JETS

A Software for Differential Calculus on Jet Spaces and Diffieties

Version 4.9 (December 2003) for Maple V Release 4

Introductory Course

by M. Marvan, Opava

SUMMARY

Jets is a set of procedures to facilitate solution of differential equations in total derivatives on diffieties. Otherwise said, Jets is a tool to compute symmetries, conservation laws, zero-curvature representations, recursion operators, any many other invariants of systems of partial differential equations. This text brings a brief description and some examples.

1 Installing and starting Jets

The distribution consists of two files: `Jets.s` and `Jets.t`. To install Jets proceed as follows:

1. Start a fresh Maple session.
2. Read in the file `Jets.s`, which contains the source code of all procedures. There should be the following output on the screen:

   ```
                                  JETS 4.9

                  Differential calculus on jet spaces and diffieties

                                  for Maple V Release 4

   as of 18 December 2003

   Blimit = 25000   ressize = 500   putsize = 40   maxsize = 20
   ```

   If only the first line, JETS 4.9, is printed, then Maple for your platform uses different end-of-line characters. Please, open `Jets.s` in an editor and fix the problem.

   If error messages are produced, then your copy of the file is corrupted and should be replaced.
3. Read in the file `Jets.t`, which contains typical examples of computations and tests them against correct results. Running `Jets.t` should produce no output other than *O.K.* and a time stamp, otherwise Jets is not compatible with your version of Maple, or `Jets.t` is outdated.

   The time stamp produced by the `Jets.t` test gives an estimate how fast will Jets run on your hardware. Few seconds is perfect for average computations.

File `Jets.m` then can be found in the Maple directory. Next time Jets can be started simply by issuing the command

```
> read ‘Jets.m‘;
```

2 History

Jets 4.9 is a successor of Jet 3.9, the last widely available version of Jet. Backward compatibility is not guaranteed. Omitted are features that became nearly obsolete due to changes in the philosophy of Jets. The new features of Jets 4.9 are oriented towards automated solution of equations in total derivatives. Among them are tools to study coverings and Bäcklund transformations.

Jets was originally inspired by DeLiA of Bernstein and Bocharov [1], with universality being one of the design objectives. Based on Jets, another PDE's-oriented software can be developed.

Jet changed its name to Jets to be distinguished from the completely different Maple package of loosely similar functionality independently developed by Meshkov [5]. Hereman's paper [2] reviews other software oriented to symmetry analysis of PDE's available in early nineties.

Unfortunate changes in the Maple syntax rendered Jets unusable with versions newer than Maple V.4. The next version of Jets will reflect the new syntax.

3 Variables

For unexplained notions from the geometry of partial differential equations (jets, diffieties, etc.) see [8]. For unexplained Maple commands consult the online help to Maple.

Most of the procedures will not work until *base* and *fibre variables* (coordinates on the underlying fibred manifold) are specified by using the command `coordinates`. Contrary to version 3.9, there are no default settings.

To introduce two base variables, x, t and a single fibre variable u, enter

```
> coordinates([t,x], [u], 5);

I, u_5t, u_4t_x, u_tttxx, u_ttxxx, u_t_4x, u_5x, u_4t, u_tttx, u_ttxx,
    u_txxx, u_4x, u_ttt, u_ttx, u_txx, u_xxx, u_tt, u_tx, u_xx, u_t, u_x
```

As a side effect, this command creates aliases for *jet variables* (coordinates on the jet space) up to the order given by the last argument (five in this example). See the Maple help for explanation of what is an alias.

When in doubts, one can judge as follows: To have, say, the eighth-order jet variable v_{8y} at one's disposal, v must be a fibre variable, y must be a base variable, and the last argument must be at least eight; the declaration then may be, e.g., `coordinates([x,y],[u,v],10)`.

Creation of aliases is only possible when names given to base variables are strings, i.e., not indexed. Indexed fibre variables are allowed (try and see what happens). The creation of aliases is suppressed when the third argument is omitted.

There is no upper bound on the number of base and fibre variables. Every new declaration overrides the former.

Internally, jet variables are represented as unevaluated calls to the procedure `jet`. For example,

```
> jet(u,t);
                                    u_t
```

The output is the name `u_t` that has been aliased to the expression `jet(u,t)`. For higher order jets one has likewise

```
> jet(u,x^3*t);
                                   u_txxx

> jet(u,x^4*t);
                                   u_t_4x
```

The rule is that more than triple repetition of a base variable is indicated by a numeric coefficient and separated by underscores. Needless to say, one should normally input jet variables as the aliases, e.g., `u_t`, `u_txxx`, `u_t_4x`.

Jet variables for which no aliases were created appear in their internal representation. This has no impact on the validity of results. To create missing aliases, reinsert the `coordinates` command with the third argument sufficiently high.

Be careful to input your variables in exactly the same form as they appear in the output of the `coordinates` command. E.g., if `u_tx` is aliased, then `u_xt` is *not*. (The name `u_xt` then represents a function of unassigned dependence.)

In the above inputs, expressions like `x^3*t` are *counts*. Counts are traditional for the differential calculus (cf. $\partial^4 u/\partial x^3 \partial t$), so there should be no confusion. Maple handles them by the fast multiplication routines of its kernel. Formally, a count is an element of the free commutative semigroup over the set of variables.

4 Total and partial derivatives

Total derivatives are named `TD` in Jets. The procedure `TD` accepts the same counts as `jet` does (see above) but is not prettyprinted. E.g.,

```
> TD(f,x); TD(f,x^3*t);
```

$$\mathtt{TD(f,\ x)}$$

$$\mathtt{TD(f,\ x^3\ t)}$$

are total derivatives $D_x f$ and $D_{txxx} f$ of a function f. Any name different from names of base and fibre variables (and names special to Jets and Maple) is regarded as a name of a differentiable function.

Unlike `jet`, total derivatives `TD` are true differentiations and accept any algebraic expression as the first argument. They are linear, and satisfy the Leibniz rule as well as the chain rule:

```
> TD(f + g, x);
TD(f*g, x);
TD(sin(f), x);
```

$$\mathtt{TD(f,\ x) + TD(g,\ x)}$$

$$\mathtt{TD(f,\ x)\ g + f\ TD(g,\ x)}$$

$$\mathtt{cos(f)\ TD(f,\ x)}$$

Jets does not automatically evaluate total derivatives of functions since typically they are huge expressions carrying rather tiny portion of useful information. This is especially true for higher order total derivatives.

However, when appropriate, the evaluation of total derivatives can be forced:

```
> dependence(f(t,x,u,u_x));
evalTD(TD(f,x));
```

$$\mathtt{f = \{x,\ u\_x,\ t,\ u\}}$$

$$\mathtt{\left(\frac{d}{dx} f\right) + \left(\frac{d}{du} f\right) u\_x + \left(\frac{d}{du\_x} f\right) u\_xx}$$

The first command declares f as a function depending on t, x, u, u_x only. Without such a declaration, `evalTD` would have no effect. To declare a constant c, one can say either `dependence(c())` or simply `parameter(c)`.

The output of the second command involves *partial derivatives*. Partial derivatives are input as follows:

```
> pd(g,x*u_x^2);
```

$$\frac{d}{du_x\ dx^2}\,g$$

Observe that partial derivatives admit counts composed of jet variables.

