variable name. The derivative of our example function would be named `df1dq1`. It would have to be specified in an included file,

```
double df1dq1(double q1, double k)
{
  if (q1>0.0)
    return 0.0;
  else
    return -k;
}
```

Information about how derivatives are named may be declared in two ways, both in the form of an equation with the identifier `diffset` or `‘diffset‘` to the left and a list on the rigth hand side. If the naming convention described above is followed, it is enough if the list contains function calls specifying names and arguments of the external functions. Arbitrary names of derivatives must be specified with their full dependencies in the list. If the last was true for the example of this section, the appropriate option would be

```
'diffset'=[diff(f1(q1,k),q1)=df1dq1(q1,k)];
```

As the names involved follow our naming convention, the less complex option

```
'diffset'=[f1(q1,k)];
```

will do perfectly. The difference in complexity makes the latter alternative considerably more useful for large systems.

2.5 The C code

Two individually specified systems of ODEs are arguments to the code generating functions of `exmex`. A single system is of course also a valid input, in that case one of the specifications is left empty. For the purpose of generating computation sequences it is assumed that the two systems represents parts of one system of ODEs. This means the union of variables for each system should be the variable set for the entire system. Subexpressions are specified in one separate argument, but are considered valid for the complete system.

2.5.1 Variable names

Quite a free naming convention is in use, only a few names are reserved. Among them is the variable `t` which represents time and must not be used for anything else. Names consisting of a `t` and a number, i.e. `t1`, `t2`, . . . , are given to temporary variables during symbolic optimization and other use is likely to destroy the validity of the optimized computation sequence. Maple's C-code generator can occasionally, for very complex expressions, generate intermediate variables which are named `s1`, `s2`, and so on. The best method to detect the introduction of such variables seem to be the compilation errors that occur as these variables are not declared. The fix is to manually edit the code and add the declarations, and it is therefore recommended to avoid these names. Internal variables are named with the string `_exmex` somewhere in the name and should be easy to avoid. The C code is adapted to the conventions in mex-files for MATLAB 5 (early versions of `exmex` generate code for both MATLAB 4 and 5).

2.6 Compilation of generated code

The script `mex` that compile external functions for use with MATLAB is available with the MATLAB distribution for the specific platform. Source codes produced with `exmex` have been used on SunOS 4, Solaris 2.5, Macintosh, Win95/NT, IBM RS/6000, and Linux. Compilers used on the Sun have been Sun Sparc C Compiler and the GNU compiler gcc, MPW C on the Macintosh, Watcom C/C++ in the PC environment, and XLC on the IBM RS/6000 series. In Linux gcc was used. A typical command is `mex file.c`, and most unices are like this. The command can be given either in MATLAB or in a shell. Similar syntax works on a correctly configured 95/NT-machine. For the IBM RS/6000 with AIX, a specific compiler had to be specified with the option `CC=c89`, and for large systems the flag `CFLAGS="-qspill=5000"` was added.

The most important isssue for compilation is to avoid compiler optimizatoin if the code has been symbolically optimized. Apparently, the use of compiler optimization on already optimized code does not increase performance, but slows down compilation significantly.

3 Manual

The use of `exmex` is described together with the supported options.

3.1 Initialization

The functions are available as text files to be loaded into Maple with the `read` command. If they are not used with the Sophia package for mechanics, the `linalg` library must be loaded first, for example by executing `with(linalg):` in Maple. There are a few support routines, among them is `init_exmex` used to load some necessary libraries by executing the statement `init_exmex():` at a Maple prompt. The libraries are `C`, `optimize`, and `cost` used to generate code, do symbolic optimization, and compute the numeric cost of evaluating expressions. In Maple V Release 3 the initialization includes a correction of a bug in the `optimize`-library that causes erratic behaviour when removing redundant temporary variables. For Maple V Release 5 there is a modified version of the C-code generator that allows writing to already open files. The modified function is called `SC`. When the package is initialized one must not use the symbol `C` as a variable because it is the name for the function that generates code.

3.2 User interface

Generating and using functions is made in two different softwares, Maple, and MATLAB. The syntaxes involved in processing of ODEs and matrices and the subsequent use of the generated functions is schematically specified in figure 2. As we can see, each environment has a specific syntax which the user must adapt to. In Maple, specifications must be made such that the generation process is completely determined. The generated function is then used with a specific syntax in MATLAB.

