taylor User's Manual

version 2.0.0

Joan Gimeno `<joan@maia.ub.es>`   Àngel Jorba `<angel@maia.ub.es>`
Maorong Zou `<mzou@math.utexas.edu>`

September 9, 2022

Contents

1 What is taylor

taylor is an Ordinary Differential Equation (ODE) solver generator. It reads a system of ODEs and outputs an ANSI C routine that performs a single step of numerical integration using the Taylor method. Each step of integration chooses the step and the order adaptively to keep the local error below a given threshold and minimize the global computational effort. This routine is meant to be called from a user main program to perform the desired numerical integration. This version of **taylor** (2.*) extends the functionalities from [1] by adding support of jet transport [2]. I.e., generating code to integrate the variational equations (of any order).

Other features of **taylor** are:

i. Automatic generator source code.

ii. High level of reusability.

iii. Flexibility in arithmetic definition, e.g., multi-precision arithmetic.

iv. Thread safe code (NEW).

taylor is a competitive software product in solving ODEs and it is useful in computing:

- high accuracy numerical ODE-solutions,

- directional variational equations,

- first order variational equations, and

- high-order variational equations.

taylor has direct applications in broad range of areas. It allows to compute, for instance, stability of equilibria, maximal Lyapunov exponents, and high-order derivatives of Poincaré maps.

taylor highlights

- **taylor** is a high-order ODE-solver.

- **taylor** needs input of ODEs in its natural form.

- **taylor** admits user-defined arithmetic via a set of predefined `MY_FLOAT` macros.

- **taylor** works with multi-precision arithmetic libraries.

- **taylor** uses an optimized step size control.

- **taylor** allows user-defined step size control.

- **taylor** is OpenMP compatible. Declaration is required for thread dependent external parameters.

- **taylor** works with jet transport via its `MY_JET` macros.

What are the major differences with previous versions

taylor (2.*) is compatible with version 1.* with the following caveats:

1.) There is a new extra argument used for jet transport in the single step function. It needs to be set to `NULL` when jet transport is not needed.

2.) The header file now is ODE model dependent. Generation of header file requires an ODE model.

2   How to obtain taylor

taylor is released under the GNU Public License (GPL). It is available on Github

`https://github.com/joang/taylor2-dist`

taylor is also available on the web using the URLs

```
http://www.math.utexas.edu/users/mzou/taylor/     (US)
http://www.maia.ub.es/~angel/taylor/              (Europe)
```

3   How to install taylor

taylor runs on Linux systems and it should compile and run on other variant of Unixes. It has been tested on Linux and OsX.

3.1   Installation from a repository

Install instructions on Debian based systems

- Install the Taylor repository key (running as the root user)

  `wget -qO - https://web.ma.utexas.edu/repos/deb/taylor.gpg.key | apt-key add -`

- Add the Taylor repository

  `echo "deb http://web.ma.utexas.edu/repos/deb focal main " > /etc/apt/sources.list.d/taylor.list`

- Install taylor

  ```
  apt-get update
  apt-get install taylor
  ```

Install instructions on RedHat/Centos systems

- Setup the repository (running as the root user)

  ```
  cd /etc/yum.repos.d
  wget https://web.ma.utexas.edu/repos/rpm/taylor-redhat8.repo
  ```

- Install taylor

  `yum install taylor`

3.2   Installation from a Tarball

After downloading the distribution `taylor-x.y.z.tgz`, where `x.y.z` is the version number, unpack the archive using the command

```
tar xvzf taylor-x.y.z.tgz
```

or, if your version of `tar` does not handle compressed files, you can also use

```
gzip -dc taylor-x.y.z.tgz | tar xvf -
```

This will create a directory `Taylor-x.y`. Change to this directory.
Now, to compile **taylor**, run `make`. It will produce the executable `taylor` in the current directory. You need an ANSI C compiler and `lex/yacc` parser generator to compile **taylor**. Using `gcc` and `flex/bison` is highly recommended.
To install **taylor**, simply run

```
make install
```

4   What is jet transport in taylor

Jet transport refers to the time transport of what is called *jet* through an ODE-solver. In **taylor**, a jet is implemented using truncated formal power series. Given m symbols $s_1, s_2, \ldots, s_m$ and a fixed degree d, we represent the state variables as

$$\vec{x} = \vec{x}_0 + \sum_{0 < k_1 + k_2 + \cdots + k_m \leq d} \vec{a}_{k_1 k_2 \ldots k_m} {s_1}^{k_1} {s_2}^{k_2} \cdots {s_m}^{k_m}. \tag{1}$$

$\vec{x}$ evolves using a Taylor ODE-solver method. During the process, we replace the normal arithmetic on numbers with the same on truncated power series. Different values of m and d give particular interpretations, such as, when

- $d = 0$: Usual Taylor method.

- $d = 1$ and $m = 1$: Directional first order variational equation.

- $m = 1$: First order variational equation.

For efficiency, we implement several versions of arithmetic on truncated power series although in the current version 2.0.0 the default one is `jet_tree`. These implementations are user transparent unless you want to modify them. Here is the list.

- `jet1_1` one symbol, degree one. The arithmetic is implemented using C macros.

- `jet1` one symbol, arbitrary degree.

- `jet_1` degree one, arbitrary number of symbols.

- `jet2` two symbols, arbitrary degree.

- `jet_2` degree two, arbitrary number of symbols.

- `jet_m` a naive implementation of the general case, arbitrary number of symbols, and arbitrary degree. It works well when the jet size is below 1000 terms, i.e., whenever $\binom{d+m}{d} < 1000$.

- `jet_tree` a general library[1].

5   What is the syntax in taylor

ODE declaration

To use **taylor**, the first order of business is to prepare an input file with the system of ODEs. ODEs are specified by statements like

```
id = expr;
diff(var, tvar) = expr;
```

where `tvar` is the time variable and `expr` is a valid mathematical expression made from numbers, the time variable, the state variables, elementary functions sin, cos, tan, arctan, sinh, cosh, tanh, exp, and log, using the four arithmetic operators and function composition. For example,

```
a = log(1 + exp(-0.5));
b = a + cos(0.1);
c = a+b;
ff = sin(x+t) * exp(-x*x);
diff(x,t) = c * ff - tan(t);
```

---

[1]based on `http://www.maia.ub.es/dsg/param/chapter2.html`

defines a single ODE.

taylor also understands `if-else` expressions and non-nested sums with static range. For example,

```
ss = sum( i*sin(i * x)+ i *cos(i*t), i=1,10);
diff(x,t) = ss;
diff(y,t) = if(y>t) {  if(y>0.0) { y } else { 1-y }  }
            else    { y+t};
```

Parameter declaration

taylor allows to declare undefined parameters in the input file following the syntax

```
extern MY_FLOAT param;
```

where `extern` is a reserved **taylor**'s name, `MY_FLOAT` is the usual type (although it may be of other type, such as, `double`, etc.), and `param` refers to the parameter name.

Jet variables declaration

Jet variables are declared using the statement

```
jet [var_list|all] symbols [Number_of_symbols] degree [Degree];
```

For example,

```
jet x,y symbols 2 degree 2;
jet all symbols 3 degree 1;
```

In the first example, the variables `x,y` will be treated as polynomials of 2 symbols of degree up to 2.
In the second example, all the state variables will be polynomials of 3 symbols and degree 1. Note that if our system of ODEs consists of 3 equations, this is equivalent of integrating the ODEs along with its first order variational equations.

If you only declare a subset of state variables as jet variables, you need to make sure that

A1.) the order of the variables must match that in the ODE specification, and they must appear before non-jet variables;

A2.) the declared jet variables must be self contained, that is, all variables depends on any of the jet variables are also jet variables.

Assumptions A1–A2 force the system to have a coherent output. For instance, the assumptions cover cases like systems of the form

$$
\begin{aligned}
\dot{x} &= f(x, y), \\
\dot{y} &= g(y).
\end{aligned}
$$

If y is a jet variable, then x must be a jet variable due to (A2). However, it is allowed to consider x a jet variable and y not; as far as f is written first (A1).

6   How to run taylor

Once the input file with the ODE system is ready, there are three main steps in the construction of the **taylor** integrator.

1.) We ask **taylor** to produce the code to compute the derivatives of the state variables, and the automatic step size (and order) control.

