PHY1610H - Scientific Computing: Ordinary Differential Equations

Ramses van Zon and Marcelo Ponce

SciNet HPC Consortium/Physics Department
University of Toronto

February, 2021

Today's class

Today we will discuss the following topics:

- How to solve ordinary differential equations, numerically (generally).
- Implementations using Libraries
- GSL & Boost

Ordinary Differential Equations

Ordinary Differential Equations

- Are equations with derivatives with respect to 1 variable, e.g.

$$\frac{dx}{dt} = f(x,t)$$

- There can be more than one such equation, e.g.

$$\frac{dx_1}{dt} = f_1(x_1, x_2, t); \qquad \frac{dx_2}{dt} = f_2(x_1, x_2, t)$$

- The derivative can be of higher order to, e.g.

$$\frac{d^2x}{dt^2} = f(x,t)$$

But this can be written, by setting $x_1 = x$, $x_2 = dx/dt$, to

$$\frac{dx_1}{dt} = x_2; \qquad \frac{dx_2}{dt} = f(x_1, t)$$

- This class includes Newtonian dynamics.

ODE Examples

Lotka–Volterra (predator/pray)

$$\frac{dx}{dt} = x(\alpha - \beta y)$$

$$\frac{dy}{dt} = -y(\gamma - \delta x)$$

Harmonic oscillator

$$\begin{aligned} \frac{dx}{dt} &= y \\ \frac{dy}{dt} &= -x \end{aligned}$$

Rate equations (chemistry)

$$\begin{aligned} \frac{dx}{dt} &= -2k_1x^2y + 2k_2z^2 \\ \frac{dy}{dt} &= -k_1x^2y + k_2z^2 \\ \frac{dz}{dt} &= 2k_1x^2y - 2k_2z^2 \end{aligned}$$

Lorenz system (weather)

$$\begin{aligned} \frac{dx}{dt} &= \sigma(y - x) \\ \frac{dy}{dt} &= x(\rho - z) - y \\ \frac{dz}{dt} &= xy - \beta z \end{aligned}$$

Numerical approaches

Start from the general form:

$$\frac{dx_i}{dt} = f(t, \{x_j\})$$

- All approaches will evaluate f at discrete points t_0, t_1, ...
- Initial conditions: specify $x_i(t_0)$. .
- Consecutive points may have a fixed step size $h = t_{k+1} - t_k$ or may be adaptive.
- $\{x_j(t_{i+1})\}$ may be implicitly dependent on f at that value.

Stiff ODEs

- A stiff ODE is one that is hard to solve, i.e. requiring a very small step size h or leading to instabilities in some algorithms.
- Usually due to wide variation of time scales in the ODEs.
- Not all methods equally suited for stiff ODEs

ODE solvers: Euler

To solve:

$$\frac{dx}{dt} = f(t, x)$$

Simple approximation:

$$x_{n+1} \approx x_n + hf(t_n, x_n) \qquad \text{“}forward\ Euler\text{”}$$

Why?

$$x(t_n + h) = x(t_n) + h\frac{dx}{dt}(t_n) + \mathcal{O}(h^2)$$

So:

$$x(t_n + h) = x(t_n) + hf(t_n, x_n) + \mathcal{O}(h^2)$$

- $\mathcal{O}(h^2)$ is the local error.
- For given interval $[t_1, t_2]$, there are $n = (t_2 - t_1)/h$ steps
- Global error: $n \times \mathcal{O}(h^2) = \mathcal{O}(h)$
- Not very accurate, nor very stable.