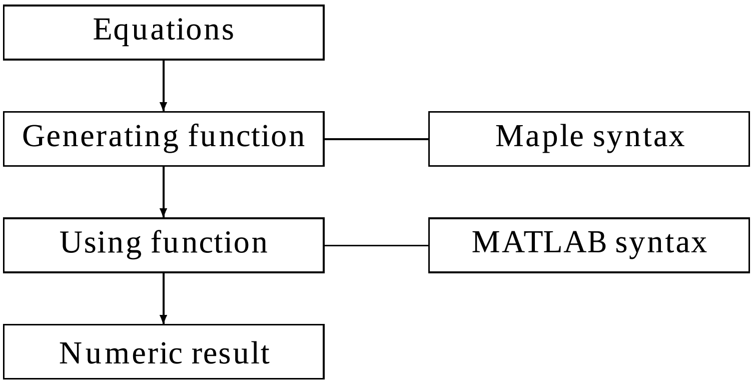

Figure 2: The flow of expressions through different environments when generating functions and using them in another environment.

3.3 exmex

The `exmex` procedure is available for Maple V Release 3, 4, and 5, but only the latest version is described here. It generates derivative evaluators, as mex-functions in C code. The calling syntax of the generated functions was designed for use with the family of solvers for ordinary differential equtaions in MATLAB, `ode45`, `ode15s`, and others. Parameters may be defined as static declarations in the source code or parsed via the integrators fourth argument. By default, code for MATLAB version 5 is generated.

The integrators in MATLAB version 5 support events, evaluation of the jacobian, and asking the function for suitable initial values etc. Events are used to trigger an action of the integrating function when a value of some expression is reached. The action can be to stop the integration or collect the state in a matrix. Trigging happens when one of any number of functions reach the value zero, and can be specified as dependent or independent of direction. Use of these possibilities is specified with an option.

`exmex` can also produce functions written in C with the syntax adapted for Simulink. This is identified by giving an option, and the choices are described later on.

3.3.1 Help files

MATLAB mex-functions can hold no comments for help information. A standard method to overcome this is to have an m-file with the same name which holds the help information, and such a file is automatically generated by `exmex`.

3.4 Maple syntax

Seven arguments are required, and options may be given in any order after those. More specifically the syntax follows,

```
exmex(name,path,eqs1,eqs2,sub_ex,der_lst,var_lst,opt,...)
```

`name` is a Maple symbol, string, or unassigned name, determining the name of the source code and the help file. The proper extensions `.c` and `.m` are added automatically. A typical name is `"file"`.

`path` is a Maple symbol, string, or unassigned name specifying the directory where the generated files are created. The exact form of the string is platform specific. On Unix systems the directory separator is the slash `/`, on Macintosh systems the colon `:` is used, and the DOS / WIN world has the backslash, \. An example path for a unix system is `"/home/user/maple/"`.

`eqs1` and `eqs2` should each be a Maple list, which *must* contain one or two lists. One list is used for equations on state space form i.e. explicit form, while two lists are used to specify equations on implicit form. The second list then contains the names of the variables to solve for. A valid example with two explicit equations is

```
[[x1t=sin(x2),x2t=x1]]
```

Implicit equations should be specified by the first list containing the implicit equations and the second list the names of the variables to solve for. The values of the variables are found by numeric solution of the system. A simple example is a list with two sublists,

```
[[x1t+cos(x2)*x2t-sin(x2),x2t-x1],[x1t,x2t]],
```

which represents the implicit equation system

$$\begin{bmatrix} 1 & \cos(x2) \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} sin(x_2) \\ x_1 \end{bmatrix}. \tag{3}$$

An empty system is specified by a list with an empty list i.e. `[[]]`.

`sub_ex` is a Maple list of equations. These equations are evaluated in the specified order before the other two groups of equations. If no subexpressions are used, an empty list should be given as argument.

`der_lst` is a Maple list of variable names used for state space derivatives. After evaluation of all given equations the values of these variables put in a vector is the result of the function. The combination of unknowns in `sys1` and `sys2` should equal the names in this list. In the examples above, the derivatives would be represented by `[x1t,x2t]`.

`var_lst` is a Maple list of variables used for state space position. These variables represent the corresponding numeric value of the vector argument to the function during the evaluation of the equations. A list with variable names for the previeous explicit and implicit cases is `[x1,x2]`.

3.4.1 Options

Options are given in any order or combination after the first seven arguments.

3.4.2 Parameters

Parameters are specified with an equation where the left hand side is the string `‘parameters‘` and the right hand side a list of parameter names, as in `‘parameters‘=[par1,par2,...]`. The generated function is written so as to expect one vector with the parameter values as components and the order of the values should correspond to the order in which they were declared in this option.

3.4.3 Static variables

Parameters may also be defined in the code as statically declared variables by an equation such as `‘statics‘=[stat1=val1,stat2=val2,...]`. The list defines names and values to be declared, which can later be edited but require recompilation to take effect.