2.) We ask **taylor** to produce a header file that contains the definition of the data type, macros for basic arithmetic and forward declarations of API functions.

3.) We write a main driver function that repeatedly call the taylor integrator.

Step 1 generates a source code that is arithmetic-independent, in the sense that the real numbers are declared as `MY_FLOAT` (type that is defined in step 2). All the arithmetic operations have been replaced by C-macros.

Step 2 generates a file that must be included by the C-file generated in Step 1 and possibly other generated or custom coded C files.

Alternatively, **taylor** can also generate a sample main driver function. The above steps can be combined in a single one, by asking **taylor** to output everything in a single file, that is, jet, step size control, header, and a sample main.

See Section 7 for illustrative examples or Section 9 for advanced details in steps 1 and 2.

7   How to use taylor by examples

The source code of examples in this manual is included in the **taylor** distribution. If **taylor** is installed from a `deb` or `rpm` package, they're in `/usr/share/taylor/`. If **taylor** is installed from source, they're in `src/examples` subdirectory.

7.1   A first taylor contact

Let `lorenz1.eq` be a four lines ASCII file that specifies the famous Lorenz equation.

File: `lorenz1.eq`

```
RR  = 28.0;
diff(x,t) = 10.0* (y - x);
diff(y,t) = RR * x - x*z - y;
diff(z,t) = x* y - 8.0* z /3.0;
```

After saving the file `lorenz1.eq`, let us ask **taylor** to generate a solver for us.

Single file output

The simplest method is to ask **taylor** to generate everything in a single file. The command

```
taylor -o lorenz1.c lorenz1.eq
```

produces a single file `lorenz1.c` ready to be compiled;

```
gcc lorenz1.c -lm
```

If we run the binary (`a.out`), the output looks like

```
    Enter Initial xx[0]: 0.03
    Enter Initial xx[1]: -0.02
    Enter Initial xx[2]: 0.15
    Enter start time: 0.0
    Enter stop time: 0.3
    Enter absolute error tolerance: 0.1e-16
    Enter relative error tolerance: 0.1e-16

0.03 -0.02 0.15 0
0.01836865  0.0061079  0.13463417  0.04051155
```

```
0.01802361   0.0269738   0.11958130   0.08501192
0.02707874   0.0550350   0.10484347   0.13448490
0.05034112   0.1086219   0.09053321   0.19018241
0.10579159   0.2304013   0.07720120   0.25359497
0.18285735   0.3986502   0.07016991   0.3
```

The output of `a.out` are the values of the state variables, in the order as they appear in the input file, plus the value of the time variable. For our last example, each row of the output are values of x, y, z, and t.

Standard use

A more common use of **taylor** is when one needs to write its own main driving procedure to call taylor stepper repeatedly, and perform other operations on the output data. Hence, we normally need to ask **taylor** to generate the header, the stepper, and the step control functions in separate files.

```
taylor -name lrnz -o lorenz1.c -jet -step lorenz1.eq
taylor -name lrnz -o taylor.h -header lorenz1.eq
```

The first line creates the file `lorenz1.c` (`-o` flag) with the code that computes the time-jet of derivatives (`-jet` flag) and the step size control (`-step` flag); the ODE description is read from the input file `lorenz1.eq`. The flag `-name` tells **taylor** the name we want to use for the stepper function; in this case the name is `taylor_step_lrnz` (the string after the `-name` flag is appended to the string `taylor_step_` to get the name of this function). The detailed description of the parameters of this function is in Section 9.2.

The second line produces a header file (named `taylor.h`) needed to compile `lorenz1.c`, that also contains the prototypes of the functions in `lorenz1.c` (this is the reason for using again the flag `-name`) so the user may also want to include it to have these calls properly declared. As we have not specified the kind of arithmetic we want, this header file will use the standard double precision of the computer.

Once we have the header and the stepper. We can write our main driving routine to call the stepper to compute the orbit –this is similar to the standard use of most numerical integrators, like Runge-Kutta or Adams-Bashford.

As an example, let us ask **taylor** to create a very simple main program for the Lorenz system,

```
taylor -name lrnz -o main_lrnz.c -main_only lorenz1.eq
```

Now we can compile and link these files,

```
gcc  main_lrnz.c lorenz1.c -lm
```

to produce an executable.

How to automatically generate a main program

The default `-main_only` flag generates a main program that asks us to input initial values at run time. We can specify the initial conditions, error tolerance, and stop conditions in the input file.

> *We stress that this information is only used to produce the main() driving function.*

The syntax for specifying initial values is:

```
initial_values = expr, expr, ..., expr;
```

In the Lorentz input file, it would be

```
initial_values = 2.03, 0.4, -0.5;
```

For time step, error tolerance, and stop conditions, **taylor** uses a few reserved variables (names). They are:

```
start_time = expr;                  /* start time */
stop_time = expr;                   /* stop time: stop condition  */
absolute_error_tolerance = expr; /* absolute error tolerance */
relative_error_tolerance = expr; /* relative error tolerance */
number_of_steps = expr;             /* stop condition */
```

Here the right hand expressions must reduce to real constants. The lines `stop_time` and `number_of_steps` provide two mechanisms to stop the integration. The solver will stop when either condition is met.

Please be advised that expressions here are evaluated to double precision values, then passed to the header macro `MakeMyFloatC(var,string_form,double_value)`. When a supported multi-precision is used, this macro uses a native conversion procedure to convert the number in string form to a native multi-precision object. **taylor** will pass numbers in it's original form (i.e, not converted to double) to this macro. Expressions, however, will be evaluated to double first, its string form is then generated use the `printf` utility.

For example, we can add the following lines to `lorenz.eq1`.

```
initial_values= 0.03, -0.02, 0.15;
start_time= 0.0;
stop_time = 0.3;
absolute_error_tolerance = 0.1e-16;
relative_error_tolerance = 0.1e-16;
```

7.2   Extended precision arithmetic

taylor has support for some extended precision arithmetic. For instance, assume we want to generate a Taylor integrator for the Lorenz example, using the GNU MPFR library.

```
taylor -mpfr -name lrnz -o lorenz1.c -jet -step lorenz1.eq
taylor -mpfr -name lrnz -o taylor.h  -header lorenz1.eq
```

The flag `-mpfr` instructs **taylor** to produce a header file with **mpfr** commands. As an example, we can ask **taylor** to generate a (very simple) main program for this case,

```
taylor -mpfr -name lrnz -o main_lrnz.c -main_only lorenz1.eq
```

We stress that the **mpfr** library is not included in the **taylor** package. In what follows, we assume that it is already installed in the computer and that **mpfr** library is somewhere in the default path used by your compiler to look for libraries, otherwise you will need to tell the compiler (`-L` flag for `gcc`) where to find that library. Thus, to compile and line the files, we use

```
gcc  main_lrnz.c lorenz1.c -lmpfr -lm
```

to produce an executable.

Important note: Extended precision libraries usually require some specific initializations that must be done by the main program. The subroutines produced by **taylor** will crash or produce wrong results if these initializations are not done properly. We strongly suggest you to read the documentation that comes with these libraries before using them.

7.3   Using External Parameters

This example demonstrate the use of `extern` variables, i.e., variable defined in somewhere other file that it must be find during the linkage process.
In some cases, a vector field can depend on one or several parameters and the user is interested in changing them at runtime. Moreover, for vector fields that depends on lots of constants, e.g. power or Fourier

expansions, it is desirable to have a separate procedure to read in those constants, rather than entering them by hand into the ODE definitions. **taylor** understands external variables and external arrays. It treats them as constants when computing the taylor coefficients.

`extern` variables are declared by the `extern` keyword. Listed below is a short example.

File: `perturbation.eq`

```
    extern MY_FLOAT e1, e2, coef[10], freq[10]; /* declare some external vars */

    diff(x,t) = e1 * y;
    diff(y,t) = -x + e2*sum( coef[i] * sin( freq[i] * t), i = 0, 9);
```

Let us save the above in `perturbation.eq`, and ask **taylor** to generate a solver for us.

```
    taylor -step -jet -o perturbation.c -name perturbation perturbation.eq
    taylor -name perturbation -header -o taylor.h perturbation.eq
```

We will have to write a driver for our integrator.

