DETools: A Library for Differential Equations

Werner M. Seiler
Lehrstuhl I für Mathematik
Universität Mannheim, 68131 Mannheim, Germany
seiler@euler.math.uni-mannheim.de
http://iaks-www.ira.uka.de/iaks-calmet/werner/werner.html

We present a preview on a new MuPAD library called `detools` *containing a number of routines for symbolically (and numerically) treating differential equations. This includes especially a package for Lie symmetry analysis.*

Introduction

The next release of *MuPAD* will contain a number of new features for differential equations. Some of them are already contained in the current version 1.4.1, although only in an undocumented and experimental form. As an "official" documentation in the usual *MuPAD* format is still missing, this article tries to give at least a brief introduction.

The *MuPAD* library is extended on two levels. The first one consists of a new library `detools` containing a number of routines for treating differential equations. This includes support for the graphical presentation of the output of the numerical routines in *MuPAD*, some methods for analysing or generating differential equations and also routines for solving some classes of partial differential equations. The use of this new library will be described in this article.

The second level is somewhat more advanced and requires a certain familiarity with the object-oriented `domains` package of *MuPAD* [5]. Thus it addresses mainly programmers who want to implement sophisticated algorithms, but "ordinary" *MuPAD* users can profit of it, too. This level contains categories and domains providing unified data structures for differential equations. Superficially seen, this results just in more concise and comfortable in- and output. However, one of the most important aspects of providing such a programming environment is that it facilitates the simultaneous usage of different packages written by different authors.

There exist many computer algebra packages for differential equations implementing algorithms for many different tasks. As a simple example we may take the Lie symmetry analysis of a physical theory. Thus our starting point is a Lagrangian and the first step is the derivation of the corresponding Euler-Lagrange equations. The second step consists of setting up the determining system for the symmetry generators of these equations. As determining systems are usually rather large and overdetermined, one will probably need some automatic simplification and completion of this system. As fourth step one might try to automatically solve the determining system. In the fifth step one would like to use the found symmetry generators for symmetry reduction, preferably to some ordinary differential equations which should finally be solved automatically, too.

For each of these six steps there exist computer algebra packages (several are actually implemented in `detools`). However, they have usually been written by different authors. Furthermore, these authors may have had different tasks in their minds and not necessarily precisely the sequence of steps we described above. In any case, chances are high that each of the authors implemented his/her own data structures for differential equations, as no computer algebra system provides standardised ones.[1] If one wants to take the output of one package as input for another

[1] This is only reasonable, as for different tasks different data structures are optimal.

one, one must usually first write some conversion routines between these different data structures. This problem can be almost completely avoided by using the object oriented approach of the `domains` package of *MuPAD*.

The more advanced parts of the `detools` library are actually implemented using the categories and domains of the second level. Some of the library routines represent only interfaces taking input in the standard notation of *MuPAD* (i. e. `diff` or `D` for representing derivatives) and converting it to a suitable domain. Then the real computations take place within this domain and the output is finally converted back into the standard notation. This strategy allows users who have no experiences with domains to use the more advanced algorithms. But obviously the conversions need some time and thus the user can save computation time, if (s)he directly uses the appropriate domains for differential equations.

Details of the categories and domains implemented in the second level will be described elsewhere. This article concentrates on the methods contained in the library `detools` and their standard usage. Users familiar with the domains of the second level can however easily guess the correct arguments: whereas in the standard usage it is necessary to explicitly state the names of the independent and dependent variables, this information is already encoded in the domains, if one enters corresponding objects. Thus one may omit these arguments.

The `detools` Library

The methods of the `detools` library can roughly be grouped into the following five categories: (a) *solving* partial differential equations, (b) *generation*, *manipulation* and *analysis* of differential equations, (c) *visualisation* of numerical results, (d) support for *numerical analysis*, and (e) *symmetry analysis*. With the exception of the last one each of these groups will now be described in a separate subsection. The symmetry package is discussed in the next section.

In the sequel we will always assume that the methods of the `detools` library have been exported with the command `export(detools)`, i.e. we will write `method` instead of `detools::method`. As usual a list of all the methods included in the library can be obtained with `info(detools)`.

Solving Partial Differential Equations

The method `pdesolve` can determine the solution of some classes of partial differential equations. In this first version of the `detools` library it can handle essentially only *linear* equations, but a number of extensions are already planned. This includes especially general first order equations.

The solution of (the initial value problem for) linear first order equations is comparatively easy, as it can be reduced to solving a system of ordinary differential equations, the so-called *characteristic system*. As this system is in general non-linear, it may of course still be a formidable task to solve it. This also implies that the ability of `pdesolve` to find the solution depends decisively on the power of the *MuPAD* solver for ordinary differential equations (incorporated in the standard `solve` command). `pdesolve` tries to return an explicit expression for the solution. However, the method of characteristics yields the solution in a parametrised form and it might not always be possible to eliminate the parameters or to solve for the dependent variables in terms of the independent ones. In this case, `pdesolve` returns an implicit solution.