Partial derivatives are our main tool to investigate expressions in total derivatives. Actually, partial derivatives of total derivatives can be evaluated without evaluating the latter. Typical examples are (f is assumed to depend on t, x, u, u_x):

```
> pd(TD(f,x),u_xx);
```

$$\frac{d}{du_x}\,f$$

The output may contain another total derivative:

```
> pd(TD(f,x),u_x);
```

$$TD\left(\frac{d}{du_x}\,f,\ x\right) + \left|\frac{d}{du}\,f\right|$$

(cf. the evaluated $D_x f$ above).

```
> pd(TD(f,x^2),u_xx);
```

$$2\,TD\left(\frac{d}{du_x}\,f,\ x\right) + \left|\frac{d}{du}\,f\right|$$

5 Conversions

It is often desirable to make a substitution for a fibre variable. For instance, let us verify that $a = 2k\,\mathrm{sech}^2(\sqrt{k}(x+4kt))$ is a solution of the KdV equation $u_t = u_{xxx}+6uu_x$ (the famous soliton solution), k being an arbitrary constant. One cannot say `subs(u = a, -u_t + u_xxx + 6*u*u_x)`, since the procedure `jet` requires its first argument to be a fibre variable, and therefore expressions like `jet(a,x)` do not make sense. Instead one should apply the procedure `convert` with `TD` as the second argument:

```
a := 2*k*(sech((k)^(1/2)*(x + 4*k*t)))^2;
dependence (k()):
T := convert(-u_t + u_xxx + 6*u*u_x, TD, u = a):
simplify(T);
```

$$0$$

The procedure effectively replaces `u` with `a`, `jet(u,x)` with `TD(a,x)`, etc. A much more powerful comand `transform` is described in one of the following sections.

With the second argument `diff` instead of `TD`, the same procedure `convert` replaces Jets' partial derivative `pd` with the standard Maple derivative `diff`:

```
> dependence(a(x,u,u_x,u_xx));

                        a = {u_xx, u_x, u, x}

> convert(a*pd(a,u_x^2*u_xx), diff, u_x = u1, u_xx = u2);
                              /    3                    \
                              |   d                     |
                 a(u1, u2, u, x) |-------- a(u1, u2, u, x)|
                              |       2                 |
                              \du2 du1                  /
```

Here the additional arguments indicate a substitution to be applied to jet variables (recall that jet variables are unevaluated function calls while `diff` requires the indeterminates to be names). Expressions obtained this way are ready for use in standard Maple procedures, e.g., `dsolve` and `pdesolve`.

The last type of conversion we consider in this section is performed automatically any time an expression is prettyprinted. Namely, the standard Maple procedure `print` has been instructed to convert `pd` to `Diff`, which is an inert (unevaluated) differentiation in Maple. This explains why Jets' partial derivatives are prettyprinted.

But `Diff` has also been redefined. Recall that if the display option is set to "typeset notation," then one can copy and paste an output into the input area to get a syntactically correct input essentially of the same meaning (although sometimes in the inert form). With `Diff` redefined to `pd`, the copy-and-paste feature of Maple works with `pd` as well.

```
> Diff(a,u_x,u_x,u_xx) - pd(a, u_x^2*u_xx);

                                    0
```

6 Diffieties

A computation dealing with, say, the KdV equation should start with the declaration

```
> equation (u_t = u_xxx + 6*u*u_x);
```

This command accepts any number of arguments separated by commas. Every new `equation` overrides the former. It may be necessary to use quotes to prevent evaluation on the left hand side:

```
> equation ('u_t' = u_xxx + 6*u*u_x);
```

It is also possible to remove any previous equation completely:

```
> equation ();
```

Every equation must be resolved with respect to a *leading* derivative q_X. This means that neither q_X nor either of its derivatives q_{XY} occurs on the right-hand side. For example, u_t and u_{xxx} are leading derivatives of the KdV equation $u_t = u_{xxx} + 6uu_x$, while u_x is not, since u_{xxx} is forbidden on the right hand side of $u_x = (u_t - u_{xxx})/6u$.

After an equation is introduced, certain jet variables obtain values:

```
> u_t;
                                u_xxx + 6 u u_x
> u_tx;
                                      2
                          u_4x + 6 u_x  + 6 u u_xx
> u_tt;
                 2                                          2       2
   u_6x + 18 u_xx  + 30 u_x u_xxx + 12 u u_4x + 72 u u_x  + 36 u  u_xx
```

Actually, these are results returned by the procedure `jet` (recall that the above inputs are equivalent to invoking `jet(u,t)`, `jet(u,t*x)`,`jet(u,t^2)`, respectively).

By giving more than one argument, systems of PDE are introduced. In that case, for `jet` to give correct (unique) results, the system must be *passive*, which roughly speaking means that there must be no non-trivial integrability conditions resulting from the system and its differential consequences (see. [6]).

The complexity of the expressions grows dramatically with the jet order. This is why computed values are (transparently to the user) stored in a table and then retrieved rather than recomputed every time when needed. The table is emptied automatically when a new equation is introduced.

However, if the right hand side of an equation depends on a function f and the function f is changed, then it is the user who must take care of refreshing the table.

For example:

```
> dependence(f(x,u));
                                f = {x, u}

> equation('u_t' = u_xxx + f*u_x);

                           u_t = u_xxx + f u_x
> u_tx;
                            /d   \   /d   \    2
                  u_4x + u_x |-- f| + |-- f| u_x  + f u_xx
                            \dx  /   \du  /
```

Now we make an assignment to f:

```
> f := x*u;
                                f := x u
```

Then the jet variables will keep their previous values

```
> u_tx;
                            /d   \   /d   \    2
                  u_4x + u_x |-- f| + |-- f| u_x  + f u_xx
                            \dx  /   \du  /
```

unless explicit evaluation is invoked

```
> eval(u_tx);
                                          2
                  u_4x + u_x u + x u_x  + x u u_xx
```

or the command `refresh` is invoked to reevaluate the table.

```
> refresh();
> u_tx;
                                          2
                  u_4x + u_x u + x u_x  + x u u_xx
```

7 Linear equations in total derivatives. Symmetries of KdV

In this section we introduce our first concrete example: computation of fifth-order symmetries of the KdV equation. As usual, the KdV equation is introduced as

```
> equation('u_t' = u_xxx + 6*u*u_x);

                              u_t = u_xxx + 6 u u_x
```

Symmetries (more precisely, their generating functions U) are solutions of the *determining equation* $S = 0$ where S is the universal linearization operator:

```
> S := symmetries(u = U);
                                                3
            S := TD(U, t) - 6 U u_x - TD(U, x ) - 6 TD(U, x) u
```

(U corresponds to the fibre variable u, as indicated by the argument $u = U$.) The PDE is always given by the last `equation` command.

Likewise, the determining equation for generating functions of conservation laws is $L = 0$ where

```
L := laws(1 = V);
                                          3
              L := -TD(V, t) + TD(V, x ) + 6 u TD(V, x)
```

(V denotes the unknown corresponding to the first equation, as indicated by the relation $1 = V$.)

In general, determining equations are equations in total derivatives. Those for symmetries and conservation laws are linear, but Jets allows one to solve nonlinear equations as well. Nonlinear are, e.g., determining systems for coverings and zero-curvature representations.

Jets' mission is solving the equations in total derivatives, linear or nonlinear. The solution process repeats two steps:

(1) deriving simple differential consequences;

(2) resolving and back-substituting simplest of them;

until the input expressions become zero. Essentially, both steps can be automated.