File: `main_params.c`

```
    #include "taylor.h"

    /* these are the variables the vector fields depends on */
    MY_FLOAT e1, e2, coef[10], freq[10];
    int main(void)
    {
      MY_FLOAT xx[2], t;
      double h, abs_err, rel_err, h_return;
      double log10abs_err, log10rel_err, endtime;
      int i, nsteps = 1000, order=10, direction=1;
      int step_ctrl_method=2;
      /* read in e1, e2, coef[] and freq[]
       * here, we just assign them to some
       * values
       */
      e1 = e2 = 1.0;
      for(i = 0; i < 10; i++) {
        coef[i] = 1.0;
        freq[i] = 0.1*(double) i;
      }
      /* set initial condition */
      xx[0] = 0.1;
      xx[1] = 0.2;
      t = 0.0;
      /* control parameters */
      h= 0.001;
      abs_err = 1.0e-16;
      rel_err = 1.0e-16;
      log10abs_err = log10(abs_err);
      log10rel_err = log10(rel_err);
      endtime = 10.0;
      /* integrate 100 steps */
      h_return = h; /* thanks to Jason James */
      while( -- nsteps > 0 && h_return != 0.0 ) {
        /* do something with xx and t. We just print it */
```

9

```
        printf("%f %f %f\n", xx[0],xx[1],t);
        taylor_step_perturbation(&t, &xx[0], direction,
                                 step_ctrl_method,log10abs_err, log10rel_err,
                                 &endtime, &h_return, &order,NULL);
    }
    return 0;
}
```

Note that the taylor steps require an initial time, the state variable, the direction of integration (forward
`+1` or backward `-1`), a step control method, log10 absolute and relative errors, and jet variables. Since
the original system does not require jet variables (see `perturbation.eq`), `main_params.c` sets to NULL
the last argument.

Now we can compile `perturbation.c` and `main_params.c` and run the executable.

```
gcc main_params.c perturbation.c -lm
./a.out
```

7.4   First order directional derivative

Let us continue with the Lorenz model. In this example, we will compute the positive maximal Lyaponov
exponent along the famous Lorenz attractor. Save the following text in `lorenz2.eq`.

File: `lorenz2.eq`

```
initialValues = 0.03,-0.02,0.15;
absoluteErrorTolerance = 1.0E-16;   /* error tolerance for step control */
relativeErrorTolerance = 1.0E-16;   /* error tolerance for step control */
stopTime = 10000;                   /* stop time */
startTime = 0.0;                    /* start time */


/* ODE specification:  lorenz */
RR = 28.0;
x' = 10.0* (y - x);
y' = RR* x - x*z - y;
z' = x* y - 8.0* z /3.0;


jet x,y,z symbols 1 deg 1;

jestartTialValues x ="(0.03   1 )";
jestartTialValues y ="(-0.02  0 )";
jestartTialValues z ="(0.15   0 )";
```

Here we declare all our variables as jet variables of degree 1 in 1 symbol. We also specify the initial values
for our jet variables. To generate a header file `lorenz2.h` and a stepper file `lorenz2.c`.

```
taylor -header -o lorenz2.h lorenz2.eq
taylor -header_name lorenz2.h -jet -jhelper -step -o lorenz2.c lorenz2.eq
```

Notice the `-jhelper` flag tells **taylor** to generate a few jet IO helper functions.

We need a main driving function. The following is modified from the driver generated by **taylor**.

File: `main_lyap.c`

```
#include "lorenz2.h"

int main(int argc, char *argv[])
```

```
  {
    int     i, j, order=20, itmp=0, direction = 1, nsteps = -1, counter=0;
    double  dstep, rtolerance, log10abs=-16, log10rel=-16;
    double  startT, stopT, nextT;
    double  xx[4], yy[4], zz[4], **jet;
    double  lyap=0.0, norm;

    MY_JET    *jetOut, jetIn[4];

    taylor_initialize_jet_library();
    for(i=0; i < 3; i++) InitJet(jetIn[i]);

    /* initialize jet vars --start */
    InputJetFromString(jetIn[0],"(0.03  1 )");
    InputJetFromString(jetIn[1],"(-0.02 0 )");
    InputJetFromString(jetIn[2],"(0.15  0 )");
    /* initialize jet vars --end */

    stopT = 10000;
    startT = 0;
    dstep=0.001;

    while(1)  {
      if(itmp != 0) {break;}
      if(startT >= stopT) { break;}
      itmp = taylor_step_lorenz2_eq(&startT, xx, direction, 1,
                                    log10abs, log10rel,
                                    &stopT, &nextT, &order, jetIn);

      if(++counter >= 1000) {  // we normalize the jet every 1000 steps
        norm = 0;
        for(i=0; i < 3; i++)  norm += MY_JET_DATA(jetIn[i],1) * MY_JET_DATA(jetIn[i],1);

        norm = sqrt(norm);
        lyap += log(norm);

        for(i=0; i < 3; i++) MY_JET_DATA(jetIn[i], 1) /= norm;
        counter = 0;
      }
    }  /* while */
    if(counter > 0) {
      for(i=0; i < 3; i++) norm += MY_JET_DATA(jetIn[i], 1) *  MY_JET_DATA(jetIn[i], 1);
      norm = sqrt(norm);
      lyap += log(norm);
    }

    lyap /= 10000;
    fprintf(stdout, "The estimated Lyaponov exponent is: %f\n", lyap);

    exit(0);
  }
```

Now we can compile and run the code.

```
gcc main_lyap.c lorenz2.c -lm
./a.out
```

After a few seconds, it should report a result similar to the following.

```
The estimated Lyapunov exponent is: 9.055203
```

7.5   First order variational derivative

Following the Lorenz example, let us use **taylor** to solve the first variational equations. We also add a trivial equation to include the variation with respect to a parameter. `lorenz3.eq` contains our Lorenz model, with an extra equation for the parameter `RR`. The last line tells **taylor** to generate code to compute the first variational flow. It declares the jet variables (or using the keyword `all`), the number of symbols (4) and the degree of jet (1).

File: `lorenz3.eq`

```
x' = 10.0* (y - x);
y' = RR* x - x*z - y;
z' = x* y - 8.0* z /3.0;
RR'= 0;

jet x,y,z,RR symbols 4 deg 1;
```

To obtain the header and source of the ODE in `lorenz3.eq` we can do

```
taylor -o lorenz3.h -header lorenz3.eq -mpfr
taylor -o lorenz3.c -jet -step -headername lorenz3.h lorenz3.eq -jethelper -mpfr
```

Note that we used the flag `-mpfr` to indicate that we want to use the `mpfr` multi-precision floating arithmetic. Note also that we used the flag `-name` to set a name of the ODE system, and used `-jethelper` flag to include jet IO helper functions.

The following `main_varieq.c` shows a main driving function. It is arithmetic indepedent. In other words, it works with code generated by **taylor** with or without the `-mpfr` flag.

File: `main_varieq.c`

```
#include "lorenz3.h"

#define NS _NUMBER_OF_STATE_VARS_
#define NJ _NUMBER_OF_JET_VARS_

#ifdef _USE_MPFR_
#define DIGITS_PRECISION 35
#define STR(x) #x
#define STR1(x) STR(x)
#define JFMT "% ." STR1(DIGITS_PRECISION) "RNE"
#else
#define DIGITS_PRECISION 16
#define JFMT "% .14e"
#endif

int main(void)
{
  int i, order, direction=+1, step_cntrl=2;
  double log10abs=-DIGITS_PRECISION, log10rel=-DIGITS_PRECISION;
```

```
      MY_FLOAT startT, nextT, stopT, x[NS];
      MY_JET xjet[NJ];

   #ifdef _USE_MPFR_
      mpfr_set_default_prec((int)(DIGITS_PRECISION*log2(10))+1);
   #endif

      taylor_initialize_jet_library();

      InitMyFloat(startT); InitMyFloat(nextT); InitMyFloat(stopT);
      for (i = 0; i < NS; i++) {InitMyFloat(x[i]);}
      for (i = 0; i < NJ; i++) {taylor_initialize_jet_variable(&xjet[i]);}

      MakeMyFloatC(x[0],"0",0);
      MakeMyFloatC(x[1],"1",1);
      MakeMyFloatC(x[2],"0",0);
      MakeMyFloatC(x[3],"28",28); /* parameter RR */

      taylor_make_identity_jets(xjet,x,NULL);

      MakeMyFloatC(startT,"0",0);
      MakeMyFloatC(stopT,"1",1);
      while (taylor_step_lorenz3_eq(&startT, x, direction,
                                    step_cntrl, log10abs, log10rel,
                                    &stopT, &nextT, &order, xjet) != 1) {}

      for (i = 0; i < NJ; i++) {
        printf("x%d=\n",i);
        taylor_output_jet(stdout, JFMT "\n", xjet[i]);
      }

      /* cleaning memory */
      for (i = 0; i < NJ; i++) {taylor_clear_jet_variable(&xjet[i]);}
      for (i = 0; i < NS; i++)  {ClearMyFloat(x[i]);}
      ClearMyFloat(stopT); ClearMyFloat(nextT); ClearMyFloat(startT);
      taylor_clear_up_jet_library();
      return 0;
   }
```

Finally, we can compile and run the code.

```
gcc main_varieq.c lorenz3.c -lgmp -lmpfr -lm
./a.out
```

8 How to become a taylor advanced user by examples

8.1 OpenMP compatibility

taylor is openMP compatible provided that the jet variables initialization are done properly in the driving routine.