In the following *MuPAD* session the constant coefficients equation $2u_x + u_y + 3u_z - 2u = 0$ where $u = u(x, y, z)$ is solved subject to the initial condition that on the plane parametrically described by $x(s, t) = 2s$, $y(s, t) = 3t$, $z(s, t) = s + t$ the unknown function takes the value $u(s, t) = s - t$. `pdesolve` takes here five arguments. The first one is the differential equation; the second one consists of a list containing the names of the independent variables; the third one is a similar list for the dependent variables; the fourth argument is a set containing the Cauchy surface in parametric form and the fifth one is a set with the names of the parameters.

```
MuPAD
>> detools::pdesolve(2*u([x])+u([y])+3*u([z])-2*u,[x,y,z],[u],
>>                   {x=2*s,y=3*t,z=s+t,u=s-t},{s,t});
```

```
Output
                    / 7 x    y    2 z \     /   3 x    2 y    6 z  \
                u = | --- - - - - --- | exp| - --- - --- + ---  |
                    \ 10     5     5  /     \    5      5      5   /
```

In this example we indirectly used our domains for differential equations in order to simplify the input. Namely, we simply entered `u([x])` instead of the much more cumbersome `diff(u(x,y,z),x)`. We will continue to use this simplified input, as it is so much easier and does not require any knowledge of the domains involved. The rules are simple: for the dependent variables there is no need to enter the arguments (but it is not harmful, if one does it); for derivatives just give a list of the independent variables with respect to which one differentiates as argument of the dependent variable.

Mathematical details on the method of characteristics can be found in any standard textbook on partial differential equations. Characteristics are also of importance for the classification of linear partial differential equations. Therefore the method `characteristics` is provided with which it is possible to work directly with the characteristics. Its arguments are very similar to `pdesolve`. The first three ones are identical; the fourth one gives the name of an identifier used to parametrise the characteristic curves; the fifth one gives again the initial conditions in parametrised form.

The following command determines the characteristic curves for the variable coefficients equation $2u_x + xu_y + 3u_z - u = 0$ with the same initial conditions as above and `r` as curve parameter.

```
MuPAD
>> detools::characteristic(2*u([x])+x*u([y])+3*u([z])-u,[x,y,z],[u],r,
>>                         {x=2*s,y=3*t,z=s+t,u=s-t});
```

```
Output
                             2
{{y(r) = 3 t + 2 r s + r , u(r) = exp(r) (s - t), x(r) = 2 r + 2 s,

   z(r) = 3 r + s + t}}
```

It is also possible to use `characteristics` to set up the characteristic system without trying to solve it. In this case no initial conditions should be given; instead the option `Solve=FALSE` must be specified. The characteristic system is returned as an object of type `ode` (see the online documentation).

```
MuPAD
>> detools::characteristic(2*u([x])+x*u([y])+3*u([z])-u,[x,y,z],[u],r,Solve=FALSE);
```

```
Output
ode({diff(u(r), r) = u(r), diff(y(r), r) = x(r), diff(x(r), r) = 2,

   diff(z(r), r) = 3}, {u(r), x(r), y(r), z(r)})
```

Currently `characteristics` can handle only single first order equations. But an extension to linear systems and higher order equations is under development.

With the option `Polynomial=[N,str]` `pdesolve` is instructed to look only for polynomial solutions. Currently, this is only implemented for linear systems with constant coefficients. `pdesolve` further assumes that the given system is complete or involutive, i. e. that it contains already all its non-trivial integrability conditions (that is an issue only for over-determined systems). The integer `N` prescribes the maximal degree of the polynomials; the string `str` is used for the names of the coefficients of the polynomials. `str` can be omitted; in this case `"a"` is used as default value and the option is simplified to `Polynomial=N`.

In the following example polynomial solutions of maximal degree 3 of the heat equation $u_t = u_{xx}$ are determined.

```
MuPAD
>> detools::pdesolve(u([t])-u([x,x]),[x,t],[u],Polynomial=[3,"c"]);

   Output

           / / c5 \  3   / c6 \  2                                   \
       poly| | -- | x  + | -- | x  + c5 x t + c3 x + c6 t + c1, [x, t] |
           \ \ 6  /      \ 2  /                                      /
```

Note that `pdesolve` returns here an object of type `DOM_POLY` (with coefficient ring `Expr`). The unknowns are just the independent variables as passed to `pdesolve`. In our example the solution contains four arbitrary parameters `c1`, `c3`, `c5`, `c6`.

Generating, Manipulating and Analysing PDEs

The method `euler` implements the *Euler operator* for Lagrangians of arbitrary order and of an arbitrary number of independent and dependent variables. Thus it allows to determine the *Euler-Lagrange equations* for variational problems. This is for example of great importance in physics where the equations of motion of almost everything from a simple mechanical system to a gauge theory describing the interaction of elementary particles are derived via variational principles.

Let

$$I[\phi_\alpha] = \int \mathcal{L}\left(x_i, \phi_\alpha(x_i), \frac{\partial^{|\mu|}\phi_\alpha}{\partial x^\mu}(x_i)\right) dx \tag{1}$$

be a functional of the functions $\phi_\alpha(x_i)$ where $\alpha = 1, \ldots, m$ and $i = 1, \ldots, n$ defined by the Lagrangian $\mathcal{L}$ depending on the independent variables x_i, the functions ϕ_α and their partial derivatives up to some order q. $\mu = [\mu_1, \ldots, \mu_n]$ represents a multi-index of length $|\mu| = \sum \mu_i$ in the usual notation of multivariate calculus. Then a necessary condition for the functions ϕ_α to yield an extremum of the functional I is that they satisfy the Euler-Lagrange equations:

$$\sum_{a=0}^{q}(-1)^a \sum_{|\mu|=a} \frac{\partial^a}{\partial x^\mu}\left(\frac{\partial \mathcal{L}}{\partial \frac{\partial^a \phi_\alpha}{\partial x^\mu}}\right) = 0\,. \tag{2}$$

The left hand side of (2) defines the component α of the Euler operator $E[\mathcal{L}]$. Generally the Euler-Lagrange equations form a system of m equations of order $2q$; however in special cases the order of at least some equations may be lower.

