`detools` — library for differential equations

Table of contents

Introduction

The `detools` library provides a number of functions for treating differential equations, especially partial differential equations. This includes functions for the generation, manipulation and analysis of differential equations, some functions for the visualisation of (numerical) solutions and a solver for partial differential equations. It should, however, be noted that the `detools` library is still in a very early stage of its development and may functions are not yet very powerful. The two main parts of the `detools` library are currently a rather general Lie symmetry package and a completion package for over-determined systems of differential equations.

The `detools` library is closely related to the MuPAD domains for differential equations; in fact, many functions are just interfaces to methods in these domains, as one important task of the `detools` library is to provide easy access to these methods for users not familiar with domains. This also implies that most functions accept a simplified input for derivatives. Instead of `diff(u(x,y,z),x,y)` one may simply enter `u([x,y])`, i.e. the arguments of the function `u` do not have to be specified; instead one passes as argument a list with the variables with respect to which `u` is differentiated.

The library functions are called by the library name `detools` and the name of the function. Thus with the input

```
>> detools::detSys(u([t]) - u([x, x]), [t, x], [u])
```

one computes the determining system for the generators of the Lie symmetry algebra of the heat equation. This mechanism avoids naming conflicts with other library functions. If this is found to be inconvenient, the routines of the `detools` library may be exported via `export`. After entering

```
>> export(detools, detSys)
```

the function `detools::detSys` may be called directly:

```
>> detSys(u([t]) - u([x, x]), [t, x], [u])
```

All routines of the `detools` library are exported simultaneously by

```
>> export(detools)
```

The functions available in the `detools` library can be listed with

```
>> info(detools)
```

`detools::arbFuns` – number of arbitrary functions in the general solution of an involutive partial differential equation

`detools::arbFuns(q,alpha)` computes the number of arbitrary functions in the general solution of an involutive partial differential equation of order `q` and with Cartan characters `alpha`.

Call(s):

- `detools::arbFuns(q, alpha)`

Parameters:

`q` — the order of the equation: a positive integer.
`alpha` — the Cartan characters: a list of nonnegative integers.

Return Value: a list of integers of the same length as `alpha`. The *i*-th entry gives the number of arbitrary functions with *i* arguments.

Related Functions: `detools::cartan`, `detools::hilbert`

Details:

- `detools::arbFuns` performs a purely combinatorial calculation trying to express the Cartan characters of an involutive partial differential equations as the number of arbitrary functions in the general solution. For first order equations this make always sense; for higher order equations it is possible that negative values occur in the answer.

Example 1. How many arbitrary functions appear in the general solution of Maxwell's equations in electrodynamics? For three-dimensional space the four Cartan characters are 6, 6, 4 and 0. So we enter

```
>> detools::arbFuns(1, [6, 6, 4, 0])

                                  [0, 2, 4, 0]
```

and obtain that (at least formally) the solution space can be parametrised by four functions of three variables and two functions of two variables.

Background:

- This combinatorial counting of arbitrary functions is based on the so-called formal theory of differential equations and essentially analyses formal power series solutions. More details can be found in the article:

- W.M. Seiler: On the arbitrariness of the general solution of an involutive partial differential equation, Journal of Mathematical Physics 35 (1994) 486–498

`detools::autoreduce` – **autoreduction of a system of differential equations**

`detools::autoreduce` autoreduces a system of differential equations, i.e. it tries to simplify the equations as much as possible by entering each equation into all the other ones. The ultimate goal is to achieve a triangular form.

Call(s):

- `detools::autoreduce(sys, indl, depl)`
- `detools::autoreduce(dfs)`

Parameters:

`sys` — the differential equations: a list ofexpressions.
`indl` — the independent variables: a list of (indexed) identifiers.
`depl` — the dependent variables: a list of (indexed) identifiers.
`dfs` — the differential equations: a list of elements of a domain in `Cat::DifferentialFunction(DV)`.

Return Value: an autoreduced list of differential equations; the equations are represented either as expressions (first form of call) or as elements of a domain in `Cat::DifferentialFunction(DV)` (second form of call).

Related Functions: `simplify`

Details:

- Autoreduction is a basic algorithm for the simplification of over-determined systems of differential equations. The appearing derivatives are ordered according to some ranking. Then each equation is solved for its leading derivative with respect to this ranking (if possible) and entered into all other equations in order to eliminate this derivative from them. This strategy is followed as long as substitutions can be performed.
- For linear equations the used algorithm will always succeed in transforming the system to triangular form. In the case of non-linear equations, this depends on whether it is possible to solve for the leading derivatives. In general, it cannot be guaranteed that dependent equations are always removed.

- Obviously, the result of an autoreduction depends on the chosen ranking. A specific ranking can only be prescribed by using an appropriate domain in `Cat::DifferentialFunction(DV)`, i.e. by using the second call of `detools::autoreduce`. Otherwise the default ranking is used which sorts first by the order of the derivative and then reverse lexicographically.

Example 1. We want to simplify the two differential equations $3u_{xy} + u_y = 0$ and $u_x = 0$. Obviously, the second order term in the first equation is a derivative of the second equation. Thus it is eliminated by `detools::autoreduce`.

```
>> detools::autoreduce([3*u([x, y]) + u([y]), u([x])], [x, y], [u])

                    [u([x]), u([y])]
```

`detools::cartan` – Cartan characters of a differential equation

`detools::cartan` determines the Cartan characters of a differential equation either from the indices of the symbol or from the Hilbert polynomial.