3.4.4 Simulink blocks

With a similar technique as the previous options, code for Simulink blocks is specified by an equation of the form `‘simulink‘=[a1,a2,a3]`, where `a1`, `a2`, and `a3` are lists. The first list holds names of input variables, and the second list holds output expressions, i.e. functions of the variables or time derivatives. The third list is a specification of subexpressions used in the expressions defined in the second list. It is also possible to generate general blocks, without any differential equations the block will generate a *feedthrough* block that can compute general functions of the inputs.

3.4.5 Equation solver

The flag `‘nocb‘` forces an internal solver of linear equation systems to be included in the source file and used for implicit equations. It is the default for variational or jacobian evaluations of implicit systems.

3.4.6 Integration options

The integrators of MATLAB version 5 support initialization, events, evaluation of the jacobian and other options. Support for this is generated by giving an equation with left hand side `‘v5‘`, and the rigth hand side a list of options. These options are either a symbol, or equations where the left hand side is a symbol, and the right hand side a list of the detailed arguments. Supported options are `‘jacobian‘`, `‘jpattern‘`, `init=[...]`, and `events=[...]`.

The option `‘init=[...]‘` is used to define what the function can perform besides of the normal evaluation of derivatives. The list is built up of three elements, of which the first two are rarely used. They hold, respectively, two numeric values that represent the time interval for integration that the function suggests if asked by the integrator, and numeric values suitable for initial values. Third is a list of names, where the names or symbols indicate what the function can perform, this means one or many of `‘events‘`, `‘jacobian‘`, or `‘jpattern‘`. A five degree of freedom system that can compute the jacobian could be given an option

```
‘init‘=[[0 1],[1.1 1.2 1.3 1.4 1.5],[‘jacobian‘]
```

Event handling is defined by an equation `‘events‘=[...]`, where the right hand list contains three lists. The first of these lists contains two lists that specifies the expressions which should trigger events. Of these two, the first holds definitions of subexpressions and the other hold the actual expressions. For each expression, there are two items to specify, the direction of passage through zero which triggers the event, and the action taken if the event occurs The second list hold integers, as many as the expressions, and they indicate if action should be taken or not, by ones and zeros. The third list also contains integers, but minus one, zero or one, which specifies directions which triggers events. Consider a two dimensional example where events occur if `q1` pass through zero from the positive side, or `q2` becomes zero regardless of direction. The integration should be stopped when `q2=0` and events triggered on `q1=0` should only be recorded. A valid option for this scenario is

```
‘events‘=[[[],[q1 q2]],[0 1],[-1 0]
```

The symbol `‘jacobian‘` forces `exmex` to include code for optional evaluation of the jacobian, for explicit and implicit equation systems as well as variational systems.

The symbol `‘jpattern‘` makes `exmex` include code for returning a conservative estimate of the pattern of non-zero entries of the jacobian of a variational system only.

To conclude,

```
‘v5‘=[‘init‘=[...],‘events‘=[...],‘jacobian‘,‘jpattern‘]
```

where each list contain appropriate definitions a generic option to `exmex`.

3.4.7 External functions

Inclusion at compilation time of files with externally defined function is made by an option of the form

```
‘includes‘=[‘"file1.c"‘,‘"file2.c"‘,...]
```

3.4.8 Variational system

The flags `‘variationalequations‘` or `variationalequations` declare that code for a variational system should be generated. It is not possible to generate Simulink blocks based on variational equations.

3.4.9 Derivatives of external functions

The option `‘diffset‘=[...]` as defined in section 2.4.2 defines names of derivatives of external functions.

3.4.10 Optimization

The options `‘notoptimized‘`, or `notoptimized`, forces `exmex` to not use the symbolic optimization but instead produce code direct from the given equations. This is not recommended, because the equations may be extremely large and the time reduction with symbolic optimization can be of several orders of magnitude.

3.5 MATLAB syntax

Suppose we have a generated function `func` with no parameters. Evaluation of the derivatives at time t and configuration x is made with a call `func(t,x)`. The third argument is reserved for flags, and parameters are passed as the fourth argument, `func(t,x,‘‘,p)`, where p is a vector with parameter values.

3.5.1 Flags

Flags are used to call for optional behaviour, we saw earlier the supported flags which are `'init'`, `'events'`, `'jacobian'`, and for certain systems `'jpattern'`. The call `[a1,a2,a3]=func([],[],'init')` would return in `a1` the suggested time period for integration, in `a2` a set of initial values, and in `a3` an object that specifies what flags can be processed, i.e. the most useful answer. Called with the flag `‘events‘` three vectors of equal size are returned. The first is the event value vector, which is a function of time and state. The two other vectors control the integrators use of the event value, termination of integration is specified in by the second and directional dependence of the third. More information about this is found in a manual for MATLAB.