Turning back to our example, we need to solve the equation $S = 0$ for U. As a smooth function, U can depend only on a finite number of variables, i.e., there must be an upper bound on the jet order of U. Assume that the bound is five. A fifth-order generating function is one that depends on x, t, u, $u_x, \ldots, u_{5x}$. Accordingly, our assumption is made known to the computer as follows:

```
> dependence(U(x,t,u,u_x,u_xx,u_xxx,u_4x,u_5x));

            U = {t, x, u, u_x, u_xx, u_xxx, u_4x, u_5x}
```

To run in automatic regime, Jets also needs to know what are the unknowns:

```
> unknowns(U);
                                    U
```

Now launch the computation:

```
> run(S);

<0>   Put:
                                  2
                                 d
                               ------ U = 0
                                    2
                               du_5x

<1>   Put:
                                   2
                                  d
                             ----------- U = 0
                             du_5x du_4x

<1>   Put:
                                    2
                                   d
                             ------------ U = 0
                             du_5x du_xxx
```

etc. The report lines have the syntax

<*elapsed time*> `Put:`

The centered formulas following every report line are assignments to partial derivatives, to be explained below.

The whole computation takes 49 such steps and ends with

```
<77>   Put:
d                    /d   \           /d   \              2 / d    \
-- U = 6 U u_x + 3/2 |-- U| u_xxx + 3 |-- U| u_x u - 6 u_x  |---- U|
dt                   \du  /           \du  /                \du_x  /

                   /  d    \      /  d    \
     - 6 u_x u_5x |----- U| - 30 |----- U| u u_x u_xxx
                  \du_5x  /      \du_5x  /
```

```
           2 /  d    \                /  d     \
- 15 u_xxx  |----- U| - 15 u_x u_xxx |------ U|
            \du_5x  /                \du_xxx  /

     /  d     \      2
- 18 |------ U| u u_x
     \du_xxx  /
```

```
<79>   Success!
```

To retrieve the results, on should proceed as follows.

```
> dependence();
```

```
                {U(t, u_5x, u_x, u_xx, u, u_xxx, x)}
```

The result shows what jet variables do the symmetry generators depend on (in our case, all but u_{4x}). Then invoke the command `clear(pds)`

```
> Z := clear(pds);
```

```
         2         2            2              2
        d         d            d              d
Z := {------ U, ------- U, ------------ U, ----------- U,
           2          2    du_5x du_xxx    du_xxx du_x
      du_5x     du_xxx

     /   2     \                 /    2      \
     |  d      |     /  d     \  |   d       |      /  d    \
     |------- U| - 6 |------ U|, |--------- U| - 10 |----- U|,
     \du_x du  /     \du_xxx  /  \du_xxx du  /      \du_5x  /
```

$$\vdots$$

Here $Z = \{Z_i\}$ is a set of expressions, which we have truncated because of its rather big size. The meaning is that the unknown U must satisfy differential equations $Z_i = 0$. The command also removes all assignments to partial derivatives (otherwise Z would be a set of zeroes). This is why `clear(pds)` should not be called repeatedly.

The equations $Z_i = 0$ must be solved by hand, keeping in mind that the integration "constants" depend on the "remaining" variables. In our example, the ODE $\partial^2 U/\partial u_{5x}^2 = 0$ is immediately solved as $U = U_1 u_{5x} + U_0$, where U_i are arbitrary functions which do not depend on u_{4x}, because U did not, and on u_{5x}, as u_{5x} was the independent variable of the ODE in question.

```
> U := U1*u_5x + U0;
> dependence(U1(t,x,u,u_x,u_xx,u_xxx), U0(t,x,u,u_x,u_xx,u_xxx));
```

```
                          U := U1 u_5x + U0

      U1 = {t, x, u, u_x, u_xx, u_xxx}, U0 = {t, x, u, u_x, u_xx, u_xxx}
```

The next step may be

```
> map(expand, Z);
```

which shows all the equations after the assignment $U = U_1 u_{5x} + U_0$; or better

```
> map(expand@pd, Z, u_5x);
```

which gives the (more substantial) equations that arise as coefficients at u_{5x}; or even better

```
> unknowns(U1,U0):
> run(Z);
```

$$\vdots$$

```
> dependence();
                    U1 = {}, U0 = {t, u, u_x, u_xx, u_xxx, x}
> Z := clear(pds);
        2          2            2
       d          d            d
Z := {----- U0, ------- U0, ----------- U0,
          2           2     du_xxx du_x
      du_x       du_xxx

    /   2      \                  / 2    \
    |  d       |     /  d      \  |d     |
    |------- U0| - 6 |------ U0|, |--- U0| - 60 U1 u_x,
    \du_x du   /     \du_xxx   /  |  2   |
                                  \du    /

    /d    \                          2 /  d      \           /d    \
    |-- U0| + 5 u_x u_xxx U1 + 3 u_x  |------ U0| - 1/2 u_x |-- U0|,
    \dx   /                           \du_xxx   /           \du   /

    /    2       \
    |   d        |          /  d     \
    |--------- U0| - 10 U1, |----- U0| - 20 U1 u_x,
    \du_xxx du   /          \du_xx   /
```

$$\vdots$$

Prior to the new `run`, one must not forget to declare the new unknowns U_1, U_0 (in this order, since U_0 depends on a larger set of variables than U_1 does).

We highly recommend that the reader performs the whole computation. Omitting details, we state the final answer that 5th order symmetries of KdV are

$$\begin{aligned} U = {} & \tfrac{1}{6}a_2 + \tfrac{1}{9}a_4 u \\ & + (a_1 + a_2 t + \tfrac{1}{18}a_4 x + (a_3 + a_4 t)u + a_5 u^2)u_x \\ & + \tfrac{2}{3}a_5 u_x u_{xx} + (\tfrac{1}{6}a_3 + \tfrac{1}{6}a_4 t + \tfrac{1}{3}a_5 u)u_{xxx} + \tfrac{1}{30}a_5 u_{5x}, \end{aligned} \tag{1}$$

where a_1, a_2, a_3, a_4, a_5 are constants. This means that the Lie algebra of 5th order symmetries is five-dimensional, with basis formed by

$$\begin{aligned} U_1 &= u_x, \\ U_2 &= \tfrac{1}{6} + tu_x, \\ U_3 &= uu_x + \tfrac{1}{6}u_{xxx}, \\ U_4 &= \tfrac{1}{9}u + (\tfrac{1}{18}x + tu)u_x + \tfrac{1}{6}tu_{xxx}, \\ U_5 &= u^2u_x + \tfrac{2}{3}u_xu_{xx} + \tfrac{1}{3}uu_{xxx} + \tfrac{1}{30}u_{5x}. \end{aligned}$$

The Jacobi bracket (the commutator of symmetries in terms of their generating functions) is computed by the procedure `Jacobi`. Its two arguments are the generating functions formatted as lists of expressions. E.g.,

```
> Jacobi([u_x], [1/6 + t*u_x]);
                                    [0]

> Jacobi([u_x], [u*u_x + 1/6*u_xxx]);

                                    [0]

> map(expand, Jacobi([1/6 + t*u_x], [u*u_x + 1/6*u_xxx]));

                                 [1/6 u_x]
```

In case of systems of equations, the ordering of the lists is determined by the order in which the corresponding fibre variables appeared in the `coordinates` declaration.