The method `euler` takes as arguments the Lagrangian $\mathcal{L}$, the list of the independent variables x_i and the list of the dependent variables or fields ϕ_α. The output consists of a list with the components of the Euler operator as defined in (2). As a first example we derive the equations of motion for a particle moving in a central force field. Its coordinates are given by (x, y) and the time variable is denoted by t. As potential we simply take $V(x, y) = 1/r$ with $r^2 = x^2 + y^2$. Thus we need the Euler-Lagrange equations for the Lagrangian $L = (\dot{x}^2 + \dot{y}^2)/2 - 1/r$.

MuPAD

```
>> detools::euler((x([t])^2+y([t])^2)/2-1/sqrt(x^2+y^2),[t],[x,y],Expr=NoDiff);
```

Output

```
--                 x                          y          --
|  x([t, t]) - ------------, y([t, t]) - ------------   |
|                2    2 3/2                2    2 3/2   |
--             (x  + y )                 (x  + y )     --
```

Here we used again our simplified notation for the input. Alternatively, we could have written `diff(x(t),t)` instead of `x([t])`. The option `Expr` specifies that we want the output in the basic type DOM_EXPR. By using `Expr=NoDiff` instead of `Expr=TRUE` we furthermore required the output in our simplified notation instead of the standard notation using `diff`. As an example for a field theory we take the ϕ^4-theory in $(1+1)$ dimensions (this time without `NoDiff`).

MuPAD

```
>> detools::euler((Phi([t])^2-Phi([x])^2)/2-Phi^4/4,[t,x],[Phi]);
```

Output

```
                                                            3
diff(Phi(t, x), t, t) - diff(Phi(t, x), x, x) + Phi(t, x)
```

The two methods `hasHamiltonian` and `hasPotential` check whether a given (autonomous) system of first order ordinary differential equations (or better: a given vector field) has certain properties, namely in the first case whether it is a Hamiltonian system and in the second case whether it is a gradient system, i. e. a system derivable from a potential. In both cases the call returns a list of integrability conditions which must be satisfied for the property in question.

In a first example we check whether the harmonic oscillator is a Hamiltonian system. Thus we take the differential system $\dot{q} = p$, $\dot{p} = -q$ where q denotes the space and p the momentum coordinate. `hasHamiltonian` requires as arguments three lists: the first one contains the components of the vector field, i. e. the right hand sides of the differential equations; the second one gives the names of the space coordinates and the third one the names of the momentum coordinates.

MuPAD

```
>> detools::hasHamiltonian([p,-q],[q],[p]);
```

Output

```
[]
```

As an empty list is returned, the system is unconditionally Hamiltonian. Next we study a system with two degrees of freedom, namely $\dot{x} = p_x$, $\dot{y} = p_y$, $\dot{p}_x = F(x, y)$ and $\dot{p}_y = G(x, y)$ where F and G are yet arbitrary functions of their arguments.

MuPAD

```
>> detools::hasHamiltonian([px,py,F(x,y),G(x,y)],[x,y],[px,py]);
```

Output

```
[diff(F(x, y), y) - diff(G(x, y), x)]
```

Now the result indicates a condition on the functions F and G, if the system is to be Hamiltonian. This condition $F_y = G_x$ just says that the force vector $(F, G)^t$ must be the gradient of a potential. Thus it is not surprising that in the following example we obtain exactly the same results.

MuPAD

```
>> detools::hasPotential([F(x,y),G(x,y)],[x,y]);
```

Output

```
[diff(F(x, y), y) - diff(G(x, y), x)]
```

Note that `hasPotential` requires only two arguments, as we have here no distinction into space and momentum coordinates. One should furthermore remark that `hasHamiltonian` checks only whether the system is Hamiltonian with respect to the canonical symplectic structure of phase space.

Visualisation

The two methods `odeplot` and `phaseplot` both take the output of the routine `numeric::odesolve` with the option `Alldata` (for a detailed description of this and other numerical routines see the online documentation) for numerically solving first order systems of ordinary differential equations and represents it graphically using the `VCam`, *MuPAD*'s graphics tool.

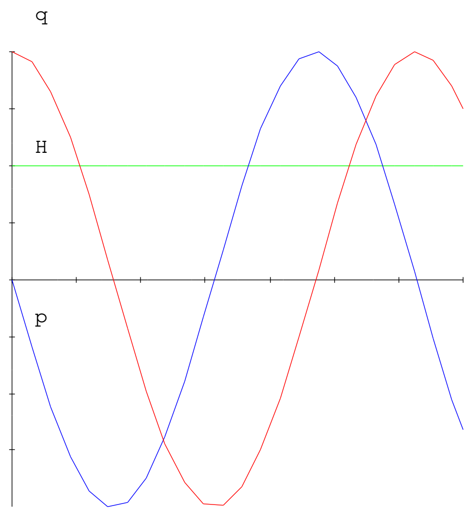
`odeplot` plots components of the solution vector over the independent variable. The components are given as a list of indices; their number can be arbitrary. Additional graphics options may be given in order to obtain a nicer picture. As third argument a list with object options may be given; as fourth argument a list with scene options. The list for the object options should actually be a list of lists; each sublist containing the object options for one of the generated curves.

As a simple example we take again the harmonic oscillator. We first integrate its equation of motion with a Runge-Kutta-Fehlberg method of order 8, then we plot position and momentum. The option `Plot` given as last argument to `odeplot` specifies that the plot should actually be rendered. Without it only a list of graphic objects would be returned which can be combined with other ones for more complex diagrams (see below for an example).