The flag `'jacobian'` causes the function to evaluate the jacobian matrix, and the flag `'jpattern'` returns a sparse matrix with the general sparsity pattern of the jacobian of a variational system.

4 Examples

The many options and complex syntax is perhaps most easily explained by examples. Here are some demonstrations of the syntax and capability of the functions. In the examples, three different dynamical systems are investigated briefly to illustrate a some of the methods a dynamicist applies and how `exmex` can help in these investigations. The systems are used here because they have a limited number of variables, yet exhibit interesting properties.

4.1 Spherical pendulum

First we demonstrate the use on a set of equations governing the motion of a spherical pendulum with length L. The system is an accurate model of a Focault pendulum since we model it as located at a latitude λ on a planet rotating with a constant angular velocity Ω and gravitational acceleration g. A difference is that the hinge point is forced vertically with a harmonic function of amplitude a and angular frequency ω. The vertical forcing, as well as the rotation of the planet, is determined by parameters. Thus our equations can be used to investigate a large family of cases.

Dissipation is introduced as drag acting on the particle in the direction opposite to the instantaneous velocity. It is proportional to a constant c times the square of velocity relative to a frame rotating with the planet. The configuration is determined by time and the two coordinates q_1 and q_2, which are angles that point out the direction from the vertical in the north-south vertical plane, and the deviation from this plane respectively. The equations of motion are two second order equations rewritten to first order,

$$\begin{aligned}
&\cos(\mathit{q2})\,L(-\sin(\mathit{q1})\,m\,g - c\,\mathit{v1mag}\,(\sin(\mathit{q1})\,a\cos(\omega\,t)\,\omega + \cos(\mathit{q2})\,\mathit{u1}\,L)\\
&- m\cos(\mathit{q2})\,\mathit{u1t}\,L + 2\,m\sin(\mathit{q2})\,\mathit{u2}\,\mathit{u1}\,L + m\,a\sin(\omega\,t)\,\omega^2\sin(\mathit{q1})\\
&- 2\cos(\lambda)\sin(\mathit{q1})\,m\,\Omega\cos(\mathit{q2})\,\mathit{u2}\,L - \sin(\lambda)\cos(\mathit{q1})\,m\,\Omega^2\,a\sin(\omega\,t)\cos(\lambda)\\
&+ 2\sin(\lambda)\cos(\mathit{q1})\,m\,\Omega\cos(\mathit{q2})\,\mathit{u2}\,L + \cos(\mathit{q1})\,m\,\Omega^2\cos(\mathit{q2})\sin(\mathit{q1})\,L\\
&- 2\cos(\mathit{q1})\,m\,\Omega^2\cos(\mathit{q2})\sin(\mathit{q1})\,L\cos(\lambda)^2\\
&+ 2\sin(\lambda)\cos(\mathit{q1})^2\,m\,\Omega^2\cos(\mathit{q2})\cos(\lambda)\,L + \cos(\lambda)^2\sin(\mathit{q1})\,m\,\Omega^2\,a\sin(\omega\,t)\\
&- \cos(\lambda)\,m\,\Omega^2\cos(\mathit{q2})\sin(\lambda)\,L)
\end{aligned}$$

and

$$\begin{aligned}
&-L(\sin(\mathit{q2})\cos(\mathit{q1})\,m\,g - c\,\mathit{v1mag}\,(-\sin(\mathit{q2})\,a\cos(\omega\,t)\,\omega\cos(\mathit{q1}) - \mathit{u2}\,L) + m\,\mathit{u2t}\,L\\
&- \cos(\lambda)^2\cos(\mathit{q1})\,m\,\Omega^2\,a\sin(\omega\,t)\sin(\mathit{q2}) + 2\cos(\mathit{q2})^2\,m\,\Omega\sin(\lambda)\cos(\mathit{q1})\,\mathit{u1}\,L\\
&- 2\cos(\mathit{q2})\,m\,\Omega\cos(\lambda)\,a\cos(\omega\,t)\,\omega + \sin(\mathit{q2})\,m\,\mathit{u1}^2\cos(\mathit{q2})\,L\\
&- 2\cos(\mathit{q2})^2\,m\,\Omega\cos(\lambda)\sin(\mathit{q1})\,\mathit{u1}\,L - \cos(\mathit{q2})\,m\,\Omega^2\cos(\lambda)^2\sin(\mathit{q2})\,L\\
&- \cos(\mathit{q2})\,m\,\Omega^2\cos(\mathit{q1})^2\sin(\mathit{q2})\,L + 2\cos(\mathit{q2})\,m\,\Omega^2\cos(\mathit{q1})^2\sin(\mathit{q2})\,L\cos(\lambda)^2\\
&+ 2\sin(\mathit{q2})\sin(\lambda)\sin(\mathit{q1})\,m\,\Omega^2\cos(\mathit{q2})\cos(\lambda)\cos(\mathit{q1})\,L\\
&- \sin(\lambda)\sin(\mathit{q1})\,m\,\Omega^2\,a\sin(\omega\,t)\cos(\lambda)\sin(\mathit{q2}) - m\,a\sin(\omega\,t)\,\omega^2\cos(\mathit{q1})\sin(\mathit{q2}))
\end{aligned}$$