Obviously, `run` can work reliably only if it recognizes zero. An expression f is considered zero if and only if the output of `simpl(f)` is `0`, where the default definition of `simpl(f)` is `normal(f)`. Under this definition, `simpl` reliably recognizes zero in the domain of rational functions, which was the case with the KdV equation. In other domains, `simpl` should be redefined appropriately.

8 Deriving differential consequences

Here we explain how `run` performs the step (1): deriving the simplest differential consequences. We use the KdV example of the preceding section.

An expression S in total derivatives is better kept unevaluated as long as possible (if in doubts, try `evalTD(S)`). So, instead of evaluating S, we use partial differentiation: if S is zero, then all derivatives of S are zero as well. It is then important to know what variables does S depend on.

```
> coordinates([t,x],[u],8);
vars(S);

      {t, u_6x, x, u, u_7x, u_8x, u_xxx, u_xx, u_4x, u_5x, u_x}
```

(the first command creates the aliases u_{6x}, u_{7x}, and u_{8x}). Among these variables, u_{8x} is of highest order. When differentiating S with respect to u_{8x}, one may expect that the result will be a rather simple expression. And indeed,

```
> pd(S,u_8x);
                                    0
```

which shows that S is actually independent of u_{8x} (all occurrences of u_{8x} would cancel out under `evalTD` and proper simplification). The next variable in row is u_{7x}:

```
> T := pd(S,u_7x);
                                      d
                          T := - 3 TD(----- U, x)
                                    du_5x
```

This result can be further processed in the same way. One easily checks that T depends on u_{6x} at most, and then the derivative

```
> pd(T,u_6x);
                                  /   2    \
                                  |  d     |
                              - 3 |------ U|
                                  |     2  |
                                  \du_5x   /
```

gives the first meaningful piece of knowledge: $\partial^2 U/\partial u_{5x}^2 = 0$ (i.e., U linearly depends on u_{5x}). This explains the first intermediate result `run` reported about (see the preceding section):

```
> run(S);

1.   <1>   Put:
                                    2
                                   d
                                 ------ U = 0
                                      2
                                 du_5x
```

In fact, `run` makes an assignment $\partial^2 U/\partial u_{5x}^2 := 0$. Assignments to partial derivatives will be discussed in one of the following sections.

The algorithm to automate derivation of differential consequences is implemented as the procedure `derive` (which is one of the subroutines called by `run`):

```
derive(S);
                                  /   2    \
                                  |  d     |
                              - 3 |------ U|
                                  |     2  |
                                  \du_5x   /
```

The procedure `derive` recursively computes derivatives of the input expression and returns nonzero expressions of minimal size. More precisely, `derive` stops if the subsequent derivative either vanishes or fails to decrease the size. By default, size(f) is the number of unknowns and their derivatives f depends on plus length(f) $\times 10^{-9}$. Hence, minimal expressions are those with the minimal number of unknowns, with ties broken by length.

Deriving will work (produce a result in finite time) if `size(`f`)` returns a nonnegative integer for any input expression f.

In order to keep the space and time complexity low, every step in the derivation process uses only the highest-order jet derivatives the input expression depends on. Therefore, derivation is in fact restricted to maximal elements of the set of jet variables ordered as follows:

$$u_X < v_Y \quad \Leftrightarrow \quad u = v \text{ and } X \text{ divides } Y$$

(divisibility of counts in the obvious sense). After maximal elements are exhausted, the procedure continues with maximal elements among the remaining variables, etc.

Still there may exist maximal jet variables e that produce a large expression $\partial S/\partial e$. Nonlocal variables of coverings are typical examples. Fortunately, these variables can be safely excluded from derivation since the derivative $\partial S/\partial e$ would be left unused anyway. To exclude certain fibre variables from derivation (along with all jet variables they generate), use

`noderive(`*list of fibre variables*`);`

9 Assignments to partial derivatives

Equations produced by `derive` do not always allow for explicit solution. However, very often these expressions can be simply resolved with respect to one of the leading derivatives and then used as substitutions. Example: instead of trying to solve the PDE $\partial^2 U/\partial u_x^2 + u_x\,\partial U/\partial u = 0$ we may routinely introduce the substitution $\partial^2 U/\partial u_x^2 = -u_x\,\partial U/\partial u$.

Jets provide a procedure `put` to make assignments to partial derivatives that propagate to all differential consequences. For instance:

```
> put('pd(U,u_x^2)' = -u_x*pd(U,u));
> pd(U,u_x^2);
                                  /d   \
                              -u_x |-- U|
                                  \du  /

> pd(U,u_x^3);
                                    /   2     \
                    /d   \          |  d      |
                   -|-- U| - u_x |------- U|
                    \du  /         \du du_x  /
```

Another example: Resolving the famous Cauchy–Riemann conditions

$$\partial U/\partial u = \partial V/\partial v, \qquad \partial V/\partial u = -\partial U/\partial v,$$

via the pair of substitutions

```
> put('pd(U,u)' = pd(V,v), 'pd(V,u)' = -pd(U,v));
```

produces the Laplace equation $\partial^2 U/\partial u^2 + \partial^2 U/\partial v^2 = 0$ effortlessly:

```
> pd(U,u^2);
```

$$\left(\frac{d}{du}U\right)^2$$

The behaviour of `put` is quite similar to that of `equation`. Again, substitutions are stored in a table. One exception is that a substitution of the form `pd(U,u) = 0` will result in the mere change of the dependence record of U in case it exists. Another exception is that a substitution of the form `U = f` will result in the direct assignment `U := f`.

There are two obvious sources of problems if `put` is used arbitrarily. Firstly, substitutions can form vicious circles, generating stack overflows. The standard way to avoid them is explained in the next chapter.

Secondly, substitutions can contradict one another, generating chaotic and session-dependent results. For instance, the above Cauchy–Riemann conditions can be alternatively resolved as

```
> put('pd(V,v)' = pd(U,u), 'pd(V,u)' = -pd(U,v));
```

Then the input

```
> pd(V,u*v);
```

will produce either

$$\frac{d^2}{du^2}U$$

or

$$\left(\frac{d}{dv}U\right)^2$$

depending on whether it resulted from the first or the second substitution. Such questions are addressed by the formal integrability theory of systems of PDE's. For simplicity, Jets performs only the standard cross-derivative checks:

```
> pd(pd(V,v),u) - pd(pd(V,u),v);
```

$$\left(\frac{d}{du}U\right)\left(\frac{d}{dv}U\right) + \left(\frac{d}{dv}U\right)\left(\frac{d}{du}U\right)$$

(which is another way of obtaining the Laplace equation). To ensure uniqueness of substitutions, all cross-derivatives must be balanced, i.e., non-zero expressions resulting from cross-derivatives must be resolved and the results substituted back again. In this way, substitutions force other substitutions, e.g.,

```
> put('pd(U,u^2)' = -pd(U,v^2)):
```

There is a procedure `cc()` which computes the compatibility conditions:

```
> cc();
```

```
{}
```

This output shows that all cross-derivatives have been already balanced.

A nonempty set of compatibility conditions returned by `cc` need not be resolved immediately. May be postponed, e.g., to see whether the subsequent `derive` does not produce a condition simpler to resolve.