Call(s):

- `detools::cartan(n, m, q, beta)`
- `detools::cartan(n, hp)`

Parameters:

`n`	—	number of independent variables: a positive integer.
`m`	—	number of dependent variables: a positive integer.
`q`	—	order of the equation: a positive integer.
`beta`	—	indices of symbol: a list of nonnegative integer.
`hp`	—	Hilbert polynomial: a univariate polynomial (type `DOM_POLY`) with rational coefficients.

Return Value: a list of nonnegative integers.

Related Functions: `detools::arbFuns`, `detools::hilbert`

Details:

- `detools::cartan` determines the Cartan characters of a differential equation from either the indices of the symbol or the Hilbert polynomial. In the first case, the number of independent and dependent variables, respectively, of the equation and its order must be given.

- In the second form to call `detools::cartan` the number of independent variables suffices, as `detools::cartan` assumes that the Hilbert polynomial is given relative to the order of the differential equation, i.e. as a function of $q + r$.

Example 1. The indices of the symbol of Maxwell's equations of electrodynamics (a first order system in four independent and six dependent variables) are `[0,0,2,6]`. They can be converted into Cartan characters with the following command.

```
>> detools::cartan(4, 6, 1, [0, 0, 2, 6])

                                  [6, 6, 4, 0]
```

Alternatively, we may start with the Hilbert polynomial of Maxwell's equations: $H(1 + r) = 2r^2 + 12r + 16$.

```
>> detools::cartan(4, poly(2*r^2 + 12*r + 16, [r], Dom::Rational))

                                  [6, 6, 4, 0]
```

Background:

- Cartan characters are used to measure the size of the (formal) solution space of a system of differential equations. They appear in the so-called formal theory of differential equations and are based on an analysis of power series solutions. More details can be found in the article:
  - W.M. Seiler: On the arbitrariness of the general solution of an involutive partial differential equation, Journal of Mathematical Physics 35 (1994) 486–498

`detools::characteristics` – characteristics of partial differential equation

`detools::characteristics(ldf,s)` determines the characteristics of the linear differential equation `ldf`. The identifier `s` is used as parameter for the curves.

Call(s):

- `detools::characteristics(ldf, s, <, init>)`

Parameters:

`ldf` — the differential equation: an element of a domain generated with the constructor `Dom::LinearDifferentialFunction`.
`s` — the independent variable: an identifier.
`init` — the initial conditions: a list of equations.

Return Value: a list of expressions representing the characteristics in parametric form.

Related Functions: `detools::charODESystem`, `detools::charSolve`

Details:

- `detools::characteristics` tries to determine the characteristics of a given differential equation. For this purpose, it first sets up the characteristic system using the method `detools::charODESystem` and then tries to solve it. As the characteristic system is an in general nonlinear system of ordinary differential equations, this can be a very hard task.
- The implemented form of the method of characteristics works only for (quasi-)linear first order equations in one dependent variable.
- With the optional third argument one can prescribe some initial conditions for the characteristic system. There should be exactly one condition for each independent and dependent variable of the original partial differential equation.

Example 1. With the following input one can determine the characteristics of the differential equation $2u_x + u_y + 3u_z - 2u = 0$.

```
>> LDF := Dom::LinearDifferentialFunction(
                Vars = [[x, y, z], u], Rest = [Types = "Indep"]):
   ldf := LDF( 2*u([x]) + u([y]) + 3*u([z]) - 2*u ):
   detools::characteristics(ldf, tau)

 {[z(tau) = C1 + 3 tau, x(tau) = C2 + 2 tau, y(tau) = C3 + tau,

    u(tau) = C4 exp(2 tau)]}
```

The result gives the characteristic curve in parametric form. The constants `C1`, `C13`, `C14`, `C15` could be fixed by adding some initial condition. It is easy to see that the basis characteristics, i.e. the projection on the space of the independent variables `x`, `y`, `z`, is a straight line and that the solution grows exponentially on it.

`detools::charODESystem` – characteristic system of partial differential equation

`detools::charODESystem(ldf,s)` determines the characteristic system of the linear differential equation `ldf`. The identifier `s` is used as independent variable of this system.

Call(s):

- `detools::charODESystem(ldf, s, <, init>)`

Parameters:

`ldf` — the differential equation: an element of a domain generated with the constructor `Dom::LinearDifferentialFunction`.
`s` — the independent variable: an identifier.
`init` — the initial conditions: a list of equations.

Return Value: an object of the type `ode`.

Related Functions: `detools::characteristics`, `detools::charSolve`

Details:

- `detools::charODESystem` only determines the characteristic system of the given differential equation; it does not attempt to solve it, i.e. to explicitly compute the characteristics. If this is the goal, call directly the method `detools::characteristics`.
- The implemented form of the method of characteristics works only for (quasi-)linear first order equations in one dependent variable.
- With the optional third argument one can prescribe some initial conditions for the characteristic system. There should be exactly one condition for each independent and dependent variable of the original partial differential equation.