where the symbol `v1mag` is used to represent the magnitude of the velocity relative to a reference frame fixed in the planet. A definition of `v1mag` is stored in a variable `sx`,

$$\begin{aligned} sx := [v1mag = \mathrm{sqrt}((&\sin(q1)\, a \cos(\omega\, t)\, \omega + \cos(q2)\, u1\, L)^2 \\ &+ (-\sin(q2)\, a \cos(\omega\, t)\, \omega \cos(q1) - u2\, L)^2 + \cos(q2)^2 \cos(q1)^2\, a^2 \cos(\omega\, t)^2\, \omega^2)] \end{aligned}$$

Rewriting the equation to first order was with $u_1 = \dot{q}_1$ and $u_2 = \dot{q}_2$. The variable `ieq` holds the implicit equations while `sx` is used for the common subexpressions. A function that evaluates the derivatives based on the implicit equations is generated by the Maple code

```
fp:="/home/user/maple/"
eqs1:=[ieq,[u1t,u2t]]:
eqs2:=[[q1t=u1,q2t=u2]]:
varst:=[u1t,u2t,q1t,q2t]:
vars:=[u1,u2,q1,q2]:
ps:=‘parameters‘=[Omega,m,L,g,omega,a,lambda,c]:
exmex("pendi",fp,eqs1,eqs2,sx,varst,vars,ps);
```

We demonstrate numerical integration of the system with parameter values $\Omega = 0$, $m = 1$, $L = 1$, $g = 1$, $\omega = 30$, $a = 0.2$, $\lambda = 0$, and $c = 0.2$, over the time interval 0 to 4 time units. Initial values are $u_1 = u_2 = 0$, $q_1 = 0.5$, and $q_2 = 0.8$, and the following MATLAB commands perform the integration and plot time histories of the configuration.

```
ti=[0 4];
x0=[0 0 0.5 0.8];
p=[0 1 1 1 30 0.2 0 0.2];
[t,x]=ode45(’pende’,ti,x0,’’,p);
plot(t,x(:,3),’--’,t,x(:,4),’-’);grid on;
xlabel(’Time / s’)
ylabel(’Angle / rad’)
```

The result of the plot command is seen in figure 3.

Another interesting case of this problem illustrates how the results can be used to uncover further facts of the system. It is tempting to see the particle track over the floor for a freely moving pendulum on a rotating planet. This track depends on the configuration and can be computed when the time histories from an integration are available. The northerly position relative to the hinge point is $L\cos(q_2)\sin(q_1)$, and the easterly is $-L\sin(q_2)$. Plotting two cases with different damping can be done with the following code.