MuPAD

```
>> F := proc(t,y) begin [y[2],-y[1]] end_proc:
>> H := proc(t,y) begin 0.5*(y[1]^2+y[2]^2) end_proc:
>> nsol := numeric::odesolve(0..7,F,[0,1],RKF78,Alldata):
>> detools::odeplot(nsol,[1,2,H],
>>         [[Title="q",TitlePosition=[2.5,0.6],Color=[Flat,[1,0,0]]],
>>          [Title="p",TitlePosition=[2.5,5.6],Color=[Flat,[0,0,1]]],
>>          [Title="H",TitlePosition=[2.5,2.8],Color=[Flat,[0,1,0]]]],
>>         [Axes=Origin,Ticks=7,Title="Harmonic Oscillator",TitlePosition=Below,
>>          Scaling=UnConstrained,FontSize=15,ForeGround=[0,0,0],BackGround=[1,1,1]],
>>         Plot):
```

Graphics



Harmonic Oscillator

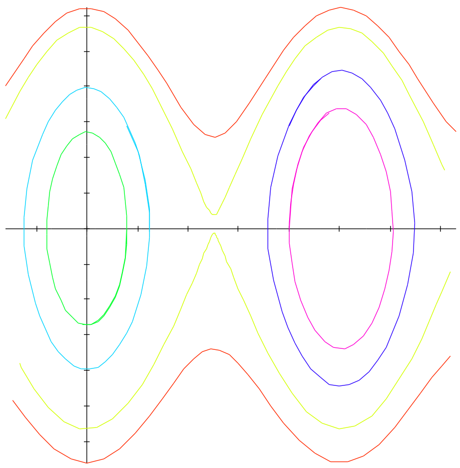
odeplot is also able to plot functions of the solution vector. For this a procedure is passed instead of an index. This procedure must take as input one float and one float vector of the dimension of the solution vector and return as output a simple float. In our example above we used this to check visually that the numerical integration preserved the energy. We defined a procedure H computing the energy at a given point of the trajectory and indeed the corresponding curve is a horizontal line.

phaseplot generates phase diagrams, i. e. it plots one or two components of the solution over another component (in the case of two components a 3D plot is generated). Again the components are specified as indices and like for odeplot it is also possible to pass a procedure instead of an index. The specification of plot options is similar to odeplot with the sole difference that only one list of object options can be passed. We will now use a sequence of calls of numeric::odesolve and phaseplot (without the option Plot) for various initial conditions to generate the phase portrait of a pendulum.

MuPAD

```
>> F := proc(t,y) begin [0.5*y[2],-sin(y[1])] end_proc:
>> ICs := [[10,[-2,2]],    [16,[-2,1.54]],    [12,[1,0]],    [12,[1,1.45]],
>>         [12,[5,1.45]],    [12,[5,0]],    [8.5,[9,-1.8]],    [19,[9,-0.6]]]:
>> nsol := map(ICs,
>>     proc(l,f) begin numeric::odesolve(0..l[1],f,l[2],RKF78,Alldata) end_proc,
>>     F):
>> out := map(nsol,detools::phaseplot,[1,2]):
>> plot2d(Axes=Origin,Title="Pendulum Phase Portrait",TitlePosition=Below,
>>     FontSize=15,ForeGround=[0,0,0],BackGround=[1,1,1],Scaling=UnConstrained,
>>     op(out)):
```

Graphics



Pendulum Phase Portrait

Note that `odeplot` and `phaseplot` are not very sophisticated methods. They will simply plot the points as computed by `numeric::odesolve`. No attempt is made to smoothen the resulting curves. It is assumed that in regions where the solution rapidly changes `numeric::odesolve` has taken sufficiently small steps to produce a reasonable output. Thus the smoothing is basically "controlled" by the choice of the numerical method and by the setting of the variable `Digits`.

Numerical Analysis

As already mentioned above `numeric::odesolve` implements a number of methods for the numerical solution of systems of ordinary differential equations. The new library `detools` contains a procedure `modode` which supports the analysis of such methods.

`modode` constructs a so-called *modified* equation for a given numerical method for solving ordinary differential equations. Modified equations represent a powerful tool for the backward error analysis of a numerical methods; a detailed introduction can be found in [7]. Loosely spoken, given a differential equation and an approximate solution of it computed with some numerical method, a modified equation is another differential equation whose exact solution is closer to the approximate solution than the exact solution of the original equation.

Thus the modified equation may be considered as a continuous model for the application of the numerical method to the original equation. Besides other possibilities it allows to study whether the numerical method preserves the qualitative features of the original equation. For example, given a Hamiltonian system and a numerical method to integrate it, one can check whether the modified equation is also a Hamiltonian system. In general, this will only be the case, if a symplectic integrator is used.

`modode` computes modified equations in the form of a Taylor polynomial in the step size h of the numerical method. Its implementation is based on the MAPLE program present in [1]; however, it contains some minor improvements which make it more efficient. The algorithm uses a brute force Taylor series expansion and is rather straight-forward. We consider in this article only explicit one step methods which can be written in the form

$$\vec{y}_{n+1} = \vec{y}_n + h\vec{\Psi}(h, t, \vec{y}_n) \tag{3}$$

where $\vec{y}$ denotes the vector of the unknown functions, t the independent variable and h the step size. For the analysis of implicit methods or for the incorporation of variable step sizes see [1].