If thread-dependent parameters are used in the ODE system, the user is responsible to:

1.) add the `#pragma omp threadprivate(param_name)` in the driving routine, and

2.) append `#pragma omp threadprivate(param_name)` after the extern variable in the generated source file.

We illustrate OpenMP use with an example. Save the following ODE spec in `lorenz4.eq`.

File: `lorenz4.eq`

```
    extern MY_FLOAT RR,SS;
    x' = SS* (y - x);
    y' = RR* x - x*z - y;
    z' = x* y - 8.0* z /3.0;

    jet x,y,z symbols 3 deg 5;
```

We then generate the source and header files using **taylor**,

```
    taylor -o lorenz4.h -header lorenz4.eq
    taylor -o lorenz4.c -jet -step -headername lorenz4.h lorenz4.eq -jethelper
```

We assume parameter `RR` is thread-dependent, so we need to modify `lorenz4.c`. Append the line

```
    extern MY_FLOAT RR;
```

with

```
    #pragma omp threadprivate(RR)
```

A driving file is provided in `main_omp.c`. Notice that under the assumption that `RR` is thread-dependent, the same `pragma` instruction is also added.

File: `main_omp.c`

```
    #include <omp.h>
    #include "lorenz4.h"
    #define NJ _NUMBER_OF_JET_VARS_
    #define NS _NUMBER_OF_STATE_VARS_
    MY_FLOAT RR, SS;
    #pragma omp threadprivate(RR)

    int main(int argc, char *argv[])
    {
      int k, j, ord;
      const int np = omp_get_max_threads();
      MY_FLOAT x[NS*np], te;
      MY_JET xjets[NJ*np];

      printf("np=%d threads\n", np);
    #pragma omp parallel
      taylor_initialize_jet_library();

      /* memory allocation */
      InitMyFloat(te); InitMyFloat(RR); InitMyFloat(SS);
      for (k = 0; k < NS*np; k++) {InitMyFloat(x[k]);}
      for (k = 0; k < NJ*np; k++) {InitJet(xjets[k]);}

      /* some initializations */
      MakeMyFloatC(SS,"10",10);
      MakeMyFloatC(te,"1",1);
```

```
    /* jets with identity matrix at first order */
    for (j = 0; j < np; j++) {taylor_make_identity_jets(xjets+NJ*j,x+NS*j,NULL);}

  #pragma omp parallel private(ord,k)
    {
      MY_FLOAT t,ht;
      InitMyFloat(t); InitMyFloat(ht);

      int tid = omp_get_thread_num();
      printf("tid=%d\n", tid);

      MakeMyFloatA(RR,tid+1);
      for (k = 0; k < NS; k++) {MakeMyFloatA(x[NS*tid+k],1/(tid+1));}

      MakeMyFloatC(t,"0",0);
      while (taylor_step_lorenz5_eq(&t, x+NS*tid, +1, 2, -16, -16,
                                    &te, &ht, &ord, xjets+NJ*tid) != 1) {}

      ClearMyFloat(ht); ClearMyFloat(t);
    } /* end parallel region */

    /* free memory */
    for (k = 0; k < NJ*np; k++) {ClearJet(xjets[k]);}
    for (k = 0; k < NS*np; k++) {ClearMyFloat(x[k]);}
    ClearMyFloat(SS); ClearMyFloat(RR); ClearMyFloat(te);
  #pragma omp parallel
    ClearUpJet();
    return 0;
  }
```
Finally, we can compile and run the code

```
gcc main_omp.c lorenz4.c -lm -fopenmp
./a.out
```

8.2 Stability of equilibria

taylor can be used to study stability of equilibrium points by computing its Jacobian and even bifurcations by computing high-order derivatives.

Let us create the input ODE file `lorenz5.eq` that codifies the famous Lorenz model.

<div align="center">File: <code>lorenz5.eq</code></div>

```
extern MY_FLOAT RR;
x' = 10.0* (y - x);
y' = RR* x - x*z - y;
z' = x* y - 8.0* z /3.0;


jet x,y,z symbols 3 deg 1;
```

and let us generate the source and header files using **taylor**,

```
taylor -o lorenz5.h -header lorenz5.eq
taylor -o lorenz5.c -jet -step -headername lorenz5.h lorenz5.eq -jethelper
```

The driving routine in `main_equilibria.c` consists in assigning to the state variable the equilibrium point and to the jet variables the equilibrium points plus the identity matrix. After that, the evaluation of the vector field gives us the differential at the equilibrium point and from there one can compute the eigenvalues using some linear algebra software.

Note that by increasing the degree, one can obtain higher derivatives at the equilibrium point that are commonly required for bifurcation studies.

File: `main_equilibria.c`

```
#include "lorenz5.h"

#define NJ _NUMBER_OF_JET_VARS_
#define NS _NUMBER_OF_STATE_VARS_
MY_FLOAT RR;

int main(int argc, char *argv[])
{
  int k, j;
  MY_FLOAT x[NS], t, A[NS*NS], *dtmp;
  MY_JET jetIn[NJ], **jetOut;

  taylor_initialize_jet_library();

  /* memory allocation */
  InitMyFloat(t);
  InitMyFloat(RR);
  for (k = 0; k < NS; k++) {InitMyFloat(x[k]);}
  for (k = 0; k < NJ; k++) {InitJet(jetIn[k]);}
  for (k = 0; k < NS*NS; k++) {InitMyFloat(A[k]);}

  /* equilibrium point */
  MakeMyFloatC(t,"0",0);
  MakeMyFloatC(RR,"28",28);
  for (k = 0; k < NS; k++) {MakeMyFloatC(x[k],"0",0);}

  /* jets with identity matrix at first order */
  taylor_make_identity_jets(jetIn,x,NULL);

  /* vector field evaluation */
  taylor_coefficients_lorenz5_eq_A(t, x, 1, 0, jetIn, &jetOut);

  for (k = 0; k < NS; k++)
  {
    /* coefficients without the state variable */
    dtmp = taylor_convert_jet_to_array(jetOut[k][1],0);

    /* save the value in the matrix A*/
    for (j = 0; j < NS; j++) A[k*NS + j] = dtmp[j];
  }

  /* print the differential at the equilibrium point */
  for (k = 0; k < NS*NS; k++)
  {
    OutputMyFloat3(stdout,"% .5e ", A[k]);
```

```
      if ((k+1) % NS==0) printf("\n");
    }

    /* free memory */
    for (k = 0; k < NS*NS; k++) {ClearMyFloat(A[k]);}
    for (k = 0; k < NJ; k++) {ClearJet(jetIn[k]);}
    for (k = 0; k < NS; k++) {ClearMyFloat(x[k]);}
    ClearMyFloat(RR); ClearMyFloat(t);
    ClearUpJet();
    return 0;
  }
```

9   Extra details for guru users

This section is intended to provide extra details that an experienced user may need, such as, **taylor** command line flags, output routines, data types, API functions, stepsize controls, and driving routines.