Stability

Example: solve harmonic oscillator numerically:

$$\frac{dx^{(1)}}{dt} = x^{(2)}$$

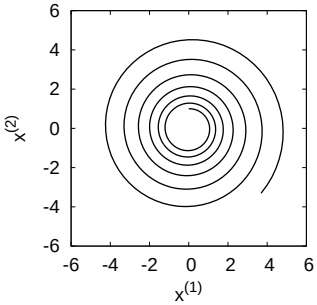
$$\frac{dx^{(2)}}{dt} = -x^{(1)}$$

Use Euler ($x_{n+1} \approx x_n + hf(t_n, x_n)$) gives

$$\begin{pmatrix} x_{n+1}^{(1)} \\ x_{n+1}^{(2)} \end{pmatrix} = \begin{pmatrix} 1 & h \\ -h & 1 \end{pmatrix} \begin{pmatrix} x_n^{(1)} \\ x_n^{(2)} \end{pmatrix}$$

Stability

$$\begin{pmatrix} x_{n+1}^{(1)} \\ x_{n+1}^{(2)} \end{pmatrix} = \begin{pmatrix} 1 & h \\ -h & 1 \end{pmatrix} \begin{pmatrix} x_n^{(1)} \\ x_n^{(2)} \end{pmatrix}$$



Stability governed by eigenvalues $\lambda_\pm = 1 \pm ih$ of that matrix. $|\lambda_\pm| = \sqrt{1+h^2} > 1$

$\Rightarrow$ Unstable for any h!

ODE solvers: implicit mid-point Euler

To solve:

$$\frac{dx}{dt} = f(t, x)$$

Symmetric simple approximation:

$$x_{n+1} \approx x_n + hf(t_n, (x_n + x_{n+1})/2) \qquad \text{“}mid\text{-}point\ Euler\text{”}$$

This is an implicit formula, i.e., has to be solved for x_{n+1}.

Harmonic oscillator

$$\begin{bmatrix} 1 & -\frac{h}{2} \\ \frac{h}{2} & 1 \end{bmatrix} \begin{bmatrix} x_{n+1}^{[1]} \\ x_{n+1}^{[2]} \end{bmatrix} = \begin{bmatrix} 1 & \frac{h}{2} \\ -\frac{h}{2} & 1 \end{bmatrix} \begin{bmatrix} x_n^{[1]} \\ x_n^{[2]} \end{bmatrix} \Rightarrow \begin{bmatrix} x_{n+1}^{[1]} \\ x_{n+1}^{[2]} \end{bmatrix} = M \begin{bmatrix} x_n^{[1]} \\ x_n^{[2]} \end{bmatrix}$$

Eigenvalues M are $\lambda_\pm = \frac{(1 \pm ih/2)^2}{1 + h^2/4}$ so $|\lambda_\pm| = 1$

$\Rightarrow$ Stable for all h

Implicit methods often more stable and allow larger step size h.

ODE solvers: implicit mid-point Euler

To solve:

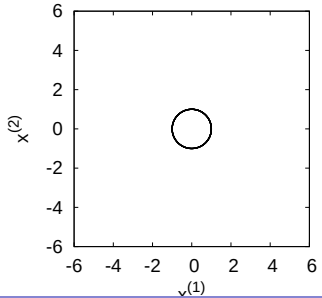
$$\frac{dx}{dt} = f(t, x)$$

Symmetric simple approximation:

$$x_{n+1} \approx x_n + hf(t_n, (x_n + x_{n+1})/2) \qquad \text{“}mid\text{-}point\ Euler\text{”}$$

This is an implicit formula, i.e., has to be solved for x_{n+1}.

Harmonic oscillator



ODE solvers: Predictor-Corrector

1. Computation of new point
2. Correction using that new point

- Gear P.C.: keep previous values of x to do higher order Taylor series (predictor), then use f in last point to correct.

  Can suffer from catastrophic cancellation at very low h.
- Runge-Kutta: Refines by using mid-points. 4th order version:

$$
\begin{aligned}
k_1 &= hf(t, x) \\
k_2 &= hf(t + h/2, x + k_1/2) \\
k_3 &= hf(t + h/2, x + k_2/2) \\
k_4 &= hf(t + h, x + k_3) \\
x' &= y + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6}
\end{aligned}
$$

Further ODE solvers techniques

Adaptive methods

As with the integration, rather than taking a fixed h, vary h such that the solution has a certain accuracy.