There are two ways to use `modode`. The generic way is to pass the following arguments: (i) a list containing the components of $\vec{\Psi}$, (ii) a list with the dependent variables, i. e. the components of $\vec{y}$, (iii) the independent variable t, (iv) the step size h and (v) the order N up to which the series for the modified equation should be evaluated. The following example computes the modified equation for the Euler method $y_{n+1} = y_n + hf(t, y_n)$ applied to the simple equation $\dot{y} = f(t, y) = y$ up to order 4. Note that although we are treating a single equation here, we must pass it in form of a list like for a system. Similarly, the output is always in form of a list.

MuPAD

```
>> detools::modode([y],[y],t,h,4);
```

Output

```
--              2       3       4    --
|      h y   h y     h y     h y     |
|  y - --- + ---- - ---- + ----      |
--      2      3       4       5     --
```

The second possibility makes use of the `symbolic` option of `numeric::odesolve`. Thus `modode` let *MuPAD* derive the function Ψ appearing in (3). One passes now as arguments a procedure encoding the differential equation as it is expected by `numeric::odesolve` and a string with the name of the numerical method one wishes to analyse. Obviously, this approach is restricted to the methods implemented in the `numerics` library.[2] Thus the following call produces exactly the same result as the previous one.

MuPAD

```
>> F := proc(t,z) begin [z[1]] end_proc:
>> detools::modode(F,EULER1,[y],t,h,4);
```

Note here that one can give in the call of `modode` new names for the dependent variables independent of those used in the procedure `F`. While this procedure always requires a list as second argument (`z` in our example), we can later on give individual names to the components of the vector (`y` for `z[1]`).

Finally, we want to combine `modode` with `odeplot` in order to show that the modified equation is indeed a better approximation to the numerical solution than the original equation. As in [1] we take as example the equation $\dot{y} = y^2$. For the initial condition $y(0) = 1$ its exact solution is $y(t) = 1/(1-t)$, as one can easily compute with `solve`. With the following *MuPAD* commands we generate a plot comparing the exact solution, the numerical solution obtained by the Euler method with step size $h = 0.1$, and a high accuracy numerical solution of the fourth order modified equation for this problem.

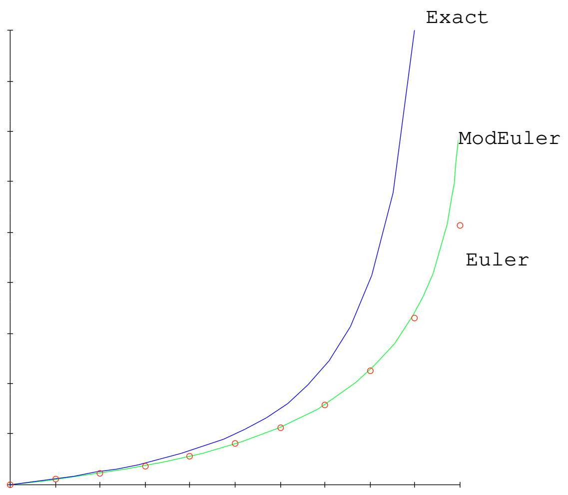
MuPAD

```
>> F := proc(t,y) begin [y[1]^2] end_proc:
>> eulersol := numeric::odesolve(0..1,F,[1],EULER1,Stepsize=0.1,Alldata):
>> euler := [Mode=List,map(eulersol,((l)->point(l[1],l[2][1]))),
```

[2] For further details and for the methods available see its online documentation.

```
>>          PointStyle=Circles,PointWidth=40,Title="Euler",TitlePosition=[8.5,5]]:
>> exsol := op(solve(ode({y'(t)=y(t)^2,y(0)=1},y(t))),1):
>> exact := [Mode=Curve,[t,exsol],t=[0,0.9],
>>          Title="Exact",TitlePosition=[8,1],Color=[Flat,[0,0,1]]]:
>> mde := detools::modode(F,EULER1,[z],t,h,4)[1]:
>> ModF := proc(t,y) begin [subs(mde,h=0.1,z=y[1])] end_proc:
>> modsol := numeric::odesolve(0..0.95,ModF,[1],RKF78,Alldata):
>> mod4 := detools::odeplot(modsol,[1],[[Title="ModEuler",TitlePosition=[9,3]]],[]):
>> plot2d(Title="Modified Equation for Euler Method",TitlePosition=Below,
>>     FontSize=15,ForeGround=[0,0,0],BackGround=[1,1,1],Scaling=UnConstrained,
>>     euler,exact,mod4[1]):
```

Graphics



Modified Equation for Euler method

The first line defines our differential equation in the form required by `numeric::odesolve`. Then we compute a numerical solution with the Euler method with fixed step size $h = 0.1$. The third command converts the output into a graphic object. The next two lines generate a curve for the exact solution. The following four commands generate a curve for the numerical solution of the fourth order modified equation: firstly, the modified equation `mde` is determined; then it is converted into a procedure `ModF` as required by `numeric::odesolve`; the final step generates a corresponding graphic object. Finally, `plot2d` is called (note that `mod4` is a *list* of objects, hence we must use `mod4[1]`).

The output clearly shows that the solution of the modified equation almost interpolates between the points obtained by the Euler method. In contrast the exact solution of the original differential equation grows much more rapidly (the Euler method completely misses the singularity at $t = 1$). This shows that it indeed makes sense to consider modified equations as a continuous model for numerical methods.