9.1   Command Line Options

taylor supports the following command line options.

```
Usage: ./taylor
  [-name ODE_NAME]
  [-o outfile]
  [-long_double | -float128 | -mpfr | -mpfr_precision PRECISION ]
  [-main | -header | -jet | -jhelper | -jet_helper | -main_only]
  [-step STEP_CONTROL_METHOD]
  [-jlib JET_LIBRARY]
  [-u | -userdefined] STEP_SIZE_FUNCTION_NAME ORDER_FUNCTION_NAME
  [-sqrt]
  [-headername HEADER_FILE_NAME]
  [-debug] [-help] [-v]  file
```

Let us explain them in detail.

- `-name ODE_NAME`

  This option specifies a name for the system of ODEs. The output functions will have the specified name appended. For example, if we run **taylor** with the option `-name lorenz`, the output procedures will be `taylor_step_lorenz` and `taylor_coefficients_lorenz`. If name is not specified, **taylor** appends the input filename (with non-alpha-numeric characters replaced by `_`) to its output procedure names. In the case when input is the standard input, the word `_NoName` will be used.

- `-o outfile`

  This option specifies an output file. If not specified, **taylor** writes its output to the standard output.

- `-mpfr`

  This option tells **taylor** to use the the GNU MPFR library.

- `-mpfr_precision PRECISION`

  This flag is almost equivalent to `-mpfr`; the only difference is when a main() program is generated. If `-mpfr` is used the main program asks, at runtime, for the lenght (in bits) of the mantissa of the **mpfr** floating point types. If `-mpfr_precision PRECISION` is used, the main program will set the precision to `PRECISION` without prompting the user.

- `-float128`

  This option tells **taylor** to use the GNU C `__float128` floating point arithmetic.

- `-main`

  Informs **taylor** to generate a very simple `main()` driving routine. This option is equivalent to the options `-main_only -jet -step 1`, so it produces a "ready-to-run" C file.

- `-header`

  This option tells **taylor** to output the header file. The header file contains the definition of the `MY_FLOAT` type (the type used to declare real variables), macro definitions for arithmetic operations and elementary mathematical function calls. In other words, this file header file is responsible for the kind of arithmetic used for the numerical integration. Hence, the flag `-header` must be combined with one of the flags `-mpfr`, `-float128` to produce a header file for the corresponding arithmetic. If none of these flags is specified, the standard double precision arithmetic will be used.

  Moreover, if the flag `-name ODE_NAME` is also used, the header file will also contain the prototypes for the main functions of the Taylor integrator.

- `-jet`

  This option asks **taylor** to generate only the code that computes the taylor coefficients. The generated routine is

  ```
  MY_FLOAT **taylor_coefficients_ODE_NAME(
      MY_FLOAT t,  /* input: value of the time variable    */
      MY_FLOAT *x, /* input: value of the state variables  */
      int    order /* input: order of the taylor polynomial */
      )
  ```

  The code needs a header file (defining the macros for the arithmetic) in order to be compiled into object code. The default header filename is `taylor.h`. The header filename can be changed using `-headername NAME` (see below). You can also use the `-header` option to include the necessary macros in the output file.

- `-jhelper`

  This option asks **taylor** to include jet IO helper functions in the output. When not combined with other output options, **taylor** will only output the helper functions. For example,

  ```
   taylor -o helper.c -headername lorenz.h -jhelper lorenz.eq
  ```

  will save the helper functions in `helper.c`. Please be advised that

  > *the IO Helper functions should be generated only once per application.*

  Otherwise, the compiler will complain about multiple defined IO helpers. If you separate **taylor** generated code in files, it is a good idea to keep the IO Helpers in its own file.

- `-jlib JETLIBRARY`

  This option select a jet library. By default, **taylor** uses the `jet_tree` library. This option allows you to overwrite that with a special purpose library. Possible value for `JETLIBRARY` are:

  - `jet1_1` one symbol, degree one. The arithmetic is implemented using C macros.
  - `jet1` one symbol, arbitrary degree.
  - `jet_1` degree one, arbitrary number of symbols.
  - `jet2` two symbols, arbitrary degree.

- **jet_2** degree two, arbitrary number of symbols.

- **jet_m** an naive implementation of the general case, arbitrary number of symbols, arbitrary degree. This implementation works well when the jet size is below 1000 terms, i.e., when $\binom{d+m}{d} < 1000$.

- **jet_tree** a general library[2]. This is the default.

- **-main_only**

  This option asks **taylor** to generate only the `main()` driving routine. It is useful when you want to separate different modules in different files. The main driving routine has to be linked with the step size control procedure and the jet derivative procedure to run.

- **-step STEP_SIZE_CONTROL_METHOD**

  This option asks **taylor** to generate only the order and step size control code supplied by the package. If combined with the **-main** or **-main_only** flags, the value `STEP_SIZE_CONTROL_METHOD` is used in the main program to specify the step size control. The values of `STEP_SIZE_CONTROL_METHOD` can be 0 (fixed step and degree), 1, 2, and -1 (user defined step size control; in this case you have to code your own step size and degree control). If the flags **-main** and **-main_only** are not used, this value is ignored.

  The generated procedure is also the main call to the numerical integrator:

  ```
  int taylor_step_ODE_NAME(MY_FLOAT *time,
                           MY_FLOAT *xvars,
                           int      direction,
                           int      step_ctrl_method,
                           double   log10abserr,
                           double   log10relerr,
                           MY_FLOAT *endtime,
                           MY_FLOAT *stepused,
                           int      *order,
                           MY_JET   *jetInOut)
  ```

  This code needs the header file to be compiled (see the remarks above). Given an initial condition (`time`,`xvars`,`jetInOut`), this function computes a new point on the corresponding orbit with the jet variables transported. The meaning of the parameters is explained in Section 9.2.

- **-userdefined STEP_SIZE_FUNCTION_NAME   ORDER_FUNCTION_NAME**

  This flag specifies the names of your own step size and order control functions. Then, the code produced with the flag **-step** includes the calls to your control functions; to use them, you must set `step_ctrl_method` to 3 (see Section 9.2).

  For more details (like the parameters for these control functions) look at the source code produced by the **-step** flag.

- **-sqrt**

  This option tells **taylor** to use the function `sqrt` instead of `pow` when evaluating terms like $(x+y)^{-\frac{3}{2}}$. The use of `sqrt` instead of `pow` produces code that runs faster.

- **-headername HEADER_FILE_NAME**

  When **taylor** generates the code for the jet and/or step size control, it assumes that the header file will be naed `taylor.h`. This flag forces **taylor** to change the name of the file to be included by the jet and/or step size control procedures to the new name `HEADER_FILE_NAME`. Of course, the user is then responsible for creating such a header file by combining the flags **-o HEADER_FILE_NAME** and **-header**. For instance,

---

[2]based on `http://www.maia.ub.es/dsg/param/chapter2.html`

```
taylor -name lz -o l.c -jet -step -headername l.h lorenz.eq1
```

stores the code for the jet of derivatives and step size control in the file `l.c`. Moreover, `l.c` includes the header file `l.h`. This file has to be created separately:

```
taylor -name lz -o l.h -header lorenz.eq1
```

- `-debug` or `-v`

  Print some debug info to `stderr`.

- `-help` (or `-h`)

  Print a short help message.

The default options are set to produce a full C program, using the standard double precision of the computer:

```
-main_only -header -jet -step 1
```

9.2 The Output Routines

taylor outputs two main procedures. The first one is the main call for the integrator and the second one is a function that computes the jet of derivatives. For details on some other routines generated by **taylor** (like degree or step size control), see the comments in the generated source code.

The numerical integrator

Its prototype is:

```
int taylor_step_ODE_NAME(MY_FLOAT *time,
                         MY_FLOAT *xvars,
                         int      direction,
                         int      step_ctrl_method,
                         double   log10abserr,
                         double   log10relerr,
                         MY_FLOAT *endtime,
                         MY_FLOAT *stepused,
                         int      *order,
                         MY_JET   *jetOut);
```

The function `taylor_step_ODE_NAME` does one step of numerical integration of the given system of ODEs, using the control parameters passed to it. It returns 1 if `endtime` is reached, 0 otherwise.

Parameters:

- `time`
  on input: time of the initial condition
  on output: new time

- `xvars`
  on input: initial condition
  on output: new condition, corresponding to the (output) time

- `direction`
  flag to integrate forward or backwards.
      1: forward
     -1: backwards
  Note: this flag is ignored if `step_ctrl_method` is set to 0.