Example 1. With the following input one can determine the characteristic system of the differential equation $2u_x + u_y + 3u_z - 2u = 0$ using τ as independent variable of the arising ordinary differential equations.

```
>> LDF := Dom::LinearDifferentialFunction(
               Vars = [[x, y, z], u], Rest = [Types = "Indep"]):
   ldf := LDF( 2*u([x]) + u([y]) + 3*u([z]) - 2*u ):
   detools::charODESystem(ldf, tau)
```

```
ode({diff(x(tau), tau) - 2, diff(y(tau), tau) - 1,

   diff(z(tau), tau) - 3, - 2 u(tau) + diff(u(tau), tau)},

   {u(tau), x(tau), y(tau), z(tau)})
```

`detools::charSolve` – solves partial differential equation with the method of characteristics

`detools::charSolve(ldf,init,pars)` solves the linear differential equation `ldf` by the method of characteristics. The initial conditions `init` should depend on $n-1$ parameters (listed in `pars`), if there are n independent variables.

Call(s):

- `detools::charSolve(ldf, init, pars)`

Parameters:

`ldf`	— the differential equation: an element of a domain generated with the constructor `Dom::LinearDifferentialFunction`.
`init`	— the initial conditions: a list of equations.
`pars`	— the parameters: a list of identifiers.

Return Value: a list of expressions representing the parametric solution of the differential equation for the given initial conditions.

Related Functions: `detools::characteristics`, `detools::charODESystem`, `detools::pdesolve`, `solve`

Details:

- `detools::charSolve(ldf,init,pars)` tries to solve the differential equation `ldf` subject to the parametric initial conditions `init`. The list `pars` contains the names of the parameters. The solution will again be in parametric form. It will be found only, if the characteristic system can be solved.

- The implemented form of the method of characteristics works only for (quasi-)linear first order equations in one dependent variable.

Example 1. With the following input one can solve the linear differential equation $2u_x + u_y + 3u_z - 2u = 0$ for the following parametrized initial condition $x = 2\sigma, y = 3\tau, z = \sigma + \tau, u = \sigma - \tau$.

```
>> LDF := Dom::LinearDifferentialFunction(
                Vars = [[x, y, z], u], Rest = [Types = "Indep"]):
   ldf := LDF( 2*u([x]) + u([y]) + 3*u([z]) - 2*u ):
   detools::charSolve(ldf,
       {x = 2*sigma, y = 3*tau, z = sigma + tau, u = sigma -
tau},
       {sigma, tau})

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

`detools::detSys` – determining system for Lie point symmetries

`detools::detSys` sets up the determining system for the generators of Lie point symmetries of a given system of differential equations. As for most methods in the `detools` library there exist several possibilities for entering the differential equations. The precise working of the method can be controlled by a number of options; especially it is possible to prescribe a special ansatz for the symmetry generators.

Call(s):

- `detools::detSys(de, indl, depl` <, *`Ansatz`* `= ans`, *`Param`* `= paraml`> <, *`Expr`* `= ebool`> <, *`Interactive`* `= bool`> <, *`Autoreduced`* `= bool`>`)`
- `detools::detSys(df` <, *`Ansatz`* `= ans`, *`Param`* `= paraml`> <, *`Expr`* `= ebool`> <, *`Interactive`* `= bool`> <, *`Autoreduced`*`= bool`>`)`

Parameters:

`de` — the differential equation(s): either an expression or a list of expressions.
`indl` — the independent variable(s): a list of (indexed) identifiers.
`depl` — the dependent variable(s): a list of (indexed) identifiers.
`df` — the differential equation(s): either an element of a domain `DF` in `Cat::DifferentialFunction` or a list of such elements.

Options:

Ansatz — prescribes an ansatz for the generators.
Param — lists the names of the parameters (functions or constants) contained in the ansatz.
Expr — determines the type of the output of `detools::detSys`.
Interactive — controls the behaviour, if `detools::detSys` has problems with solving the differential equations for their leading derivatives.
Autoreduced — controls whether the equations of the determining system are automatically simplified (autoreduced) by `detools::detSys`. If `bool=FALSE`, no simplifications are performed.

Return Value: The determining system is returned as a list. The type of the list elements is controlled by the option `Expr`.

Side Effects: `detools::detSys` reads and writes some entries of the table `detools::data`. This includes especially further information about the used domains.

Related Functions: `detools::ncDetSys`

Details:

- `detools::detSys` sets up the determining system for the generators of Lie point symmetries of the given differential equation(s). Thus it returns again a (generally rather overdetermined) system of differential equations whose solutions represent symmetry generators. The many options allow for a rather tight control of the way the calculations proceed. Especially, it is possible to prescribe a special ansatz for the generators which leads often to a considerable speed up.

- Internally all calculations are performed within a suitably chosen domain of `Cat::DifferentialFunction(DV)`. If no specific domains are prescribed, `detools::detSys` generates automatically for `DV` the domain `Dom::DifferentialVariable(indl,depl)`. Thus for `indl` and `depl` anything can be entered that is accepted by this constructor.

  Which domain in `Cat::DifferentialFunction(DV)` `detools::detSys` actually chooses, depends on the form of the entered differential equations. `detools::detSys` prefers to perform all calculations in polynomial arithmetic, as this is considerably faster. Hence the first (and most common) choice is a domain constructed with `Dom::DifferentialPolynomial`. If the equations are of a more general form, `Dom::DifferentialExpression` is used.

- For non-classical symmetries, the determining system can be set up with the method `detools::ncDetSys`

Option <*`Ansatz`*`=ans`>:

- With this option one can prescribe a special ansatz for the symmetry generators. `ans` is either an element of the domain `Dom::JetVectorField(DF)` where `DF` is the domain in which internally the calculations are performed or any expression which can be converted into such an element. If no ansatz is prescribed, a generic one is employed. If this option is used, the option *`Param`* must be used, too.
- A very simple use of this option is to choose one's own names for the coefficients of the generic ansatz. Assume we are given a differential equation for the unknown function $u(x,t)$. Then a symmetry generator is a vector field on a three dimensional manifold with the coordinates x, t, u. In order to call the corresponding coefficients ξ, τ, η, one can use this option with the value `Ansatz=[[xi(x,t,u),x], [tau(x,t,u),t], [eta(x,t,u),u]]`.

Option <*`Param`*`=paraml`>:

- This option makes sense only in connection with the option *`Ansatz`*. `paraml` is a list of identifiers; the determining system consists of equations for these parameters. In the example above `paraml` would take the value `[xi,tau,eta]`.

Option <*`Expr`*`=ebool`>:

- This option controls the type of the output. If `ebool=TRUE`, the output will consists of expressions. The same holds for `ebool=NoDiff`; however, in this case for derivatives the condensed notation of the domains in `Cat::DifferentialVariable` is used. For `ebool=FALSE`, the output will consist of elements of a domain generated by a call of the constructor `Dom::LinearDifferentialFunction` with appropriate arguments.

  The default behaviour is determined by the way `detools::detSys` is called. If the differential equations are entered as expressions, the default corresponds to `ebool=NoDiff`. If domain elements are used, it corresponds to `ebool=FALSE`.

Option <`Interactive`=`bool`>:

- `detools::detSys` might encounter problems in solving each differential equations for a different derivative. If `bool=TRUE`, `detools::detSys` will ask interactively the user for help. If `bool=FALSE`, the computation will be aborted in case of troubles.

Example 1. We compute the determining system for the heat equation $u_t - u_{xx} = 0$.