Don't code this yourself! Adaptive schemes are implemented in libraries such as the `gsl` and `boost::numeric::odeint`.

- GSL is C based, ie. it uses C-style functions calls with structs and pointers.
- Boost is written in C++ and relies heavily on templates.

GSL ODE Example I

$$\frac{dx}{dt} = \frac{3}{2t^2} + \frac{x}{2t}$$

```
/// @file GSLexample.cpp
/// Solves the simple ODE x' = 3/(2t^2) + x/(2t) with initial condition x(1)=0
/// Analytic solution is x(t) = sqrt(t) - 1/t

#include <iostream>
#include <gsl/gsl_odeiv2.h>
#include <gsl/gsl_errno.h>

/// @brief Time derivatives for the ODE (aka right-hand side).
/// @param  t       time value (input)
/// @param  y       array with the state values (input)
/// @param  dydt    array with the values of the time derivatives (output)
/// @param  param   void pointer to any parameters  (input)
/// @return integer value that should be GSL_SUCCESS if nothing went wrong.
int f(double t, const double y[], double dydt[], void* params) {
    dydt[0] = 1.0/(t*t*t) + y[0]/(2.0*t);
    return GSL_SUCCESS;
}
```

GSL ODE Example II

```
/// @brief solution of ode
int main() {
    double hstart  = 1.e-3; //< time step to start with
    double abstol  = 1.e-6; //< required absolute precision
    double reltol  =  0.0;  //< required relative precision (0.0=no check)
    double tbegin  =  1.0;  //< final time
    double tend    = 10.0;  //< final time
    int    npoints = 100;   //< number of time points to write out
    // Create ODE system and a driver for 'Runge-Kutta Caah-Karp'
    gsl_odeiv2_system sys = {f, nullptr, 1, nullptr};
    gsl_odeiv2_driver* d = gsl_odeiv2_driver_alloc_y_new(&sys, gsl_odeiv2_step_rkck, hstart,
         abstol, reltol);
    // Solve at equidistant time points
    double t = tbegin;   // time
    double y[1] = {0.0}; // 2 dimensional state
    for (int i = 1; i <= npoints; i++) {
        double ti = tbegin + (i * (tend-tbegin)) / npoints;
        int status = gsl_odeiv2_driver_apply(d, &t, ti, y);
        if (status != GSL_SUCCESS) {
            std::cout << "error, return value=" << status << std::endl;
            break;
        }
        std::cout << t << ' ' << y[0] << std::endl;
    }
    gsl_odeiv2_driver_free(d);
}
```

GSL ODE Example: Van de Pol equation (from GSL docs):

$$\frac{d^2x}{dt^2} - \mu(1-x^2)\frac{dx}{dt} + x = 0 \quad \Rightarrow \quad \begin{cases} \dot{x} = y \\ \dot{y} = \mu(1-x^2)y - x \end{cases}$$

```
#include <iostream>
#include <gsl/gsl_errno.h>
#include <gsl/gsl_matrix.h>
#include <gsl/gsl_odeiv2.h>

int func(double t,const double y[],double f[],void*
     params)
{
 double mu = *(double *)params;
 f[0] = y[1];
 f[1] = -y[0] - mu*y[1]*(y[0]*y[0] - 1);
 return GSL_SUCCESS;
}

int jac(double t, const double y[], double *dfdy,
        double dfdt[], void *params) {
 double mu = *(double *)params;
 gsl_matrix_view dfdy_mat =
  gsl_matrix_view_array(dfdy, 2, 2);
 gsl_matrix * m = &dfdy_mat.matrix;
 gsl_matrix_set (m, 0, 0, 0.0);
 gsl_matrix_set (m, 0, 1, 1.0);
 gsl_matrix_set (m, 1, 0, -2.0*mu*y[0]*y[1] - 1.0);
 gsl_matrix_set (m, 1, 1, -mu*(y[0]*y[0] - 1.0));
 dfdt[0] = 0.0;
 dfdt[1] = 0.0;
 return GSL_SUCCESS;
}
```

```
int main (void)
{
 double mu = 10;
 gsl_odeiv2_system sys = {func, jac, 2, &mu};

 gsl_odeiv2_driver * d =
  gsl_odeiv2_driver_alloc_y_new(&sys,
       gsl_odeiv2_step_rk8pd,
                                  1e-6, 1e-6, 0.0);
 int i;
 double t = 0.0, t1 = 100.0;
 double y[2] = { 1.0, 0.0 };

 for (i = 1; i <= 100; i++) {
   double ti = i * t1 / 100.0;
   int status = gsl_odeiv2_driver_apply(d, &t, ti, y)
        ;
   if (status != GSL_SUCCESS) {
     std::cout << "error, return value="<<status<<"\n
          ";
     break;
   }
  std::cout << t << y[0] << y[1] << std::endl;
 }
 gsl_odeiv2_driver_free (d);

 return 0;
}
```