- **step_ctrl_method**
flag for the step size control. Its possible values are:

  0: no step size control, so the step and order are provided by the user. The parameter `stepused` is used as step, and the parameter `order` (see below) is used as the order.

  1: standard stepsize control. It uses an approximation to the optimal order and to the radius of convergence of the series to approximate the "optimal" step size. It tries to keep the absolute and relative errors below the given values. See the paper [1] for more details.

  2: as 1, but adding an extra condition on the stepsize h: the terms of the series – after being multiplied by the suitable power of h – cannot grow.

  -1: user defined stepsize control. The code has to be included in the source routine called `compute_timestep_user_defined` (see the code). The user must also include code for the selection of degree, in the function `compute_order_user_defined`.

- **log10abserr**
decimal log of the absolute accuracy required.

- **log10relerr**
decimal log of the relative accuracy required.

- **endtime**
if `NULL`, it is ignored.
if `step_ctrl_method` is set to 0, it is also ignored.
otherwise, if next step is going to be outside `endtime`, reduce the step size so that the new time `time` is exactly `endtime` (in that case, the function returns 1).

- **ht**
on input: ignored/used as a time step (see parameter `step_ctl_method`)
on output: time step used

- **order**

  input: this parameter is only used if `step_ctrl_method` is 0, or if you add the proper code for the case `step_ctrl_method=3`.
  If `step_ctrl_method` is 0, its possible values are:
  < 2: the program will select degree 2,
  ≥ 2: the program will use this degree.

  output: degree used.

- **jetInOut**

  on input: the value of jet variables

  on output: the new value of jet variable at the output time `ti`.

Returned value:

- 0: ok.

- 1: ok, and `time=endtime`.

The jet of derivatives

Its prototype is

```
MY_FLOAT **taylor_coefficients_ODE_NAME(MY_FLOAT t,
                                        MY_FLOAT *x,
                                        int order);
```

`taylor_coefficients_ODE_NAME` returns a **static** two dimensional array. The rows are the Taylor coefficients of the state variables.

Parameters

- `t`: value of the time variable. It is used only when the system of ODEs is nonautonomous.

- `x`: value of the state variables.

- `order`: degree of Taylor polynomial.

If you want to compute several jets at the same point but with increasing orders, then you should consider using the call

```
MY_FLOAT **taylor_coefficients_ODE_NAMEA(MY_FLOAT t,
                                         MY_FLOAT *x,
                                         int order,
                                         int rflag,
                                         MY_JET *jetIn,
                                         MY_JET ***jetOut)
```

(note the "`A`" at the end of the name). The first three parameters have the same meaning as before, and the meaning of the fourth one is:

0: the jet is computed from order 1 to order `order`.

1: the jet is computed starting from the final order of the last call, up to `order`.

Care must be exercised if you invoke this routine with `rflag=1`. Indeed, if you modify the Taylor coefficients and/or the base point, you need to restore them before the next call.

9.3 The data type `MY_FLOAT`

`MY_FLOAT` is a customizable floating point data type. Arithmetic on `MY_FLOAT` is defined through a set of C macros. **taylor** currently supports the following floating point data types.

- `double`. The standard C `double`. This is the default.

- `long double`. The C `long double` type.

- `__float128`. The GNU C 128 bit floating point type[3].

- `mpfr_t`. An arbitary precision floating-point data type from `mpfr` library[4].

- `gmp_t`. An arbitrary precision floating point data type from `gmp` library[5].

These data types can be selected using **taylor** command line flags, that is, `-long_double`, `-float128`, `-gmp` or `-gmp_precision NBITS` and `-mpfr` or `-mpfr_precision NBITS`, see Section 9.1.

To extend **taylor** to support a custom floating point data type, one needs to redefine the set of C macros **taylor** calls for. See `mpfrheader.h` in the source code for reference.

9.4 The data type `MY_JET`

`MY_JET` is a new data type that encodes truncated multivariate polynomial through an array of monomial coefficients. The exact implementation is library dependent and hence is opaque to the end user. In all implementations, monomial coefficients are stored as an array of `MY_SCAL`s. A C macro `MY_JET_DATA(jet,i)` is provided to access the ith monomial coefficient, arranged in "graded lexicographical order". In the current version of **taylor**, `MY_SCAL` is just an alias of `MY_FLOAT`. This may change in future releases.

---

[3]`https://gcc.gnu.org/onlinedocs/libquadmath/index.html`
[4]`https://www.mpfr.org`
[5]`https://gmplib.org`

API functions related to `MY_JET`

For your convenience, **taylor** includes a few API functions pertained to Jet Transport. Most of these functions are IO helpers. These functions will be generated when the `-jhelper` command line option is passed to **taylor**.

The API functions use size related constants or global variables generated by **taylor**. Those variables are computed from the number of symbols and the degree from jet declaration line in ode specification. It is assumed that those variables stay unchanged in an application. Thus, if you use **taylor** to integrate multiple ODEs in the same program, the jet must be declared the same way in all, i.e, use the same degree and number of symbols in all ODE specifications.

Some of the API functions take both `MY_FLOAT` and `double` arguments to set values of a `MY_JET` variable. For high accuracy computations, it is advised that you use the `MY_FLOAT` argument, as the conversion of double to `MY_FLOAT` is not precise.

- `int taylor_make_jet(MY_JET a, MY_FLOAT *myfloats, double *values)`
  This function fills in the data array of a `MY_JET` with either `myfloats` or `values`. Caller is responsible to make sure there are enough values in `myfloats` or `values`. It should include the coefficients of all monomials for the current model, including the constant term.

- `int taylor_make_identity_jets(MY_JET *inOut, MY_FLOAT *myfloats, double *values)`
  This function fills the array `inOut` with an identity matrix. If either `myfloats` or `values` is not NULL, it fills the constant terms in `inOut` with elements in `myfloats` or `values`. This function uses the global variable `_number_of_jet_vars_`, `_number_of_symbols_` and `_size_of_jet_variable_`. Please note, this function only fills the first `_number_of_symbols_` or `_number_of_jet_vars` rows of `inOut`, whichever is smaller. The constant terms are filled in full, if provided.

- `int taylor_make_unit_jet(MY_JET a, int idx, MY_FLOAT *myfloat, double *value)`
  This function assigns 1 to the coefficient of the `idx`th symbol in `a`, assign 0 to other coefficients. If either `myfloat` or `value` is not NULL, its value is assigned to the constant term in `a`.

- `int taylor_set_jet(MY_JET a, MY_FLOAT *myfloats, double *values, int include_state)`
  This function fills in the jet variable a with values supplied in `myfloats` or `values`. `include_state` signals if the constant term is included in `myfloats` or `values`. If not, 0 will be assigned to the constant term in a.

- `int taylor_input_jet_from_stdin(MY_JET a, int idx)`
  This function asks the user to fill in a jet variable from stdin. `idx` signals which state variable `a` is associated with. This function prints out a prompt that includes all monomials.

- `int taylor_input_jet_from_string(MY_JET a, const char *str)`
  This function fills in the jet variable `a` with values from a string. It is an easy method to fill in a jet variable. For example, `InputJetFromString(jetIn[0],"(0.03 1 0 0 0 )");`.

- `int taylor_output_jet(FILE *file, char *fmt, MY_JET a)`
  This function output a jet variable to `file`, using the supplied format to format each element. Please note, some `MY_FLOAT` type may require special flags in the format string. For example, `long double` requires a L specifier, like `%.18Lf`.

- `void taylor_initialize_jet_library()`
  This function initializes the JET library.

- `void taylor_initialize_jet_variable(MY_JET *jet)`
  This function initializes the jet variable `jet`. All jet variables need to be initialized before use.

- `void taylor_clear_jet_variable(MY_JET *jet)`
  This function clears the storage space allocated for a jet variable `jet`.

- `int taylor_set_jet_variable_degree(int d)`
  This function sets the working degree of all jet variables. The degree must be smaller than the maximum degree set in the ODE specification. This function returns the current working degree.

- `void taylor_jet_reduce(MY_JET a, double *v)`
  This function reduces a jet variable `a` to a number, using the values in `v` to substitute the symbols in the corresponding position.

- `const char **taylor_get_variable_names()`
  This function returns the list of names for all state variables, in the order as they appear in the model specification.

- `const char **taylor_get_jet_monomials()`
  This function returns the list of monomials in 'graded lexicographic' order for the current model. Because generating the list of monomials is an expensive operation. This function will return an empty list when the total number of monomials exceeds 1000.