```
>> detools::detSys(u([t])-u([x,x]),[t,x],[u])
 [2 XI1([u]), 2 XI1([x]), XI2([u, u]), XI1([u, u]),

    2 XI2([x, u]) - PHI1([u, u]), 2 XI2([u]) + 2 XI1([x, u]),

    XI2([x, x]) - XI2([t]) - 2 PHI1([x, u]),

    2 XI2([x]) - XI1([t]) + XI1([x, x]),

    PHI1([t]) - PHI1([x, x])]
```

The output is a linear system of nine differential equations. The unknown functions `XI1`, `XI2` and `PHI1` represent the coefficients of the `t`-, `x`- and `u`-component of the symmetry generator, resp.

Background:

- Lie symmetry analysis is one of the most important techniques for studying differential equations. More information about it and especially about its mathematical background and its many application can be found in the following text books:
  - P.J. Olver: Applications of Lie groups to Differential Equations, Graduate Texts in Mathematics 107, Springer, New York 1986
  - G.W. Bluman, S. Kumei: Symmetries and Differential Equations, Applied Mathematical Sciences 81, Springer, New York 1989
- It is not so easy to give a completely fool proof implementation of setting up the determining system. `detools::detSys` should work correctly for any system of differential equations satisfying the following conditions: (i) the system is formally integrable (this is always true for single equations and for systems in Cauchy-Kowalevsky form); (ii) nonlinearities in the derivatives are only of polynomial type and (iii) all equations can be solved for a different derivative. If one of these conditions is violated, the user should careful check the results (note: these are not artificial restrictions of `detools::detSys` but fundamental mathematical problems!).

- If the system is not formally integrable, the infinitesimal approach which underlies most of Lie symmetry analysis is no longer sufficient; it may not find all existing symmetries. In cases of doubt, a completion with the method `detools::complete` can be used to check for integrability
- If transcendent terms are present in the equations, the determining system may be set up incorrectly, as `detools::detSys` is not able to decide whether there exist any algebraic dependencies between these terms. The determining system is set up under the assumption that no such relations exist.

  Essentially for the same reason, the current implementation also requires that all equations can be solved for different derivatives. If this is the case, it is trivial to take into account the relations introduced by the differential equations themselves.

`detools::derList2Tree` – minimal tree with a given list of derivatives as leaves

`detools::derList2Tree(derl)` takes a list of derivatives (more precisely, their multi indices) and determines a minimal tree of derivatives such that the given derivatives are the leaves.

Call(s):

- `detools::derList2Tree(derl)`

Parameters:

`derl` — list of multi indices: a list of lists of nonnegative integer.

Return Value: a list structure representing the spanning tree. Each leaf is represented by an integer denoting its position in the list `derl`. A node consists of a multi index saying by what derivative the node can be reached and of a subtree with the same structure.

Details:

- `detools::derList2Tree` is an auxiliary procedure that is used, for example, by some methods in domains in `Cat::DifferentialFunction`. It is useful, if several derivatives of the same function must be computed. The determined tree describes a way to compute all required derivatives with a minimal number of differentiations.
- The used algorithm is heuristic. It is not known whether it always produces an optimal result.

Example 1. Assume we are given a function $F(x, y, z)$ and we need the following three derivatives of it: F_{xyyzzz}, $F_{xxyyzzzz}$, F_{xyzzzz}. What is the most efficient way to compute them?