A Lie Symmetry Package

Some Background

Sophus Lie's symmetry theory represents one of the most powerful approaches to differential equations. Especially for nonlinear equations it often provides the only possibility for finding closed form solutions. But symmetries have also many other applications like the construction of conservation laws, separation ansätze or the classification of differential equations. It is beyond the scope of this article to give a detailed introduction into Lie theory, but there exists by now a number of excellent textbooks [4, 10, 18] where the reader can find in depth information both on theoretical and practical aspects.

After Lie's death symmetry theory was almost forgotten for a long time. One reason for this was definitely that it is computationally very demanding. Determining systems containing several hundred of equations are not exceptional. Thus it is not surprising that already very early in the history of computer algebra the first packages for the symmetry analysis of differential equations emerged. Today it is almost impossible to maintain an overview on the many existing packages; comparatively recent surveys can be found in [8, 9].

MuPAD has been the only one among the major general purpose computer algebra system without a symmetry package. Therefore the first larger component of the `detools` library is such a package. In this first version it contains basically only methods for setting up determining systems; later versions will also provides methods supporting their automatic solution and the subsequent use of the found symmetries.

Our package can handle not only classical point symmetries but also *generalised* (or *Lie-Bäcklund*) symmetries, as this represents a fairly straightforward extension of the basic algorithm for setting up determining systems. However, we do not think that this represents a very useful approach to generalised symmetries and thus not much time has been invested in the corresponding methods. For alternative approaches and their implementation in *MuPAD* we refer to the recent work of Fuchssteiner et al. [6].

On the other hand we provide probably one of the most general implementation for the so-called *non-classical method* going beyond the usual Bluman-Cole approach. In principle, this allows us to determine every solution of a given system of differential equations via a symmetry reduction. However, in practice one is usually not able to obtain all solutions of the arising determining systems.

Classical Symmetries

As already said we cannot give an introduction to symmetry theory. For this we refer again to the above mentioned textbooks. Here we only sketch the basic concepts. The most general definition of a symmetry of a differential equation is that of a transformation mapping solutions into solutions. In the case of classical symmetries one considers only point transformations, i. e. transformations of the independent and dependent variables that do not depend on derivatives.

A more geometric picture can be given as follows. If we consider the derivatives in a differential equation as further unknowns, we may say that the equation defines a hypersurface in an extended space. A symmetry is now a transformation that leaves this hypersurface invariant. In order to relate this geometric idea with the definition above, we must extend the point transformations to transformations in this extended space. But this is a straightforward exercise in using the chain rule.

In general, it is very difficult to find such point transformations that leave a differential equations invariant, as this leads to very complicated non-linear equations. The decisive insight of Lie was that without (well, almost) loss of generality one can linearise the problem by considering only infinitesimal transformations; in a more abstract language one can pass from the Lie group to the Lie algebra. He furthermore provided an effective criterion for deciding whether or not a given vector field generates a symmetry group.

The main steps of determining the symmetries of a given differential equations are: (a) making an ansatz for the symmetry generators, (b) prolonging this ansatz to the order of the equation (this means finding the generator in the extended space), (c) evaluating the symmetry criterion. The last step yields the so-called *determining system* for the coefficients of the ansatz. It is an in general very over-determined linear system of partial differential equations.

Steps (b) and (c) are very tedious and for them the `detools` library provides support in form of the method `detSys`. Step (b) will usually be invisible for the user, as it is performed in a domain for vector fields. As a concrete example we consider the heat equation $u_t = u_{xx}$. The basic call for users who do not want to have anything to do with domains is:

```
MuPAD
>> detools::detSys(u([t])-u([x,x]),[t,x],[u],Expr=NoDiff);
Output
[XI2([x]), XI2([t]) + (-2) XI1([x]), XI1([u]), XI2([u]),

   PHI1([x, x]) + (-1) PHI1([t]), PHI1([x, u]) + (-1/2) XI1([x, x]) +

   1/2 XI1([t]), PHI1([u, u])]
```

The first three arguments specify as for other methods in the `detools` library the differential equations, the independent and the dependent variables, respectively. The option `Expr` specifies again that the output should be of the basic type `DOM_EXPR`. As we used `Expr=NoDiff` instead of `Expr=TRUE`, the output is furthermore in our simplified notation.

As no special ansatz for the symmetry generator has been given to `detSys`, it has used the most general one $v = \tau(t,x,u)\partial_t + \xi(t,x,u)\partial_x + \eta(t,x,u)\partial_u$. As default the names `XI1`, `XI2`, ... are used for the coefficients in direction of the independent variables and `PHI1` etc. for the dependent variables. This can be changed by prescribing an ansatz. Thus in the following call we indeed use τ, ξ, η for the coefficients. This requires the setting of two options: `Ansatz` for the ansatz itself and `Coeff` for a list of the parameters of the used ansatz. The ansatz is given as a list of lists; each sublist consists of a coefficient and a variable indicating the basis vector (i. e. `t` for ∂_t and so on).

```
MuPAD
>> detool::detSys(u([t])-u([x,x]),[t,x],[u],
>>                Ansatz=[[tau(t,x,u),t],[xi(t,x,u),x],[eta(t,x,u),u]],
>>                Coeff=[tau,xi,eta],Expr=NoDiff,Unsimplified=TRUE);
Output
[2 tau([u]), 2 tau([x]), tau([u, u]), 2 tau([x, u]) + 2 xi([u]),

   tau([x, x]) + (-1) tau([t]) + 2 xi([x]), xi([u, u]),

   (-1) eta([u, u]) + 2 xi([x, u]), (-2) eta([x, u]) + xi([x, x]) +

   (-1) xi([t]), (-1) eta([x, x]) + eta([t])]
```

Here we set additionally the option `Unsimplified`. This instructs `detSys` to return the determining system in "raw" form. By default, detSys automatically performs some simplification (a so-called auto reduction) in order to eliminate redundant equations. Comparing the output of the two calls one can clearly see the effect of this option even for this small example. The number of equations has been reduced from 9 to 7 and the number of terms from 16 to 11.

detSys suffers from the same restrictions as other symmetry packages. Its result can be incorrect, if it is applied to differential equations which are not polynomial in the derivatives. This will be the case, if the appearing transcendental terms are not algebraically independent. detSys must also be able to solve each equation for a derivative; by default it tries to solve for the leading derivative. Alternatively, it is possible to prescribe with the option Solve for each equation a derivative for which it can be solved.