Accessing elements in `MY_JET`

Internally, `MY_JET` stores monomial coefficients in an array of `MY_FLOAT`s. The order of the coefficients differs by implementation. A macro `MY_JET_DATA(jet,idx)` is provided for accessing these coefficients in "graded lexicographic" order. That is, the monomials are ordered by total degree first, monomials of the same total degree are ordered lexicographically. For example, for degree 2 with 3 symbols, the "graded lexicographic" order of the monomials are

$$1, s_1, s_2, s_3, s_1^2, s_1 s_2, s_1 s_3, s_2^2, s_2 s_3, s_3^2$$

`MY_JET_DATA(jet,6)` refers the coefficient of $s_1 s_3$. This macro can be used as lvaue, i.e., to assign/change values of the monomial coefficients.

The following example demonstrates the use `MY_JET_DATA`.

Save the following ODE specification in `model.eq`

<div align="center">File: <code>model.eq</code></div>

```
x' = -y + x *( 1-x*x -y*y);
y' =  x + y *( 1-x*x -y*y);
r' = r*(1-r*r);

jet x,y  symbols 2 deg 2;
```

and save the following code to `jdata_main.c`.

<div align="center">File: <code>jdata_main.c</code></div>

```
#include "jdata.h"

/**
 *  This example demostrates how to access MY_JET storage array directly.
 */

#define NN  _NUMBER_OF_STATE_VARS_
#define JJ  _NUMBER_OF_JET_VARS_

double initial_values[] = {0.03,-0.02,0.15,0.1,-0.1,0.1,0.2,-0.2,0.2,0.1};
double T0 = 0.0, T1 = 0.2;
```

```c
int main(int argc, char *argv[])
{
    int      i, j, order=20, itmp=0, direction = 1, nsteps = -1;
    double   dstep, log10abs=-16, log10rel=-16;
    MY_FLOAT startT, stopT, nextT, xx[NN];
    MY_JET   jetIn[JJ];

#ifdef _USE_MPFR_
    mpfr_set_default_prec(256);
#endif
#ifdef _USE_GMP_
    mpf_set_default_prec(256);
#endif
    taylor_initialize_jet_library();

    InitMyFloat(startT);
    InitMyFloat(stopT);
    InitMyFloat(nextT);

    for(i = 0; i<NN; i++) {InitMyFloat(xx[i]);}
    for(i = 0; i<JJ; i++) {InitJet(jetIn[i]);}


    dstep=0.001; /* only nedeed when step_ctrl_method==0 (see manual) */

    taylor_make_identity_jets(jetIn, NULL, NULL);

    MakeMyFloatA(stopT, T1);
    MakeMyFloatA(startT, T0);
    order=20;
    itmp = 0;

    for(i =0; i< NN; i++) {
      MakeMyFloatA(xx[i],  initial_values[i]);
      // here we assign value to the constant term in jetIn[i]
      AssignMyFloat(MY_JET_DATA(jetIn[i], 0),  xx[i]);
    }

    // print monomial names
    {
      char **monomials = taylor_get_jet_monomials();
      fprintf(stdout,"%-18s", "      1");
      i = 0;
      while(monomials[i] != NULL) {
        fprintf(stdout, "%-13s", monomials[i]);
        i++;
      }
      fprintf(stdout,"\n");
    }

    while(1)  {
      for(i = 0; i < JJ; i++) {
        char **var_names = taylor_get_variable_names();
```

```
                // here we print all monomial coefficients
                fprintf(stdout, "%s: ", var_names[i]);
                for(j = 0; j < _SIZE_OF_JET_VAR_; j++) {
    #ifdef _USE_MPFR_
                    fprintf(stdout, "%12.8f ", mpfr_get_d(MY_JET_DATA(jetIn[i],j),GMP_RNDN));
    #else
    #ifdef _USE_GMP_
                    fprintf(stdout, "%12.8f ", mpf_get_d(MY_JET_DATA(jetIn[i],j),GMP_RNDN));
    #else
                    fprintf(stdout, "%12.8f ", MY_JET_DATA(jetIn[i],j));
    #endif
    #endif
                }
                fprintf(stdout, "\n");
            }
            fprintf(stdout, "\n");

            if(itmp != 0) {break;}
            if(MyFloatA_GE_B(startT,stopT)) { break;}

            itmp = taylor_step_jdata( &startT, xx, direction, 1,
                                      log10abs, log10rel,
                                      &stopT, &nextT, &order, jetIn);

        }  /* while */

        exit(0);
    }
```

Now generate the required header `jdata.h` and a taylor stepper `jdata.c`.

```
taylor -header -name jdata -o jdata.h model.eq
taylor -jet -jhelper -name jdata -step -header_name jdata.h model.eq -o jdata.c
```

Finally compile the code and run

```
cc -g jdata.c jdata_main.c -lm
./a.out
```

You'll get a nice printout of elements in `MY_JET` like the following.

```
       1            s1            s2           s1^2          s1s2         s2^2
x:   0.03000000    1.00000000    0.00000000    0.00000000    0.00000000   0.00000000
y:  -0.02000000    0.00000000    1.00000000    0.00000000    0.00000000   0.00000000

x:   0.04075176    1.19607263   -0.24217713   -0.02764699    0.01532271  -0.01239364
y:  -0.01665614    0.24282333    1.19650979    0.00051102   -0.01525349   0.01585447
```

9.5 Write a Driving Routine

The main driving routine produced by the **-main** flag of **taylor** is rather simple, it just keeps on integrating the system and print out the solution along the way. This may be enough for some tasks, but it is definitely too primitive for real applications. In this section, we provide two sample driving routines. These examples demonstrate what you need to do to write your own driving routes. The input files are provided in the doc subdirectory in the **taylor** distribution.

We first ask **taylor** to generate a integrator and a header file for us.

```
taylor -o lorenz.c -jet -step -name lorenz lorenz1.eq
taylor -o taylor.h -header lorenz1.eq
```

The first command will produce a file `lorenz.c` with no driving routine in it. This file will be compiled and linked with our main driving routine. The second command generates the header file `taylor.h`. It is needed in `lorenz.c` and our main driving function.

Using the Supplied Integrator

Our first example is very similar to the driving routine generated by **taylor**. It uses the one step integrator provided by **taylor**.

File `main1.c`

```
#include <stdio.h>
#include <math.h>
#include "taylor.h"
int main(int argc, char *argv[])
{
    MY_FLOAT  xx[3], t;
    double    h, h_return, log10abs_err, log10rel_err, endtime;
    int       nsteps = 100, step_ctrl_method = 2, direction = 1;
    int       order = 10;
    /* set initial conditions */
    xx[0] = 0.1;
    xx[1] = 0.2;
    xx[2] = 0.3;
    t     = 0.0;
    /* control parameters       */
    h= 0.001;
    log10abs_err = -16 /* i.e. 1.0e-16 absolute error */
    log10rel_err = -16 /* i.e. 1.0e-16 relative error */
    endtime = 10.0;

    /* integrate 100 steps */
    while( -- nsteps > 0 && h_return != 0) {
        /* do something with xx and t. We just print it */
        printf("%f %f %f %f\n", xx[0],xx[1],xx[2],t);
        taylor_step_lorenz(&t, &xx[0], direction,
                           step_ctrl_method,log10abs_err, log10rel_err,
                           &endtime, &h_return, &order);
    }
    return 0;
}
```

After saving the code in `main1.c`, you can compile them using the command

```
gcc lorenz.c main1.c -lm
```

and run the executable `a.out` as before.

Writing Your Own Driver

This example provides a skeleton for writing your own one step integrator.