```
>> detools::derList2Tree([[1, 2, 3], [2, 2, 4], [1, 1, 4]])

   [[1, 1, 3], [[[0, 1, 0], 1], [[0, 0, 1], 3, [1, 1, 0], 2]]]
```

This result can be interpreted as follows. First compute $G = F_{xyzzz}$. Then the first required derivative is given by G_y, the third one by $H = G_z$ and the second one by H_{xy}.

`detools::euler` – Euler operator of variational calculus

`detools::euler(L,t,z)` applies the Euler operator to the Lagrangian L and returns the left hand side of the corresponding Euler-Lagrange equations. The Lagrangian can be of any order and there can be any number of independent variables t and of dependent variables z.

Call(s):

- `detools::euler(L, t, z)`
- `detools::euler(L, DV)`

Parameters:

`L` — the Lagrangian: an expression or an element of a domain of `Cat::DifferentialFunction(DV)`.
`t` — the independent variable(s): either a single (indexed) identifier or a list of (indexed) identifiers.
`z` — the dependent variable(s): either a single identifier or a list of identifiers.
`DV` — the domain of the differential variables: `DV` must belong to `Cat::DifferentialVariable`.

Return Value: either a single expression or a list ofexpressions; if for L an element of a domain of `Cat::DifferentialFunction` was given, the output will also consist of elements of this domain.

Details:

- Let z be some functions of the variable(s) t. Let furthermore L be a function depending on t, z and the derivatives of z with respect to t up to a fixed order q. Then the Euler operator yields differential equations for z which are necessary conditions for the function z to be a minimum of the action integral, i.e. the integral over L with respect to all variables t.

- Internally all calculations are performed within suitably chosen domains of `Cat::DifferentialFunction`. If an object of type `DOM_EXPR` is passed as Lagrangian *L*, conversions are needed which may cost some time.

- If no specific domains are prescribed, `detools::euler` generates automatically for `DV` the domain `Dom::DifferentialVariable(t,z)` and the calculations are performed in the domain `Dom::DifferentialExpression(DV)`. Thus for `t` and `z` anything can be entered that is accepted by the constructor `Dom::DifferentialVariable`.

Example 1. This is a finite dimensional example computing the equations of motion of a particle moving in the plane under the influence of a potential V.

```
>> L := 1/2*(diff(x(t), t)^2 + diff(y(t), t)^2) - V(x(t), y(t)):
   detools::euler(L, t, [x, y])

   [x([t, t]) + D([1], V)(x, y), y([t, t]) + D([2], V)(x, y)]
```

Example 2. This is a simple example for the generation of field equations. The field u depends here on two variables t, x.

```
>> L := 1/2*(diff(u(t, x), t)^2 + diff(u(t, x), x)^2) - u(t, x)^2:
   detools::euler(L, [x, t], u)

                       u([t, t]) + u([x, x]) + 2 u
```

`detools::hasHamiltonian` – check for Hamiltonian vector field

`detools::hasHamiltonian(vf,q,p)` checks whether the vector field `vf` in the variables `q` and `p` is Hamiltonian.

Call(s):

- `detools::hasHamiltonian(vf, p, q)`

Parameters:

`vf` — the vector field: a list of expressions; its length must be twice the length of the list `q`.
`q` — the position variables: a list of (indexed) identifiers.
`p` — the momentum variables: a list of (indexed) identifiers; must have the same length as the list `q`.

Return Value: a list of expressions; each component represents an integrability condition which must be satisfied for the vector field `vf` to be Hamiltonian. If the list is empty, `vf` is unconditionally Hamiltonian.

Related Functions: `detools::hasPotential`

Details:

- A vector field v with $2n$ components is called Hamiltonian, if there exists a function $H(q, p)$ where q and p are vectors of the length n such that the first n components of v are given by $\partial H/\partial p$ and the last n components by $-\partial H/\partial q$. `detools::hasHamiltonian` computes necessary and sufficient conditions for the existence of such a function H; it does not try to determine H.
- `detools::hasHamiltonian` assumes that `q` and `p` represent canonical variables; i.e. it tests only whether `vf` is Hamiltonian with respect to the standard symplectic structure of $\mathbb{R}^{2n}$ for some integer n.

Example 1. In the following example it is checked whether the vector field describing the motion of a one-dimensional particle under the influence of a force `F` is Hamiltonian.

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

                                  []
```

As one can see, in one dimension the motion is Hamiltonian for any force `F`. In higher dimensions this is no longer true, cf. Ex. 2.

Example 2. This is basically the same example as Ex. 1 but now in two dimensions.

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

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

Now we obtain an integrability condition which must be satisfied by the force components `F` and `G`.

`detools::hasPotential` – check for gradient vector field

`detools::hasPotential(vf,x)` checks whether the vector field `vf` in the coordinates `x` is the gradient of some potential.

Call(s):

- `detools::hasPotential(vf, x)`

Parameters:

`vf`	— the vector field: a list of expressions; its length must be the same as that of the list `x`.
`x`	— the coordinates: a list of (indexed) identifiers.

Return Value: a list of expressions; each component represents an integrability condition which must be satisfied for the vector field `vf` to possess a potential. If the list is empty, `vf` is unconditionally a gradient.

Related Functions: `detools::hasHamiltonian`

Details:

- A vector field v is a gradient field, if there exists a function $V(x)$ depending on a vector x such that the components of v are given by $\partial V/\partial x$. `detools::hasPotential` computes necessary and sufficient conditions for the existence of such a potential V; it does not try to determine V.

Example 1. With the following input one can determine the condition on the components of a two-dimensional vector field so that the field is a gradient.

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

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

`detools::hilbert` – Hilbert polynomial of a differential equation

`detools::hilbert(alpha,r)` computes the Hilbert polynomial of a differential equation with Cartan characters `alpha`. The identifier `r` is used as variable of the polynomial.