Once all cross-derivatives are balanced, it is possible to reduce the table by eliminating all derivatives that are derivatives of the others:

```
reduce();
```

```
                                        2       / 2   \
d      d     d       /d   \  d         |d    |
-- V = -- U, -- V = -|-- U|, --- U = -|--- U|
dv     du    du      \dv  /    2      |  2  |
                                 du      \dv   /
```

10 Avoiding vicious circles

Suppose we want to resolve the equation $\partial U/\partial u = \partial^2 U/\partial u^2$ with respect to one of the partial derivatives. Obviously, the substitution $\partial U/\partial u \to \partial^2 U/\partial u^2$ will induce a vicious circle of substitutions

$$\partial U/\partial u \to \partial^2 U/\partial u^2 \to \partial^3 U/\partial u^3 \to \partial^4 U/\partial u^4 \to \cdots$$

Not so the inverse substitution $\partial^2 U/\partial u^2 \to \partial U/\partial u$, which therefore presents the only acceptable way to resolve the equation.

Standard methods to avoid vicious circles use special orderings $\prec$ of partial derivatives. By a partial derivative we mean an expression `pd(`U, X`)` where U is a function and X is a count over some set of variables. Functions themselves are regarded as partial derivatives with $X = 1$. The orderings should satisfy

$$P \prec \frac{\partial P}{\partial s} \tag{2}$$

and

$$Q \prec P \quad \Rightarrow \quad \frac{\partial Q}{\partial s} \prec \frac{\partial P}{\partial s} \tag{3}$$

for all partial derivatives P, Q and every variable s. Given such an ordering, one imposes the rule that

$$\begin{array}{l}\text{in every substitution } P \to \sigma\text{, the expression } \sigma \\ \text{involves only derivatives } Q \text{ satisfying } Q \prec P.\end{array} \tag{4}$$

Then it is easy to prove that *if substitutions satisfy the rule* (4), *then every composition of substitutions and every differential consequence of a substitution satisfy the same rule* (4). [Prove as an exercise. Hint: A differential consequence of the rule $P \to \sigma$ is the rule

$$\frac{\partial P}{\partial s} \to \sum_Q \frac{\partial \sigma}{\partial Q} \cdot \frac{\partial Q}{\partial s},$$

where Q runs over all partial derivatives occurring in σ.]

Substitutions satisfying rule (4) cannot produce vicious circles. Every string of substitutions will eventually arrive to a "minimal state" where no further substitution can be applied.

Orderings that satisfy conditions (2), (3) can be introduced as follows. First we need a workable ordering $\prec_{\mathrm{var}}$ of variables.

1. $x \prec_{\mathrm{var}} u$ for every base variable x and every fibre variable u;
2. $u \prec_{\mathrm{var}} u_X$ for every fibre variable u and every count X;
3. given an ordering $\prec_{\mathrm{b}}$ of base variables and an ordering $\prec_{\mathrm{f}}$ of fibre variables, we have $u_X \prec_{\mathrm{var}} v_Y$ if

   `(degree)` the degree of the count X is less than the degree of the count Y;

   with ties broken by

   `(function)` $u \prec_{\mathrm{f}} v$;

   with ties broken by

   `(count)` under the ordering $\prec_{\mathrm{b}}$, the greatest variable that occurs in Y/X has a positive exponent.

By default, the orderings $\prec_{\mathrm{b}}$ and $\prec_{\mathrm{f}}$ are those introduced by the `coordinates` command. Alternative orderings $\prec_{\mathrm{var}}$ are obtained by changing the order in which criteria `degree`, `function`, `count` are applied (obviously, after `count` there is no room for `degree`). E.g., the ordering governed by the succession `function`, `degree`, `count`, is introduced by

```
varordering(function,degree,count);
```

Having defined an ordering of variables, one obtains an ordering of partial derivatives when replacing $\prec_{\mathrm{f}}$ with any ordering $\prec_{\mathrm{fun}}$ of functions and $\prec_{\mathrm{b}}$ with any ordering $\prec_{\mathrm{var}}$ of variables.

In Jets, the ordering $\prec_{\mathrm{fun}}$ is determined by the particular order the functions are declared with `unknowns`. The ordering $\prec_{\mathrm{var}}$ is that introduced above. The command to change the succession of criteria is `Varordering`. The default is `Varordering(function,degree,reverse)`, where `reverse` means `count` with respect to the ordering reverse to the ordering of variables. The orderings just constructed are linear (every two elements are comparable).

In general, computations in automatic regime incline to expression swell. The predisposition to produce memory overflow is minimal if the `Varordering` declaration starts with `degree`.

However, in the presence of two or more unknowns, it is often desirable to obtain the resulting system Z in a "triangular" form. It is then possible to solve the equations on the first unknown first; then, knowing explicit expression for the first unknown, to solve the equations on the second unknown, etc. Triangular form is produced if the ordering options start with `function`. To select expressions from a set Z that involve only unknowns from a set A, call `unksselect(Z,A)`.

To obtain the triangular form while avoiding expression swell, two runs should be performed: first `run(S)` with respect to the ordering by `degree,function,reverse`, followed by `Z := clear(pds)`, and then `run(op(Z))` with respect to `function,degree,reverse`.

11 Classification problems

Another example, taken from [9, 8.1.5], is

```
> equation('u_t' = f*u_x + u_xx):
> dependence(f(u)):
```

Here the right-hand side of the equation depends on u via an unknown function f. In loc. cit. authors solve the following problem: how does the symmetry algebra depend on f. Cases 1--5 found in the book will be reconstructed below. The universal linearization is

```
> S := symmetries(u = U);
                                          /d   \              2
           S := TD(U, t) - TD(U, x) f - U |-- f| u_x - TD(U, x )
                                          \du  /
```

Assuming

```
> dependence(f(u), U(t,x,u,u_x,u_xx));
```

we soon get the following output:

```
<10>   linear resolving failed for   pd(U,x*u_x)

       /   2     \             / 2   \
/d   \ |  d      |    / d    \ |d    |       /d   \ /d   \
|-- f| |------- U| = -|---- U| |--- f| u_x + |-- U| |-- f|
\du  / \du_x dx  /    \du_x  / |  2  |       \du  / \du  /
                               \du   /

         / 2   \   // 2   \                    \
         |d    |   ||d    |        /d   \2     | /  d    \
     + U |--- f| - ||--- f| u_xx + |-- f|  u_x| |----- U|
         |  2  |   ||  2  |        \du  /      | \du_xx  /
         \du   /   \\du   /                    /

                              FAIL
```

This means that Jets is about to solve a linear equation on $\partial^2 U/\partial u\,\partial u_x$ with leading coefficient $\partial f/\partial u$, but lacks information that the leading coefficient is nonzero.

In such cases, Jets is expecting an instruction from the user. If there is more than one possible continuation, then it is recommended to fork the computation: to save the worksheet, make one more copy, and continue them separately. The current status of the computation can be stored in a file; see the command `store` below, and then input into another session.

1. When $\partial f/\partial u = 0$, we say `put('pd(f,u)' = 0)` and continue. The result will be that of Case 5 in the loc. cit.