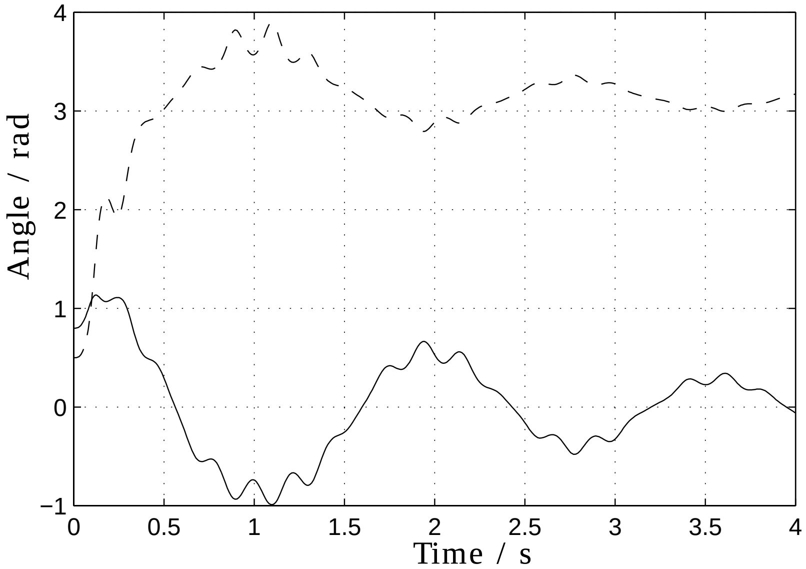

Figure 3: Time histories of the coordinates q_1 (dashed) and q_2 (continuous) for a case integrated in the text. From a dynamics viewpoint, the most interesting observation here is that q_1 tends toward a value of π. That means the vertical position with the mass above the hinge point is stable when a rapid enough forcing is applied to the hinge point, a well known fact and explained in [1] among others.

```
ti=[0 100];
x0=[0 0 0.5 0];
L=10;
p=[0.01 1 L 10 0 0 1.04 0.01];
[tx,x]=ode45('pende',ti,x0,'',p);
p=[0.01 1 10 10 0 0 1 0.001];
[ty,y]=ode45('pende',ti,x0,'',p);
subplot(1,2,1);
plot(-L*sin(x(:,4)),L*cos(x(:,4)).*sin(x(:,3)));grid on;
axis('equal')
subplot(1,2,2);
plot(-L*sin(y(:,4)),L*cos(y(:,4)).*sin(y(:,3)));grid on;
axis('equal')
```

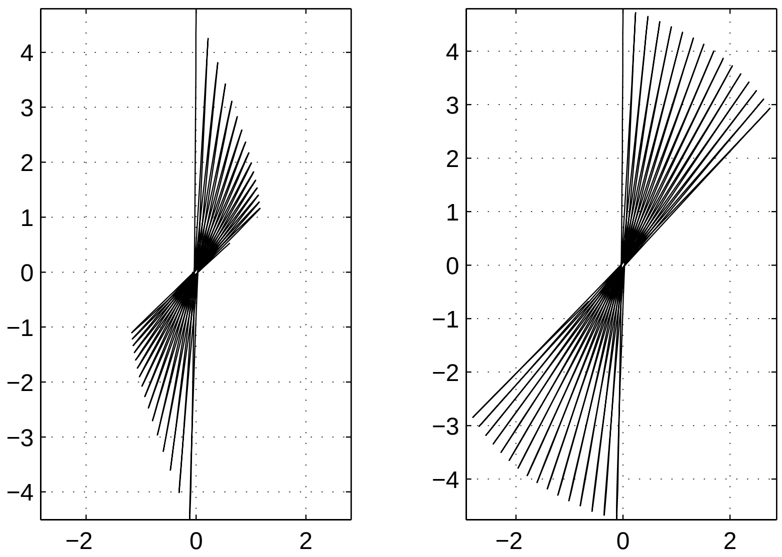

Figure 4: Track over floor of pendulum moving freely under a hinge point with all parameters defined in the previous code. It is located at a latitude corresponding to Stockholm's position. The left track is for a damping of $c = 0.01$, and the right is for $c = 0.001$.

4.1.1 A van der Pol oscillator

Our next example concerns a van der Pol oscillator governed by the two equations

$$\begin{aligned} \dot{x}_1 &= \mu\, x_1\, (1 - x_2^2) - x_2, \\ \dot{x}_2 &= x_1. \end{aligned}$$

In Maple this may be represented as a list of two explicit equations,

```
eqns:=[x1t=mu*x1*(1-x2^2)-x2,x2t=x1];
```

To produce a function that calculates the derivatives `x1t` and `x2t`, `exmex` could be invoked as follows,

```
fp:="/home/user/maple/"
xt:=[x1t,x2t]:
x:=[x1,x2]:
ps:=‘parameters‘=[mu]:
exmex("vdp",fp,[eqns],[[]],[],xt,x,ps);
```

A few integrations for a parameter value of $\mu = 1$ is done with the following MATLAB code, the corresponding plot is seen in figure 5.

```
[t,x]=ode45(’vdp’,[0 10],[0 1],’’,[1]);
plot(x(:,2),x(:,1),x(1,2),x(1,1),’o’);
grid on;hold on;
[t,x]=ode45(’vdp’,[0 10],[0 0.2],’’,[1]);
plot(x(:,2),x(:,1),x(1,2),x(1,1),’o’);
[t,x]=ode45(’vdp’,[0 10],[0 0],’’,[1]);
plot(x(:,2),x(:,1),x(1,2),x(1,1),’o’);
[t,x]=ode45(’vdp’,[0 10],[0 -0.2],’’,[1]);
plot(x(:,2),x(:,1),x(1,2),x(1,1),’o’);
[t,x]=ode45(’vdp’,[0 10],[2 -2],’’,[1]);
plot(x(:,2),x(:,1),x(1,2),x(1,1),’o’);
xlabel(’x2’);ylabel(’x1’);
```