Generalised symmetries [2, 10] are not so easy to understand from a geometric point of view. Computationally however, not much changes. The transformations of the independent and dependent variables are now allowed to depend on derivatives, too. This leads to only minor modifications in the algorithm for setting up the determining system. However, generalised symmetries are most important for evolution equations, i. e. differential equations of the form $u_t = K(u, u_x, u_{xx}, \ldots)$. Here one can use completely different techniques for finding them. Some of them have been implemented in *MuPAD* by Fuchssteiner et al. [6]. The detools library nevertheless provides a method genDetSys for setting up the determining system for generalised symmetries of a prescribed order.

The main difference between detSys and genDetSys is that the latter one uses by default a vertical ansatz for the symmetry generator, i. e. an ansatz of the form $v = Q(x, t, u, u_t, u_x, \ldots)\partial_u$. The order of the ansatz, i. e. the maximal order of the derivatives appearing in the coefficient Q is specified by the option GenOrder.

```
MuPAD
>> detools::genDetSys(u([t])-u([x,x]),[t,x],[u],GenOrder=1,
>>                     Expr=NoDiff,Unsimplified=TRUE);
```

```
Output
[(-1) Q1([u([t]), u([t])]), ((-2 u([t]))) Q1([u([x]), u([t])]) +

   ((-2 u([x]))) Q1([u, u([t])]) + (-2) Q1([x, u([t])]),

               2
   ((- u([t]) )) Q1([u([x]), u([x])]) +

                                                  2
   ((-2 u([t]) u([x]))) Q1([u, u([x])]) + ((- u([x]) )) Q1([u, u]) +

   ((-2 u([t]))) Q1([x, u([x])]) + ((-2 u([x]))) Q1([x, u]) +

   (-1) Q1([x, x]) + Q1([t])]
```

Non-classical Symmetries

One of the most important applications of symmetries is the so-called *symmetry reduction*. Here we must distinguish between ordinary and partial differential equations. If we know a point symmetry of an ordinary differential equation, we can construct a reduced equation whose order is one less than that of the original equation. If we can find the general solution of the reduced equation, the general solution of the original equation can be obtained from it with one quadrature.

For partial differential equations the situation is very different. Here the reduced equation is not of lower order but has less independent variables. Furthermore it is *not* possible to construct the general solution of the original equation from the general solution of the reduced equation. Thus symmetry theory yields here only special, namely *group invariant solutions*.

Only rather recently it was fully realized that group invariant solutions exist also for groups which are *not* symmetry groups of the differential equations in the sense defined above. One started to distinguish between strong and weak symmetry groups [11, 13, 17]. *Strong symmetries* are symmetries in the sense we have used so far; *weak symmetries* are symmetries for which it is possible to perform symmetry reductions. The naming is somewhat misleading, as opposed to what the names seem to indicate not every strong symmetry is also a weak one!

With the introduction of weak symmetries one starts to move away from the original geometric ideas and gets more into the field of solvability of over-determined systems. If a solution is to be invariant under the group generated by $v = \tau\partial_t + \xi\partial_x + \eta\partial_u$, it must satisfy the corresponding *invariant surface condition* $\xi u_t + \tau u_x = \eta$. Hence group invariant solutions must simultaneously solve this equation and the original differential equation.

The determining system arises now from the condition that this over-determined system possesses solutions. In the so-called Bluman-Cole method one requires that the arising integrability conditions are identically satisfied (one says that the over-determined system is *compatible*). But this is a stronger requirement than necessary. It suffices that the over-determined system is *consistent*, i. e. that it has solutions. Thus if the first integrability conditions are not satisfied, we may add them to our system and check when this larger system is compatible. This way we may continue until we find either a compatible or an inconsistent system.

The procedure `ncDetSys` takes this situation into account. The user can prescribe with the option `Steps` at what step (s)he wants the integrability conditions to be satisfied. Thus the Bluman-Cole approach is obtained by specifying `Steps=1` (which is the default value) and we can recover the famous analysis of the heat equation [3] with the following line where we use a simplified ansatz for the vector field assuming that $\tau(t, x, u)$ does not vanish and can therefore be set to 1.

MuPAD

```
>> detools::ncDetSys(u([t])-u([x,x]),[t,x],[u],Expr=NoDiff,Steps=1,
>>                   Ansatz=[[1,t],[xi(t,x,u),x],[eta(t,x,u),u]],Coeff=[xi,eta]);
```

Output

```
--
|  - eta([t]) + eta([x, x]) - 2 eta xi([x]) ,
--

   xi([t])   xi([x, x])
   ------- - ---------- + eta([x, u]) + xi xi([x]) - eta xi([u]),
      2          2

                                                                  --
   xi([u, u]), - 2 xi([x, u]) + eta([u, u]) + 2 xi xi([u])   |
                                                                  --
```

Non-classical symmetries face two main obstacles. As one can clearly see from the output of our last example, the determining systems become non-linear and thus much harder to solve. Furthermore the higher one sets the value of `Steps` the less and the more complicated equations one gets. In some sense one thus looses the main reasons for the effectivity of Lie theory! Nevertheless non-classical reductions have proven to be a valuable tool for constructing solutions to partial differential equations.