When $\partial f/\partial u \neq 0$, we instruct Jets about this fact as follows:

```
> nonzero(pd(f,u));
```

```
                                  d
                                 {-- f}
                                  du
```

Continuing, we rather quickly get another condition:

```
<9>   linear resolving failed for   pd(U,u)
```

```
        / 2   \
 /d   \ |d    | /d   \
-|-- U| |--- f| |-- f| =
 \du  / |  2  | \du  /
        \du   /
```

```
    /    / 3   \              / 2   \2\              / 2   \2
    |    |d    | /d   \       |d    | | / d    \     |d    |
   -|u_x |--- f| |-- f| - u_x |--- f| | |---- U| - U |--- f|
    |    |  3  | \du  /       |  2  | | \du_x  /     |  2  |
    \    \du   /              \du   / /              \du   /
```

```
        / 3   \
        |d    | /d   \
    + U |--- f| |-- f| -
        |  3  | \du  /
        \du   /
```

```
   /        / 2   \              / 3   \        / 2   \2     \
   |/d   \2 |d    |       /d   \ |d    |        |d    |      |
   ||-- f|  |--- f| u_x + |-- f| |--- f| u_xx - |--- f|  u_xx|
   |\du  /  |  2  |       \du  / |  3  |        |  2  |      |
   \        \du   /              \du   /        \du   /      /
```

```
   /  d    \
   |----- U|
   \du_xx  /
```

```
FAIL
```

Now we choose between $\partial^2 f/\partial u^2$ zero and nonzero.

2. The first choice leads to Case 4 of loc. cit.

The alternative leads to:

```
<19>   linear resolving failed for   pd(U,x)

/ / 3   \            / 2   \2\ / 2   \
| |d    | /d   \     |d    | | |d    | /d   \ /d   \
|-|--- f| |-- f| + 2 |--- f| | |--- f| |-- f| |-- U| =
| |  3  | \du  /     |  2  | | |  2  | \du  / \dx  /
\ \du   /            \du   / / \du   /

     / / 3   \            / 2   \2\2
     | |d    | /d   \     |d    | |     2 / d    \
    -|-|--- f| |-- f| + 2 |--- f| |  u_x  |---- U|
     | |  3  | \du  /     |  2  | |       \du_x  /
     \ \du   /            \du   / /

       / / 3   \            / 2   \2\2
       | |d    | /d   \     |d    | |           /  d    \
     - |-|--- f| |-- f| + 2 |--- f| |  u_x u_xx |----- U|
       | |  3  | \du  /     |  2  | |           \du_xx  /
       \ \du   /            \du   / /

       / / 3   \            / 2   \2\2
       | |d    | /d   \     |d    | |
     + |-|--- f| |-- f| + 2 |--- f| |  U u_x
       | |  3  | \du  /     |  2  | |
       \ \du   /            \du   / /

                                FAIL
```

Now

$$\frac{\partial^3 f}{\partial u^3}\cdot\frac{\partial f}{\partial u} - 2\left(\frac{\partial^2 f}{\partial u^2}\right)^2 \tag{5}$$

is either zero or nonzero.

3. In the first case, $f := c_1 \ln(u + c_2) + c_3$, where c_1, c_2, c_3 are constants:

```
> f := c1*ln(u + c2) + c3;
> parameters(c1,c2,c3);
> refresh();
                        f := c1 ln(u + c2) + c3

                              c3, c1, c2
```

(`refresh` clears remember tables which could still contain unassigned f.) This is exactly Case 2 of loc. cit.

Alternatively, we can declare the expression (5) to be nonzero. After that we get one more fork very soon, namely:

```
<15>   linear resolving failed for   pd(U,u_x)

/ / 2   \2 / 3   \   / 2   \        / 4   \           / 3   \2\
| |d    |  |d    |   |d    | /d   \ |d    |     /d   \ |d    | |
|-|--- f|  |--- f| - |--- f| |-- f| |--- f| + 2 |-- f| |--- f| | u_x
| |  2  |  |  3  |   |  2  | \du  / |  4  |     \du  / |  3  | |
\ \du   /  \du   /   \du   /        \du   /           \du   / /

    / d    \
    |---- U| = -
    \du_x  /

    / / 2   \2 / 3   \   / 2   \        / 4   \           / 3   \2\
    | |d    |  |d    |   |d    | /d   \ |d    |     /d   \ |d    | |
    |-|--- f|  |--- f| - |--- f| |-- f| |--- f| + 2 |-- f| |--- f| |
    | |  2  |  |  3  |   |  2  | \du  / |  4  |     \du  / |  3  | |
    \ \du   /  \du   /   \du   /        \du   /           \du   / /

         /  d    \
    u_xx |----- U| +
         \du_xx  /

    / / 2   \2 / 3   \   / 2   \        / 4   \           / 3   \2\
    | |d    |  |d    |   |d    | /d   \ |d    |     /d   \ |d    | |
    |-|--- f|  |--- f| - |--- f| |-- f| |--- f| + 2 |-- f| |--- f| |
    | |  2  |  |  3  |   |  2  | \du  / |  4  |     \du  / |  3  | |
    \ \du   /  \du   /   \du   /        \du   /           \du   / /

    U

                                FAIL
```

Maple can solve the condition that the leading coefficient is zero after we rewrite the latter in the form

$$\left(\frac{f'f'''}{f''^2}\right)' = 0,$$

which is

$$f'f''' = af''^2,$$

where a is a constant.

4. For $a \neq 1$ we get Case 3.

5. For $a = 1$ we get Case 1.

Another type of failure may be due to nonlinearity of the equation to be solved, which occurs only if the input equation in total derivatives is nonlinear. It is recommended to avoid solving nonlinear equations since their solutions tend to be expressions difficult to simplify.

12 Fighting expression swell

Jets has high predisposition to expression swell when running in automatic regime. We already observed sensitivity to the `Varordering`, with ordering by `degree` giving more stable behaviour than the others.

The succession of unknowns is also significant. The rule to be followed is that unknowns should be listed in the order of increasing size of their dependence set. E.g.,

```
dependence(a(x), b(x,u,u_x)):
unknowns(a,b):
```

is preferable to `unknowns(b,a)`. The reason is that with `unknowns(a,b)` (the derivatives of) b will become expressed through (the derivatives of) a rather than the converse.

Four global variables control the data flow in automatic regime. The output of `derive` and `cc` typically contains a large number of expressions. The `ressize` is a number that determines the portion of these expressions to become the input of `resolve`. Namely, expressions are sorted by size and harvested one by one until the product of their sizes does not exceed the `ressize`. The default definition of `size` ensures that one-term expressions have (nearly) unit size and therefore all pass to resolve.

The results of `resolve` also involve expressions of variety of sizes. Expressions of size exceeding `maxsize` never pass to the output and never generate a 'resolving failure' report. Of `resolve`'s results only those of size not exceeding `putsize` are used in substitutions. It makes sense to set `maxsize` higher than `putsize`, because otherwise important 'resolving failure' reports could be missed.

Finally, `Blimit` is the threshold of 'bytes used' where `run` will call `reduce` during every loop.

However, very often the expression swell can be associated with the growing size of numerators, leading to various 'object too big' errors. In such cases, one should prefer `reverse` to `count` in `Varordering`. Transforming the system to another set of variables may be also helpful.

13 Transformation of variables

The procedure `transform` is suitable for transforming an expression to another set of variables. For example,

```
> normal(transform(x = u_x, y = u_y, u = x*u_x + y*u_y - u, u_x));

                                  x

> normal(transform(x = u_x, y = u_y, u = x*u_x + y*u_y - u, u_xx + u_yy));

                                 u_xx + u_yy
                               -----------------
                                              2
                               u_xx u_yy - u_xy
```

is the famous Legendre transformation. Needless to say, the `equation` should be empty.

14 About declarations

A procedure whose purpose is to change settings of global variables is called a declaration. If named in plural, it resets the previous declaration of the same type. If named in singular, it has a cummulative effect. Some declarations exist only in the plural form (e.g., `coordinates`).