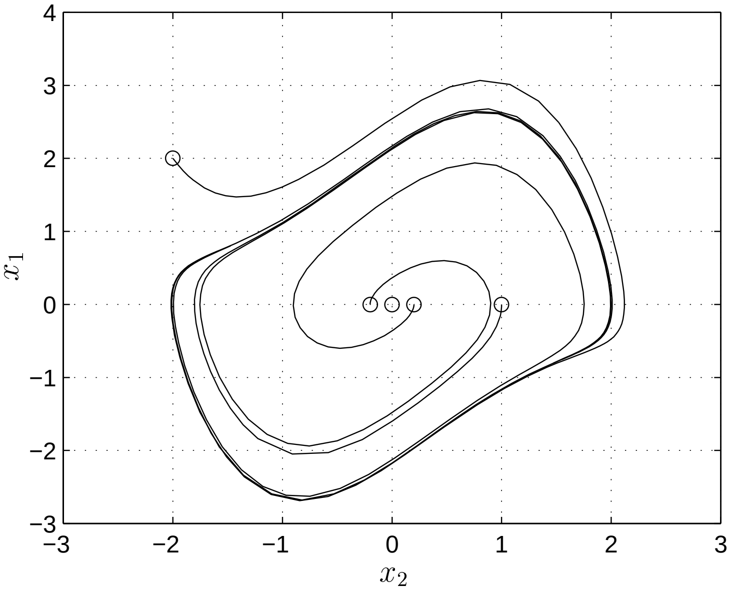

Figure 5: Phase section plot of a van der Pol oscillator for five different initial conditions, indicated by circles. All but the one starting in the origin tend to the stable limit cycle, whereas the origin seem to be a fixed point.

To get a quantitative measure on the stability of the apparent fixed point in the origin we generate another function, `vdpj`, that can compute the jacobian,

```
exmex(‘vdpj‘,fp,[eqns],[[]],[],xt,x,ps,‘v5‘=[‘jacobian‘]);
```

After compilation with `mex vdpj.c`, two lines of MATLAB code returns the eigenvalues at the origin,

```
>> vj=vdpj(0,[0 0],'jacobian',1)
vj =
     1    -1
     1     0
>> ev=eig(vj)
ev =
  0.50000000000000 + 0.86602540378444i
  0.50000000000000 - 0.86602540378444i
```

The real part of the eigenvalues are greater than zero and we conclude that the fixed point is unstable.

4.1.2 The Rössler system

The Rössler system is three ordinary differential equations,

$$\begin{aligned} \dot{x} &= -y - z && (4) \\ \dot{y} &= x + ay && (5) \\ \dot{z} &= b + z(x - c) && (6) \end{aligned}$$

where three variables a, b, and c are parameters. The system has limit cycles that become unstable and exhibit period doubling when the parameter a is changed. To investigate this we need to generate a function that includes the variational equations. For processing with Maple we store a representation of the equations in a list `rs`,

```
rs:=[xt=-y-z,yt=x+a*y,zt=b+z*(x-c)];
```

Generating a function `ross` for the base system is then accomplished with

```
fp:="./":
dl:=[xt,yt,zt]:
vl:=[x,y,z]:
ps:=`parameters`=[a,b,c]:
v5o:=`v5`=[`jacobian`]:
exmex("ross",fp,[rs],[[]],[],dl,vl,ps,v5o);
```

The function is also prepared to compute the jacobian. Use of the function `ross` in MATLAB could for instance an integration between times 0 and 100, followed by a plot of the results.

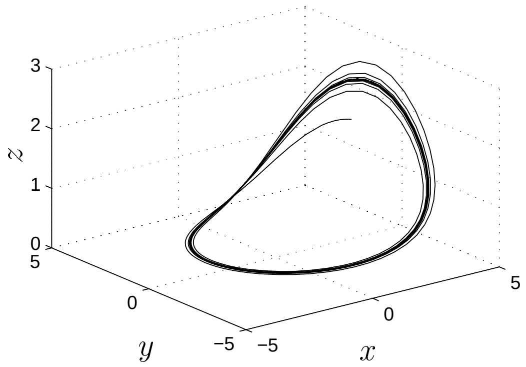

Figure 6: A trajectory of the Rössler system started in $x = 3$, $y = 0$, and $z = 2$. It seems to settle on a periodic orbit, that intersects the x-z plane close to the initial point.

```
x0=[3 0 2];
[t,x]=ode45(’ross’,[0 100],x0,’’,[0.3 2 4]);
plot3(x(:,1),x(:,2),x(:,3));grid on
xlabel(’x’);ylabel(’y’);zlabel(’z’);
```