BOOST ODE Example I

$$\frac{dx}{dt} = \frac{3}{2t^2} + \frac{x}{2t}$$

```
/// @file BOOSTexample.cpp
/// Solves the simple ODE x' = 3/(2t^2) + x/(2t) with initial condition x(1)=0
/// Analytic solution is x(t) = sqrt(t) - 1/t

#include <iostream>
#include <boost/exception/diagnostic_information.hpp>
#include <boost/numeric/odeint.hpp>

/// @brief Time derivatives for the ODE (aka right-hand side)
/// @param  x      array with the state values (input)
/// @param  dxdt   array with the values of the time derivatives (output)
/// @param  t      time value (input)
/// @return void
void f(const double x, double &dxdt, const double t) {
    dxdt = 1.0/(2.0*t*t) + x/(2.0*t);
}
```

BOOST ODE Example II

```
/// @brief solution of ode
int main() {
    double hstart  = 0.1;   //< initial time step to try
    double abstol  = 1.e-6; //< required absolute precision
    double reltol  =  0.0;  //< required relative precision (0.0=no check)
    double tbegin  =  1.0;  //< initial time
    double tend    = 10.0;  //< final time
    int    npoints = 100;   //< number of time points to write out
    // Create a stepper object using runge_kutta_cash_karp54
    using namespace boost::numeric::odeint;
    typedef runge_kutta_cash_karp54<double> stepper_type;
    auto stepper = make_controlled(abstol, reltol, stepper_type());
    // Solve at equidistant time points
    double t = tbegin;
    double x = 0.0;
    for (int i = 1; i <= npoints; i++) {
        double ti = tbegin + (i * (tend-tbegin)) / npoints;
  try {
      integrate_adaptive(stepper, f, x, t, ti, hstart);
      t = ti;
  }
  catch (boost::exception& e) {
      std::cout << boost::diagnostic_information(e) << std::endl;
            break;
  }
        std::cout << t << ' ' << x << std::endl;
    }
}
```

BOOST ODE Example

```
/* we solve the simple ODE x' = 3/(2t^2) + x/(2t)
 * with initial condition x(1) = 0.
 * Analytic solution is x(t) = sqrt(t) - 1/t */

#include <iostream>
#include <boost/numeric/odeint.hpp>
using namespace boost::numeric::odeint;

void rhs(const double x, double &dxdt, const double t) {
  dxdt = 3.0/(2.0*t*t) + x/(2.0*t);
}

void write_cout(const double &x, const double t) {
  std::cout << t << '\t' << x << std::endl;
}

typedef runge_kutta_dopri5<double> stepper_type;

int main() {
  double x = 0.0;

  integrate_adaptive(
          make_controlled(1E-6, 1E-6, stepper_type()),
          rhs, x, 1.0, 10.0, 0.1, write_cout);
}
```

Final Comments

Applications

- Molecular Dynamics (to be discussed later in the course)
- N-body

References

- https://www.gnu.org/software/gsl/doc/html/ode-initval.html
- https://www.boost.org/doc/libs/1_62_0/libs/numeric/odeint/doc/html/boost_numeric_odeint/tutorial/all_examples.html