Through a declaration, unassigned names acquire a meaning and become *registered*. For example, base and fibre variables are registered names once they are declared through the coordinates command.

Another possible meaning for a registered name is *parameter*. The declaration

```
parameter(c);
                                    c
```

has the same effect as `dependence(c())`, but allows c to be still usable as a variable. For instance,

```
> dependences(f(c,x));

                                f = {c, x}
> pd(f,c);
                                   d
                                   -- f
                                   dc
```

Meanings other than 'variable' are stored in a special table. The command `registered()` prints meanings of all registered names; `registered(`m`)` prints all names of the given meaning m.

```
> registered();
                                c = {parameter}

> registered(parameter);
                                    c
```

To remove a particular meaning from all names that have it, use `clear`. For instance

```
> clear(parameter);
                                    c
> pd(f,c);
Error, pd expects its 2nd argument, p, to be of type ar/count, but received c
```

15 Storing intermediate results

The command `store(`*filename*`)` has the effect that all assignments to unknowns and their derivatives, all dependence sets, and the set of all expressions declared nonzero, are written to the specified file. The format is so chosen that the file can be read into another Maple session.

```
> unknowns(a,b,c):
> a := b + c:
> put('pd(b,x)' = 1):
```

```
> nonzero(b,c):
> dependence(b(x,u)):
> store(terminal):
                              storing in terminal
assign({a = b+c})
;
dependence(b = {u, x})
;
put(
pd(b,x) = 1
);
nonzero(b,c)
;
```

Maple also offers an alternative solution, namely to store the snapshot of memory content in a file, which is, however, rather useless in case of memory overflow.

16 Computation of coverings

Jets can be useful when computating coverings, albeit not in automatic mode. As an example we consider the Burgers equation [8, Ch. 6, Section 2.1]. What follows is a commented full record of computation.

```
> coordinates([x,t], [u], 3):
```

The Burgers equation is

```
> equation('u_t' = u_xx + u*u_x);

                              u_t = u_xx + u u_x
```

For further reference, we denote by B the corresponding diffiety.

Recall that Phase I of computation of coverings uses abstract vector fields defined on the product $\tilde{B} = B \times W$, where W is an abstract manifold, finite- or infinite-dimensional.

Jets' objects are vector fields if they are of one of the following three types: a) names declared as such, i.e., names that have been written among arguments of the command `vectorfields`; b) total derivatives; c) partial derivatives; d) commutators of vector fields; e) linear combinations of the above.

The total derivative with respect to x is denoted by `TD[x]` (we mean the local one, i.e., one defined on the diffiety B). It is possible to apply a total derivative to a function f by saying either

```
> apply(TD[x],f);
                                    TD(f, x)
```

or simply

```
> TD[x](f);
                                    TD(f, x)
```

E.g., we have

```
> TD[x](x*t);
                                    t
```

Of course, no expanded form of the total derivatives is available.

Similarly, `pd[u]` means the vector field $\partial/\partial u$ and can be applied to a function in one of the following ways:

```
> apply(pd[u],f);
> pd[u](f);
                                   d
                                   -- f
                                   du

                                   d
                                   -- f
                                   du
```

The commutator is `comm`. As expected, D_x commutes with D_t:

```
> comm(TD[x], TD[t]);
                                    0
```

It is advisable to alias a shorter name to `comm`, say C:

```
> alias(C = comm);

     I, u_xxx, u_xxt, u_xtt, u_ttt, u_xx, u_xt, u_tt, u_x, u_t, C

> C(TD[x], TD[t]);
                                    0
```

Denoted `Ex`, `Et`, the nonlocal (extended) total derivatives $\tilde{D}_x$, $\tilde{D}_t$ on the covering are the local total derivatives plus the corresponding *tails* X, T:

```
> Ex := TD[x] + X;
> Et := TD[t] + T;
                            Ex := TD[x] + X

                            Et := TD[t] + T
```

Now we must tell Jets that X, T serve as tails, i.e., that they are abstract vector fields and, moreover, purely nonlocal ones. This is done by a single command:

```
> tail(X,T);
                                   X, T
```

Tails are distinguished by the property that their action on local variables is zero:

```
> apply(X,u_xx);
```

$$0$$

Let R be the commutator $[\tilde{D}_x, \tilde{D}_t]$:

```
> R := C(Ex,Et);
```

$$\texttt{R := TD(T, x) - TD(X, t) + C(X, T)}$$

Vector fields $\tilde{D}_x$, $\tilde{D}_t$ determine a covering if and only if $R = 0$.

To explain the above formula for R, we note that the second argument g of `apply(f,g)` can be a function, but can be also another vector field, in which case the vector field f is applied to coefficients of the vector field g. E.g.,

```
> apply(pd[u], u^2*pd[x]);
```

$$\texttt{2 u pd[x]}$$

A tail applied to a local total derivative gives zero:

```
> apply(T, TD[x]);
```

$$0$$

Therefore, we have

$$[D_x, T] = \mathrm{apply}(D_x, T) - \mathrm{apply}(T, D_x) = \mathrm{apply}(D_x, T)$$

which explains the simplification

```
> C(TD[x],T);
```

$$\texttt{TD(T, x)}$$

in the above formula for R. This ends Phase I.

In Phase II we solve the equation $R = 0$ under some assumptions on the dependence of X, T:

```
> dependence(X(u,u_x), T(u,u_x));

                    X = {u_x, u}, T = {u_x, u}
> vars(R);
                      {u_x, u, u_xxx, u_xx}

> pd(R,u_xxx);
                                / d    \
                               -|---- X|
                                \du_x  /

> dependence(X(u));
                      X = {u}, T = {u_x, u}

> pd(R,u_xx*u_x);
```

```
                                  2
                                 d
                                ----- T
                                    2
                                du_x
> T := T1*u_x + T0;
                           T := T1 u_x + T0

> tail(T1,T0);
                              X, T1, T0

> dependence(T1(u),T0(u));

                     T1 = {u}, T0 = {u}, X = {u}

> pd(R,u_xx);
                                  /d   \
                              T1 - |-- X|
                                  \du  /
> T1 := pd(X,u);
                                    d
                               T1 := -- X
                                    du
> pd(R,u_x^2);
                                  / 2   \
                                  |d    |
                                2 |--- X|
                                  |  2  |
                                  \du   /
> X := X1*u + X0;
                             X := X1 u + X0
> tail(X1,X0);
                               X1, X0, T0

> dependence(X1(), X0());

                      X1 = {}, T0 = {u}, X0 = {}

> pd(R,u_x*u^2);
                                   3
                                  d
                                  --- T0
                                    3
                                  du

> T0 := T02*u^2 + T01*u + T00;
```

```
                                    2
                    T0 := T02 u  + T01 u + T00

> tail(T02,T01,T00);
                        T00, T01, X0, T02, X1

> dependence(T02(), T01(), T00());

           X1 = {}, T02 = {}, T01 = {}, T00 = {}, X0 = {}

> pd(R,u_x*u);
                              2 T02 - X1

> T02 := 1/2*X1;
                             T02 := 1/2 X1

> pd(R,u_x);
                            T01 - C(X1, X0)

> T01 := -C(X0,X1);
                            T01 := C(X1, X0)

> F := collect(R,u);
                                                2
F := (- 1/2 C(X1, X0) + C(X1, C(X1, X0))) u

     + (-C(T00, X1) + C(X0, C(X1, X0))) u - C(T00, X0)
```

Coefficients at the powers of u give the three equations (2.8), loc. cit., albeit in different notation: $A = X_1$, $B = X_0$, $C = T_{00}$. This ends Phase II.