After inspection of the plot in figure 6 it is plausible to suspect the existence of a stable periodic orbit. Two natural questions are how locate periodic orbits, and how to categorize them as stable or not. Stability is computed by integration of the variational equations along the orbit. We thus need a function that evaluates the variational system. Identification of passages through a Poincaré-section can be determined by event handling of the MATLAB integrators. For this problem we use the x-z plane as Poincaré-section and the condition for being on the surface is thus $y = 0$. A parameter `sf` is introduced that determines if the integration should be stoped or not by taking the values 0 or 1. This information is specified in the option `v5a`, where the third list defines the direction through the plane for which events are trigged. A suitable function can be generated in Maple by

```
veq:=‘variationaleqs‘:
pse:=‘parameters‘=[a,b,c,sf]:
v5a:=‘v5‘=[‘events‘=[[[],[y]],[sf],[1]]]:
exmex("rossve",fp,[rs],[[]],[],dl,vl,pse,veq,v5a);
```

The following MATLAB script solves for a point x_{s} on the surface $y = 0$ where the periodic orbits passes, and a plot of a few iterations are shown in figure 7. The method is Newton-Raphson iteration preceeded by an initialization of the variables needed during the iteration process.

```
% This script solves for periodic orbits in the Rössler system
io=odeset('AbsTol',1e-8,'RelTol',1e-6,'events','on');
% initial conditions
xn=[3 0 2];Xn=[xn reshape(eye(3),1,9)];
tn=6.3;
% parameter values
p=[0.3 2 4];P=[p 0];
% first integration and plot
[t,x,te,xe,ie]=ode45('rossve',[0 tn],Xn,io,P);
plot3(x(:,1),x(:,2),x(:,3),x(1,1),x(1,2),x(1,3),'o');grid on;hold on
% flow gradient and vector field at integration end point
m=length(t)
rj1=reshape(x(m,4:12),3,3);
f1i=rossve(t(m),x(m,:),'',P);
f1=f1i(1:3)
% equation system for dX
cm=[rj1-eye(3) f1 ; 0 -1 0 0];
rv=[xn'-x(m,1:3)' ; 0];
dX=cm\rv
% iterate until error small enough
while norm(dX)>10^(-6)
  % new initial conditions
  xn=xn+dX(1:3)'
  Xn=[xn reshape(eye(3),1,9)];
  tn=tn+dX(4)
  % integrate and plot
  [t,x,te,xe,ie]=ode45('rossve',[0 tn],Xn,io,P);
  plot3(x(:,1),x(:,2),x(:,3),x(1,1),x(1,2),x(1,3),'o');grid on;drawnow
  % flow gradient and vector field at integration end point
  m=length(t)
  rj1=reshape(x(m,4:12),3,3);
  f1i=rossve(t(m),x(m,:),'',P);
  f1=f1i(1:3);
  % equation system for dX
  cm=[rj1-eye(3) f1 ; 0 -1 0 0];
  rv=[xn'-x(m,1:3)' ; 0];
  dX=cm\rv
end
```

4.1.3 Simulink block

Code for S-functions to Simulink blocks is generated based on the contents of an option that specifies input variables, output expressions, and

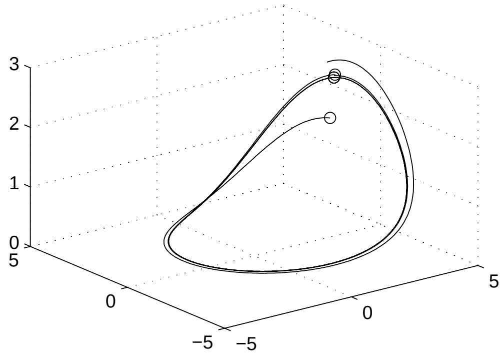

Figure 7: Iterations of periodic orbits in the Rössler system. Initial approximation is $x = 3$, $y = 0$, and $z = 2$, and the intersections with $y = 0$ are marked with a ring. After three iterations the components of the difference is on the order of 10^{-7}, and the solution is close to $x_\mathrm{s} \approx 3.1525486$, $z_\mathrm{s} \approx 2.6631716$, with a period of $t_\mathrm{s} \approx 6.1746375$.

subexpressions. We will not demonstrate computations with a function generated for Simulink, only give an example of arguments to `exmex`.

```
'simulink'=[XXXX]
```

4.2 Trademarks

Sun, MPW, THINK C, Maple, Mathematica, MATLAB, Macintosh, and others, are trademarks of their respective owner. `exmex` is available from the Sophia home page located at URL `http://www.mech.kth.se/sophia/`

References

[1] D. W. Jordan and P. Smith. *Nonlinear Ordinary Differential Equations*. Oxford University Press, 1986. ISBN 0-19-859656-1.