File `main2.c`

```c
#include <stdio.h>
#include <math.h>
#include "taylor.h"

MY_FLOAT **taylor_coefficients_lorenz(MY_FLOAT, MY_FLOAT *, int);

int main(int argc, char *argv[])
{
   MY_FLOAT  xx[3], tmp[3], t, **coef;
   int       j, order=20, nsteps = 100;
   double    step_size;
   /* set initiaial conditions */
   xx[0] = 0.1;
   xx[1] = 0.2;
   xx[2] = 0.3;
   t     = 0.0;
   /* control parameters   */
   step_size= 0.1;

   /* integrate 100 steps */
   while( -- nsteps > 0) {
      /* do something with xx and t. We just print it */
      printf("%f %f %f %f\n", xx[0], xx[1], xx[2], t);

      /* compute the taylor coefficients */
      coef = taylor_coefficients_lorenz(t, xx, order);

      /* now we have the taylor coefficients in coef,
       * we can analyze them and choose a best step size.
       * Here we just integrate use the given stepsize.
       */
      tmp[0] = tmp[1] = tmp[2] = 0.0;
      for(j=order; j>0; j--) /* sum up the taylor polynomial */
        {
          tmp[0] = (tmp[0] + coef[0][j])* step_size;
          tmp[1] = (tmp[1] + coef[1][j])* step_size;
          tmp[2] = (tmp[2] + coef[2][j])* step_size;
        }
      /* advance one step */
      xx[0] = xx[0] + tmp[0];
      xx[1] = xx[1] + tmp[1];
      xx[2] = xx[2] + tmp[2];
      t += step_size; /* advance time */
    }
   return 0;
}
```

A   The Taylor method

Taylor method is one of the best known one step method for solving ordinary differential equations numerically. The idea is to advance the solution using a truncated Taylor expansion of the variables

about the current solution. Let

$$\mathbf{y}' = f(t, \mathbf{y}) \qquad \mathbf{y}(t_0) = \mathbf{y}_0 \tag{2}$$

be an initial value problem and let h be the integration step. To find $\mathbf{y}(t_0 + h)$, we expand $\mathbf{y}$ around t_0 and obtain

$$\mathbf{y}(t_0 + h) = \mathbf{y}(t_0) + \mathbf{y}'(t_0)h + \frac{1}{2!}\mathbf{y}''(t_0)h^2 + \cdots + \frac{1}{k!}\mathbf{y}^{(k)}(t_0)h^k + \cdots \tag{3}$$

e A numeric approximation of $\mathbf{y}(t_0 + h)$ is obtained by truncating (3) at a pre-determined order.
The main problem connected with the Taylor method is the need to compute high-order derivatives $\mathbf{y}'', \mathbf{y}''', \ldots, \mathbf{y}^{(k)}$ at t_0.

Van der Pol's Equation

To illustrate how to derive an integration scheme using the Taylor method, let's look at a special case of the famous Van der Pol's equation

$$\begin{array}{rcl} x' & = & y \\ y' & = & (1 - x^2)y - x \end{array} \tag{4}$$

with initial value $(x, y) = (2, 0)$. The second and third order derivatives of x, y with respect to time are

$$\begin{array}{rcl} x'' & = & (1 - x^2)y - x \\ y'' & = & x^3 - x - 2xy^2 + (x^4 - 2x^2)y \\ x''' & = & x^3 - x - 2xy^2 + (x^4 - 2x^2)y \\ y''' & = & 2x^3 - x^5 + (-1 + 5x^2 + 3x^4 - x^6)y + (-8x + 4x^3)y^2 - 2y^3 \end{array} \tag{5}$$

Hence a third order Taylor method for the initial value problem (4) is

$$\begin{array}{rcl} \begin{pmatrix} x_{n+1} \\ y_{n+1} \end{pmatrix} & = & \begin{pmatrix} x_n \\ y_n \end{pmatrix} + \begin{pmatrix} y_n \\ (1 - x_n^2)y_n - x_n \end{pmatrix} h \\ & & + \dfrac{1}{2!} \begin{pmatrix} (1 - x_n^2)y_n - x_n \\ x_n^3 - x_n - 2x_ny_n^2 + (x_n^4 - 2x_n^2)y_n \end{pmatrix} h^2 \\ & & + \dfrac{1}{3!} \begin{pmatrix} x_n^3 - x_n - 2x_ny_n^2 + (x_n^4 - 2x_n^2)y_n \\ 2x_n^3 - x_n^5 + (-1 + 5x_n^2 + 3x_n^4 - x_n^6)y_n + (-8x_n + 4x_n^3)y_n^2 - 2y_n^3 \end{pmatrix} h^3 \\ \begin{pmatrix} x_0 \\ y_0 \end{pmatrix} & = & \begin{pmatrix} 2 \\ 0 \end{pmatrix} \end{array}$$

As one can see from these equations, expressions for higher order derivatives are quite complicated, and the complexity increases dramatically as order increases. This difficulty is precisely the reason that Taylor method is not widely used.

Fortunately, for initial value problems where f is composed of polynomials and elementary functions, the higher order derivatives can be generated automatically. In fact, this is precisely the motivation of writing **taylor**.

Automatic Generation of Taylor Coefficients

The algorithm for computing Taylor coefficients recursively has been known since the 60s and is commonly referenced as *automatic differentiation* in the literature. It has been employed in software packages such as ATOFMT. A detailed description of the algorithm can be found in [1] (see more references therein). Here we give a brief account of the idea involved.

Let $f(t)$ be an analytic function and denote the ith Taylor coefficient at t_0 by

$$(f)_i = \frac{f^i(t_0)}{i!}$$

where $f^i(t)$ is the ith derivative of f at t_0. The Taylor expansion of $f(t)$ around t_0 can be conveniently expressed as

$$f(t_0 + h) = (f)_0 + (f)_1 h + (f)_2 h^2 + \cdots + (f)_n h^n \cdots$$

Let $(p)_i, (q)_i$ be the ith Taylor coefficients of p, q at t_0. The Taylor coefficients for $p \pm q$, pq and p/q can be obtained recursively using the following rules.

$$
\begin{aligned}
(p \pm q)_i &= (p)_i \pm (q)_i \\
(pq)_i &= \sum_{r=0}^{i} (p)_r (q)_{i-r} \\
\left(\frac{p}{q}\right)_i &= \frac{1}{q}\left\{ (p)_i - \sum_{r=1}^{i} (q)_r \left(\frac{p}{q}\right)_{i-r} \right\}
\end{aligned}
\tag{6}
$$

To compute the Taylor coefficients for (2), one first decomposes the right hand side of the differential equation into a series of simple expressions by introducing new variables, such that each expression involves only one arithmetic operation. These expressions are commonly called *code lists*. One then uses the recursive relations (6) and the initial values to generate the Taylor coefficients for all the the variables. For example, the Van der Pol equation (4) can be decomposed as

$$
\begin{aligned}
&u_1 = x, \quad u_2 = y, \quad u_3 = 1, \quad u_4 = u_1 u_1 \\
&u_5 = u_3 - u_4, \quad u_6 = u_5 u_2, \quad u_7 = u_6 - u_1 \\
&u_1' = u_2, \quad u_2' = u_7
\end{aligned}
$$

Using the initial value $(x_0, y_0) = (2, 0)$, the Taylor coefficients of all u_is can be easily generated using (6). The Taylor coefficients for elementary functions can also be generated recursively. Some of the rules are:

$$(p^a)_i = \frac{1}{p}\sum_{r=0}^{i-1}\left(a - \frac{r(a+1)}{i} \right)(p)_{i-r}(p^a)_r \quad \text{where } a \text{ is a real constant}$$

$$(e^p)_i = \sum_{r=0}^{i-1}\left(1 - \frac{r}{i} \right)(e^p)_r (p)_{i-r}$$

$$(\ln p)_i = \frac{1}{p}\left\{ (p)_i - \sum_{r=1}^{i-1}\left(1 - \frac{r}{i} \right)(p)_r (\ln p)_{i-r} \right\}$$

$$(\sin p)_i = \sum_{r=0}^{i-1}\left(\frac{r+1}{i} \right)(\cos p)_{i-1-r}(p)_{r+1}$$

$$(\cos p)_i = -\sum_{r=0}^{i-1}\left(\frac{r+1}{i} \right)(\sin p)_{i-1-r}(p)_{r+1}$$

$$(\tan^{-1} p)_i = \sum_{r=0}^{i-1}\left(1 - \frac{r}{i} \right)\left(\frac{1}{1+p^2} \right)_r (p)_{k-r}$$

References

[1] À. Jorba and M. Zou. A software package for the numerical integration of ODE by means of high-order Taylor methods. *Exp. Math.*, 14(1):99–117, 2005.

[2] J. Gimeno, À. Jorba, M. Jorba-Cuscó, N. Miguel, and M. Zou. Numerical integration of high-order variational equations of odes. Revisions requested, 2022.