In Phase III we choose the space W coordinatized by nonlocal variables. One-dimensional coverings have one nonlocal variable, which we denote w:

```
> nonlocal(w);
                                    w
```

We restrict our demonstration to the case $X_1 = A \neq 0$. Then we can put $X_1 = \partial/\partial w$ (see loc. cit.), therefore:

```
> X1 := pd[w];
                              X1 := pd[w]

> X0 := b*pd[w];
                             X0 := b pd[w]

> T00 := c*pd[w];
                            T00 := c pd[w]
```

```
> dependence(b(w), c(w));
```

$$b = \{w\},\ c = \{w\}$$

and insert into F above:

```
> G := collect(F, u, expand);
```

$$G := -\frac{1}{2}\left(\frac{d}{dw}b\right)\mathrm{pd}[w] + \left(\frac{d^2}{dw^2}b\right)\mathrm{pd}[w]\,u$$

$$+\left(\frac{d}{dw}c\right)\mathrm{pd}[w] + b\,\mathrm{pd}[w] - \left(\frac{d}{dw}b\right)\mathrm{pd}[w]\,u$$

$$-c\,\mathrm{pd}[w]\left(\frac{d}{dw}b\right) + b\left(\frac{d}{dw}c\right)\mathrm{pd}[w]$$

Coefficients at the powers of u provide us with the three expressions found in loc. cit.:

```
> coeff(coeff(G,u,2), pd[w]);
```

$$-\frac{1}{2}\left(\frac{d}{dw}b\right) + \frac{d^2}{dw^2}b$$

```
> coeff(coeff(G,u,1), pd[w]);
```

$$\left(\frac{d}{dw}c\right) + b\left(\frac{d^2}{dw^2}b\right) - \left(\frac{d}{dw}b\right)$$

```
> coeff(coeff(G,u,0), pd[w]);
```

$$-c\left(\frac{d}{dw}b\right) + b\left(\frac{d}{dw}c\right)$$

One-dimensional coverings correspond to functions b, c such that the above expressions are zero. It is now a matter of routine to find them (see loc. cit. for the final result).

17 Zero-curvature representations

Jets originated as a tool to compute zero-curvature representations in automatic regime using the method described in [3]. We introduce the commands to perform the task by means of example of the mKdV equation:

```
> with(linalg):
Warning, new definition for norm
Warning, new definition for trace

> coordinates([t,x],[u],12):
> equation('u_t' = u_xxx - 6*u^2*u_x);
```

$$u\_t = u\_xxx - 6\, u^2\, u\_x$$

```
> A := matrix(2,2, [a1,a2,1,-a1]);
> B := matrix(2,2, [b1,b2,b3,-b1]);
> R := matrix(2,2, [r,0,0,-r]);
```

$$A := \begin{bmatrix} a1 & a2 \\ 1 & -a1 \end{bmatrix}$$

$$B := \begin{bmatrix} b1 & b2 \\ b3 & -b1 \end{bmatrix}$$

$$R := \begin{bmatrix} r & 0 \\ 0 & -r \end{bmatrix}$$

The matrix R is the characteristic matrix of [3] in the normal form with respect to conjugation (the Jordan normal form). The matrix A is in the normal form with respect to gauge equivalence due to the stabilizer of R.

```
> S := zero_curvature(x = A, t = B, 1 = R):
> unknowns(a2,a1,b3,b2,b1,r):
> dependence(a1(u,u_x,u_xx), a2(u,u_x,u_xx),
    b1(u,u_x,u_xx), b2(u,u_x,u_xx), b3(u,u_x,u_xx),
    r(u,u_x,u_xx)):
> nonzero(a2,r):
```

We assume a_2 to be nonzero since otherwise the zero-curvature representation comes out as lower-triangular, hence equivalent to a couple of conservation laws. After a while, `run(S)` stops at:

```
> run(S);
```

```
                                    ⋮

<25>   Put:
                                3
                               d
                               --- b2 = 0
                                 3
                               du

                  2          2      2           /d    \
         b2 = 4 a2  + 4 a2 a1  - 6 u  a2 - 2 a2 |-- a1| u_x
                                                 \du   /

<27>   resolving failed for   pd(a1,u)   nonlinear

                              /d    \2
                              |-- a1|  - 1
                              \du   /

<27>   linear resolving failed for   pd(a1,u)

                             /d    \
                             |-- a1| a1 = u
                             \du   /

                                 FAIL
```

We end up with two simple nonlinear equations easy to solve: $a_1 = u$, which immediately leads to the well-known zero-curvature representation of mKdV with a_2 as a parameter (the spectral parameter):

```
> a1 := u;
                                a1 := u
> run(S);

<2>   Success!

> dependence();
                            a2 = {}, r = {}

> map(eval, op(A));
                               [u    a2]
                               [       ]
                               [1    -u]

> map(expand@eval, op(B));
```

```
[          3                  2      2              ]
[u_xx - 2 u  + 4 u a2    4 a2  - 2 u  a2 - 2 a2 u_x]
[                                                   ]
[           2                        3              ]
[2 u_x - 2 u  + 4 a2       -u_xx + 2 u  - 4 u a2    ]
```

Other features

Jets contains a number of other undocumented procedures. See `Jets.s` for their definitions.

License

Jets is a freeware, distributed under the GNU General Public License as published by the Free Software Foundation <`http://www.gnu.org/copyleft/gpl.html`>. In particular, it is distributed without any warranty, just in the hope that it will be useful. Whenever results of your scientific work depend on this software, you should consider making a proper reference [4].

References

[1] A.V. Bocharov and M.L. Bronstein, Efficiently implementing two methods of the geometrical theory of differential equations: An experience in algorithm and software design, *Acta. Appl. Math.* **16** (1989) 143–166.

[2] W. Hereman, Review of symbolic software for the computation of Lie symmetries of differential equations, *Euromath Bulletin* **1** (1994) 45–82.

[3] M. Marvan, A direct procedure to compute zero-curvature representations. The case sl_2 *Secondary Calculus and Cohomological Physics* Proc. Conf. Moscow 1997 (Electronic version: ELibEMS http://www.emis.de/proceedings/SCCP97, 1998) pp.10

[4] M. Marvan, Jets. A software for diferential calculus on jet spaces and diffieties, ver. 4.9 (December 2003) for Maple V Release 4.

[5] A.G. Meshkov, Computer package for investigation of the complete integrability, A.M. Samoilenko, ed., *Symmetry in Nonlinear Mathematical Physics*, Proc. Third Int. Conf., Kyiv, 1999 (Inst. Math. N.A.S. Ukraine, Kyiv, 2000) 35--46.

[6] V.L. Topunov, Reducing systems of linear differential equations to a passive form, *Acta. Appl. Math.* **16** (1989) 191–206.

[7] A.M. Vinogradov, Local symmetries and conservation laws, *Acta. Appl. Math.* **2** (1984) 21–78.

[8] A.M. Vinogradov, I.S. Krasil'shchik, eds., *Symmetries and Conservation Laws of Systems of Equations of Mathematical Physics* (Amer. Math. Soc., Providence, Rhode Island, 1999).

[9] A.M. Vinogradov, I.S. Krasil'shchik and V.V. Lychagin, *Geometry of Jet Spaces and Nonlinear Partial Differential Equations* (Gordon and Breach, New York, 1986).