Conclusions and Outlook

This article has given a preview on the new detools library which will be contained in the next release of *MuPAD*. It has described the current status of the library which is only at the very beginning of its development. Obviously, with the exception of the symmetry package the detools library represents currently a fairly accidental collection of methods which were developed on different occasions for different tasks. Most of them are not very sophisticated but rather small utilities.

A number of extensions are currently under development and will hopefully be included in the near future. The major points we are working at right now are

1. Extending the capabilities of pdesolve: (a) generalising the method of characteristics to arbitrary first order equations, (b) some heuristics for the at least partial solution of linear systems as the arise in symmetry analysis, and (c) series solutions for general systems.
2. Improvements of the symmetry package: (a) special routines for polynomial differential equations as these represent the most important case for applications, (b) more efficient prolongation of vector fields, and (c) further analysis of the results like determination of the structure constants of the symmetry algebra, computation of differential invariants etc.
3. Implementation of a completion package on the basis of the formal theory of differential equations [12, 15].
4. As a special application more support for the analysis of mechanical systems: (a) generation of Hamiltonian equations of motion, (b) support for systems with constraints, (c) automatic generation of the equations of motion in a form suitable for numerical integration.

Acknowledgements

Eduard Niess (Universität Karlsruhe) ported large parts of our earlier AXIOM version of a programming environment for differential equations [14, 16] to *MuPAD*. Marcus Hausdorf (Universität Karlsruhe) wrote the Lie symmetry package; Jay Belanger (Northeast Missouri State University) implemented the method of characteristics. Both provided examples for this article and helped to improve our programming environment by busily reporting bugs.

This article was written using the Emacs modes MuPAD and EMuPAD interfacing *MuPAD* and LaTeX within the Emacs editor. These very useful modes have been developed by Jay Belanger.

This work was partially supported by Deutsche Forschungsgemeinschaft.

References

[1] M.O. Ahmed and R.M. Corless. The method of modified equations in Maple. Technical report, Dept. of Applied Mathematics, University of Western Ontario, London (Canada), 1997.

[2] R.L. Anderson and N.H. Ibragimov. *Lie-Bäcklund Transformations in Applications*. SIAM, Philadelphia, 1979.

[3] G.W. Bluman and J.D. Cole. The general similarity solution of the heat equation. *J. Math. Mech.*, 18:1025–1042, 1969.

[4] G.W. Bluman and S. Kumei. *Symmetries and Differential Equations*. Applied Mathematical Sciences 81. Springer-Verlag, New York, 1989.

[5] K. Drescher. Axioms, categories and domains. Automath Technical Report No. 1, Universität Paderborn, 1996.

[6] B. Fuchssteiner, S. Ivanov, and W. Wiwianka. Algorithmic determination of infinite-dimensional symmetry groups for integrable systems in 1+1 dimensions. *Math. Comp. Model.*, 25:91–100, 1997.

[7] D.F. Griffiths and J.M. Sanz-Serna. On the scope of the method of modified equations. *SIAM J. Sci. Stat. Comp.*, 7:994–1008, 1986.

[8] W. Hereman. Review of symbolic software for the computation of Lie symmetries of differential equations. *Euromath Bull.*, 2:45–82, 1994.

[9] W. Hereman. Symbolic software for Lie symmetry analysis. In N.H. Ibragimov, editor, *CRC Handbook of Lie Group Analysis of Differential Equations, Volume 3: New Trends in Theoretical Development and Computational Methods*, chapter 13. CRC Press, Boca Raton, Florida, 1995.

[10] P.J. Olver. *Applications of Lie Groups to Differential Equations*. Graduate Texts in Mathematics 107. Springer-Verlag, New York, 1986.

[11] P.J. Olver and P. Rosenau. Group-invariant solutions of differential equations. *SIAM J. Appl. Math.*, 47:263–278, 1987.

[12] J.F. Pommaret. *Systems of Partial Differential Equations and Lie Pseudogroups*. Gordon & Breach, London, 1978.

[13] E. Pucci and G. Saccomandi. On the weak symmetry groups of partial differential equations. *J. Math. Ana. Appl.*, 163:588–598, 1992.

[14] J. Schü, W.M. Seiler, and J. Calmet. Algorithmic methods for Lie pseudogroups. In N. Ibragimov, M. Torrisi, and A. Valenti, editors, *Proc. Modern Group Analysis: Advanced Analytical and Computational Methods in Mathematical Physics*, pages 337–344. Kluwer, Dordrecht, 1993.

[15] W.M. Seiler. *Analysis and Application of the Formal Theory of Partial Differential Equations*. PhD thesis, School of Physics and Materials, Lancaster University, 1994.

[16] W.M. Seiler. Applying AXIOM to partial differential equations. Internal Report 95–17, Universität Karlsruhe, Fakultät für Informatik, 1995.

[17] W.M. Seiler. Involution and symmetry reductions. *Math. Comp. Model.*, 25:63–73, 1997.

[18] H. Stephani. *Differential Equations: Their Solution Using Symmetries*. Cambridge University Press, Cambridge, 1989.