Call(s):

- `detools::hilbert(alpha, r)`

Parameters:

`alpha`	— the Cartan characters: a list of nonnegative integers.
`r`	— variable for the polynomial: an identifier.

Return Value: a univariate polynomial (of type `DOM_POLY`) in the variable `r` with rational coefficients.

Related Functions: `detools::cartan`

Details:

- `detools::hilbert(alpha,r)` determines the Hilbert polynomial of a differential equation with Cartan characters `alpha`. The result is a univariate polynomial in the variable `r`. The Hilbert polynomial should always be interpreted relative to the order of the differential equation. If the Cartan characters have been computed at order q, the arising polynomial is $H(q+r)$.

Example 1. The Cartan characters of Maxwell's equations of electrodynamics are `[6,6,4,0]`. The corresponding Hilbert polynomial is

```
>> detools::hilbert([6, 6, 4, 0], r)

                  2
          poly(2 r  + 12 r + 16, [r], Dom::Rational)
```

`detools::modode` – **modified equation**

`detools::modode` implements the method of the modified equation for the analysis of numerical integration methods applied to ordinary differential equations.

Call(s):

- `detools::modode(Psi, depvars, indvar, step, order)`
- `detools::modode(F, method, depvars, indvar, step, order)`

Parameters:

`Psi` — the step function for the numerical method applied to the given differential equation: a list of expressions

`depvars` — the names of the unknown functions: a list of (indexed) identifiers.

`indvar` — the name of the independent variable: an (indexed) identifier.

`step` — the name of the step size: an (indexed) identifier.

`depvars` — the order (in `step`) to which the modified equation should be computed: a positive integer.

`F` — the right hand side of the differential equation: a procedure of the same form as required by `numeric::odesolve`.

`method` — the name of the chosen numerical method: a string.

Return Value: a list of expressions representing the right hand side of the modified equation.

Related Functions: `numeric::odesolve`

Details:

- The method of the modified equation is based on the following observation. Given a differential equation a numerical integration scheme will produce a sequence of points approximating a solution of the equation. There exist differential equations such that these points lie very close to their solutions (closer than to the approximated solution of the original equation). Such equations are called modified equations and can be computed as truncated power series in the step size `step`. Their analysis allows statements about the properties of the chosen numerical method applied to the given equation.

- The application of the method of the modified equation is straightforward for explicit one-step methods.

- For numerical methods implemented in `numeric::odesolve` it is not necessary to give explicitly the step function `Psi`. Instead one can give to `detools::modode` the right hand side `F` of the differential equation in the form used by `numeric::odesolve` and the name `method` of the integrator one is interested in. `detools::modode` will then call `numeric::odesolve` with the option *`Symbolic`* and thus automatically derive the step function `Psi`.

Example 1. The following input determines the modified equation of order 3 for the (forward) Euler method applied to the differential equation $\dot{y} = z$ and $\dot{z} = -y$.

```
>> detools::modode([z, - y], [y, z], t, h, 3)

       --            2      3               2      3   --
       |      h y   h  z   h  y  h z       h  y   h  z  |
       |  z + --- - ---- - ----, --- - y + ---- - ----  |
       --      2     3      4     2         3      4   --
```

The same result is obtained with the following sequence of commands.

```
>> F := proc(t,y) begin [y[2], - y[1]] end_proc:
   detools::modode(F, EULER1, [y, z], t, h, 3)

       --            2      3               2      3   --
       |      h y   h  z   h  y  h z       h  y   h  z  |
       |  z + --- - ---- - ----, --- - y + ---- - ----  |
       --      2     3      4     2         3      4   --
```

`detools::ncDetSys` – determining system for non-classical Lie symmetries

`detools::ncDetSys` sets up the determining system for the generators of non-classical Lie point symmetries of a given system of differential equations. As for most methods in the `detools` library there exist several possibilities for entering the differential equations. The precise working of the method can be controlled by a number of options; especially it is possible to prescribe a special ansatz for the symmetry generators.

Call(s):

- detools::ncDetSys(de, indl, depl <, *Ansatz* = ans, *Param* = paraml> <, *Expr* = ebool> <, *Interactive* = bool> <, *Autoreduced* = bool> <, *Steps* = n>)
- detools::ncDetSys(df <, *Ansatz* = ans, *Param* = paraml> <, *Expr* = ebool> <, *Interactive* = bool> <, *Autoreduced*= bool> <, *Steps* = n>)

Parameters:

`de`	—	the differential equation(s): either a single expression or a list of expressions.
`indl`	—	the independent variable(s): a list of (indexed) identifiers.
`depl`	—	the dependent variable(s): a list of (indexed) identifiers.
`df`	—	the differential equation(s): either an element of a domain `DF` in `Cat::DifferentialFunction` or a list of such elements.

Options:

`Ansatz` — prescribes an ansatz for the generators.
`Param` — lists the names of the parameters (functions or constants) contained in the ansatz.
`Expr` — determines the type of the output of `detools::ncDetSys`.
`Interactive` — controls the behaviour, if `detools::ncDetSys` has problems with solving the differential equations for their leading derivatives.
`Autoreduced` — controls whether the equations of the determining system are automatically simplified (autoreduce) by `detools::ncDetSys`. If `bool=FALSE`, no simplifications are performed.
`Steps` — determines the level at which non-classical symmetries are sought.

Return Value: The determining system is returned as a list. The type of the list element is controlled by the option `Expr`.

Side Effects: `detools::ncDetSys` reads and writes some entries of the table `detools::data`. This includes especially further information about the used data types.

Related Functions: `detools::detSys`

Details:

- `detools::ncDetSys` sets up the determining system for the generators of non-classical Lie symmetries of the given differential equation(s). Thus it returns again a (generally rather overdetermined) system of differential equations whose solutions represent symmetry generators. The many options allow for a rather tight control of the way the calculations proceed. Especially, it is possible to prescribe a special ansatz for the generators which leads often to a considerable speed up.

- Internally all calculations are performed within a suitably chosen domain of `Cat::DifferentialFunction(DV)`. If no specific domains are prescribed, `detools::ncDetSys` generates automatically for `DV` the domain `Dom::DifferentialVariable(indl,depl)`. Thus for `indl` and `depl` anything can be entered that is accepted by this constructor.

  Which domain in `Cat::DifferentialFunction(DV)` `detools::ncDetSys` actually chooses, depends on the form of the entered differential equations. `detools::ncDetSys` prefers to perform all calculations in polynomial arithmetic, as this is considerably faster. Hence the first (and most common) choice is a domain constructed with `Dom::DifferentialPolynomial`.

If the equations are of a more general form, `Dom::DifferentialExpression` is used.

- For classical symmetries, the determining system can be set up with the method `detools::detSys`

Option <*`Ansatz`*`=ans`>:

- With this option one can prescribe a special ansatz for the symmetry generators. `ans` is either an element of the domain `Dom::JetVectorField(DF)` where `DF` is the domain in which internally the calculations are performed or any expression which can be converted into such an element. If no ansatz is prescribed, a generic one is employed. If this option is used, the option *`Param`* must be used, too.
- A very simple use of this option is to choose one's own names for the coefficients of the generic ansatz. Assume we are given a differential equation for the unknown function $u(x,t)$. Then a symmetry generator is a vector field on a three dimensional manifold with the coordinates x, t, u. In order to call the corresponding coefficients ξ, τ, η, one can use this option with the value `Ansatz=[[xi(x,t,u),x], [tau(x,t,u),t], [eta(x,t,u),u]]`.

Option <*`Param`*`=paraml`>:

- This option makes sense only in connection with the option *`Ansatz`*. `paraml` is a list of identifiers; the determining system consists of equations for these parameters. In the example above `paraml` would take the value `[xi,tau,eta]`.

Option <*`Expr`*`=ebool`>:

- This option controls the format of the output. If `ebool=TRUE`, the output will consists of expressions. The same holds for `ebool=NoDiff`; however, in this case for derivatives the condensed notation of the domains in `Cat::DifferentialVariable` is used. For `ebool=FALSE`, the output will consist of elements of a domain generated by the constructor `Dom::LinearDifferentialFunction` with appropriate arguments.

  The default behaviour is determined by the way `detools::ncDetSys` is called. If the differential equations are entered as expressions, the default corresponds to `ebool=NoDiff`. If domain elements are used, it corresponds to `ebool=FALSE`.

Option <*`Interactive`*=`bool`>:

- `detools::ncDetSys` might encounter problems in solving each differential equations for a different derivative. If `bool=TRUE`, `detools::ncDetSys` will ask interactively the user for help. If `bool=FALSE`, the computation will be aborted in case of troubles.

Option <*`Steps`*=`n`>:

- This option controls at which level the non-classical symmetries are sought. This means that the parameter `n` controls after how many steps consistency between the original system and the invariant surface condition has been achieved. The value `n=1` corresponds to the well-known Bluman-Cole approach but higher values are possible, too.

Example 1. The first example for a non-classical symmetry was found for the heat equation $u_t - u_{xx} = 0$. With the following command one sets up the determining equations for the first level of non-classical symmetries.

```
>> detools::ncDetSys(u([t]) - u([x, x]), [t, x], [u], Steps=1)

 [(-2) (XI1([u]) XI1), 2 (XI1([u]) PHI1) + 2 (XI1([x]) XI2),

                       2                      3
    XI1([u, u]) XI2 XI1  - XI2([u, u]) XI1 ,

                           2
    3 (XI2([u, u]) PHI1 XI1 ) - 2 (XI1([u, u]) PHI1 XI2 XI1) -


                        2                     2
    PHI1([u, u]) XI2 XI1  - 2 (XI1([x, u]) XI2  XI1) +

                          2                   2
    2 (XI2([x, u]) XI2 XI1 ) - 2 (XI2([u]) XI2  XI1),

                    2                          2
    XI1([u, u]) PHI1  XI2 - 3 (XI2([u, u]) PHI1  XI1) +

                                                               2
    2 (PHI1([u, u]) PHI1 XI2 XI1) + 2 (XI1([x, u]) PHI1 XI2 ) -


                                                          2
    4 (XI2([x, u]) PHI1 XI2 XI1) + 2 (PHI1([x, u]) XI2  XI1) +

                   3                  2
    XI1([x, x]) XI2  - XI2([x, x]) XI2  XI1 +

                        2                   3
    2 (XI2([u]) PHI1 XI2 ) + 2 (XI2([x]) XI2 ) -

                3               2                       3
    XI1([t]) XI2  + XI2([t]) XI2  XI1, XI2([u, u]) PHI1  -


                     2                           2
    PHI1([u, u]) PHI1  XI2 + 2 (XI2([x, u]) PHI1  XI2) -

                            2                        2
    2 (PHI1([x, u]) PHI1 XI2 ) + XI2([x, x]) PHI1 XI2  -

                    3                    2                3
    PHI1([x, x]) XI2  - XI2([t]) PHI1 XI2  + PHI1([t]) XI2 ]
```

If one compares with the determining system for the classical symmetries, one sees that the equations obtained here are not only considerably more involved, they are also non-linear. This is typical for non-classical symmetries and makes it much harder to solve the determining system. The problem becomes even more severe for higher values of *`Steps`*.

Background:

- Lie symmetry analysis is one of the most important techniques for studying differential equations. More information about it and especially about its mathematical background and its many application can be found in the following text books:

  - P.J. Olver: Applications of Lie groups to Differential Equations, Graduate Texts in Mathematics 107, Springer, New York 1986
  - G.W. Bluman, S. Kumei: Symmetries and Differential Equations, Applied Mathematical Sciences 81, Springer, New York 1989

  Non-classical symmetries have been introduced by Bluman and Cole already in the late 60s. But only in recent years a better understanding and a more complete theory has been developed.

- ⚠ It is not so easy to give a completely fool proof implementation of setting up the determining system. `detools::ncDetSys` should work correctly for any system of differential equations satisfying the following conditions: (i) the system is formally integrable (this is always true for single equations and for systems in Cauchy-Kowalevsky form); (ii) nonlinearities in the derivatives are only of polynomial type and (iii) all equations can be solved for a different derivative. If one of these conditions is violated, the user should careful check the results (note: these are not artificial restrictions of `detools::ncDetSys` but fundamental mathematical problems!).

- If the system is not formally integrable, the infinitesimal approach which underlies most of Lie symmetry analysis is no longer sufficient; it may not find all existent symmetries. In cases of doubt, a completion with the method `detools::complete` can be used to check for integrability.
- If transcendent terms are present in the equations, the determining system may be set up incorrectly, as `detools::ncDetSys` is not able to decide whether there exist any algebraic dependencies between these terms. The determining system is set up under the assumption that no such relations exist.

  Essentially for the same reason, the current implementation also requires that all equations can be solved for different derivatives. If this is the case, it is trivial to take into account the relations introduced by the differential equations themselves.

`detools::pdesolve` – **solver for partial differential equations**

`detools::pdesolve` solves partial differential equations.

Call(s):

- `detools::pdesolve(pde, indl, depl)`
- `detools::pdesolve(df, DV)`

Parameters:

`pde`	— the differential equation(s): either a single expression or a list of expressions.
`indl`	— the independent variables: a list of (indexed) identifiers.
`depl`	— the dependent variables: a list of (indexed) identifiers.
`df`	— the differential equation(s): either a single element of a domain in `Cat::DifferentialFunction(DV)` or a list of such elements.
`DV`	— the differential variables: a domain in `Cat::DifferentialVariable`.

Return Value: a single expression or a list of expressions; each entry represents a component of the solution. If `detools::pdesolve` is not able to solve the equation, it is returned unchanged.

Related Functions: `detools::charSolve`, `detools::detSys`, `solve`

Details:

- Solving partial differential equations is a very difficult task and one should not expect miracles from `detools::pdesolve`. Currently, the main technique used is the method of characteristics. Thus `detools::pdesolve` can basically only solve some quasi-linear first order equations. Further solution techniques will be added in the future.

Example 1. The following call solves the quasi-linear equation $u_t - u(u_x + u_y) = 0$ with the initial condition $u(t = 0) = x + y$. For the input of the differential equation a condensed notation is used (`u([t])` instead of `diff(u(x,y,t),t)` etc.); the initial condition is given in parametrized form.

```
>> detools::pdesolve(u([t]) - u*(u([x]) + u([y])), [t, x, y], [u],
            {t = 0, x = sigma, y = tau, u = sigma + tau},
            {sigma, tau})

                              2 (t x - x - t y)
                  u = y - x + -----------------
                                   2 t - 1
```

One can easily check by entering this expression into the differential equations that it is indeed a solution (don't forget to use `normal`!) and that for $t = 0$ we have $u = x + y$ as required by the initial condition.

`detools::transform` – change of variables for differential equations

`detools::transform` performs variable transformations in differential equations.

Call(s):

- `detools::transform(de, indl, depl, mode, <, NewVars = varl> <ChangeOfVars = cl>)`

Parameters:

`de` — the differential equation: an expression.
`indl` — the independent variable(s): a list of (indexed) identifiers.
`depl` — the dependent variable(s): a list of (indexed) identifiers.
`mode` — transformation mode: either the string `"Indep"` or `"Dep"`.

Options:

NewVars — lists the names of the new variables.
ChangeOfVars — defines the new variables.

Return Value: an expression.

Details:

- `detools::transform` performs variable transformations in differential equations. It represents only an interface to methods implemented in domains in `Cat::DifferentialFunction`. So the allowed transformations depend on the type of the differential equation. For example, for linear equations only linear transformations of either the dependent or the independent variables are permitted.

Example 1. We transform the independent variables in a simple linear differential expression. Note that the new variables are given as linear functions of the old ones.

```
>> detools::transform(u([x]) + u([y]), [x, y], [u], NewVars = [X, Y],
                 ChangeOfVars = [X = x + y, Y = x - y], "Indep")

                                2 u([X])
```

Example 2. Now we transform the dependent variable. Here the old variable must be given as a linear function of the new one.

```
>> detools::transform(u([x]) + u([y]), [x, y], [u], NewVars = [U],
                 ChangeOfVars = [u=3*U], "Dep")

                           3 U([x]) + 3 U([y])
```