Ordinary Differential Equations

Ordinary Differential Equations

An nth order ODE is one where the dependent variables are functions of a single variable, and the highest order derivative is n. A second-order ODE might look like

$$m\frac{\mathrm{d}^2x}{\mathrm{d}t^2} = f(x, \dot{x}, t)$$

We can reduce to a system of 2 first-order ODEs by a change of variables: define

$$Y_0 = x, Y_1 = \frac{\mathrm{d}x}{\mathrm{d}t}$$

And we now have:

$$\frac{\mathrm{d}Y_0}{\mathrm{d}t} = Y_1, \frac{\mathrm{d}Y_1}{\mathrm{d}t} = f(Y_0, Y_1, t)/m$$

So we need to focus on solving coupled first-order ODEs. We can reduce many interesting problems to this form.

Simple ODE example: Spinning Ring in Magnetic Field

Old IB problem:

$$\frac{\mathrm{d}^2\theta}{\mathrm{d}t^2} = -\frac{2}{\tau}\sin^2\theta\,\frac{\mathrm{d}\theta}{\mathrm{d}t}$$

with approximate solution (for light damping)

$$\frac{\mathrm{d}\theta}{\mathrm{d}t} \approx \omega_0 e^{-t/\tau}$$

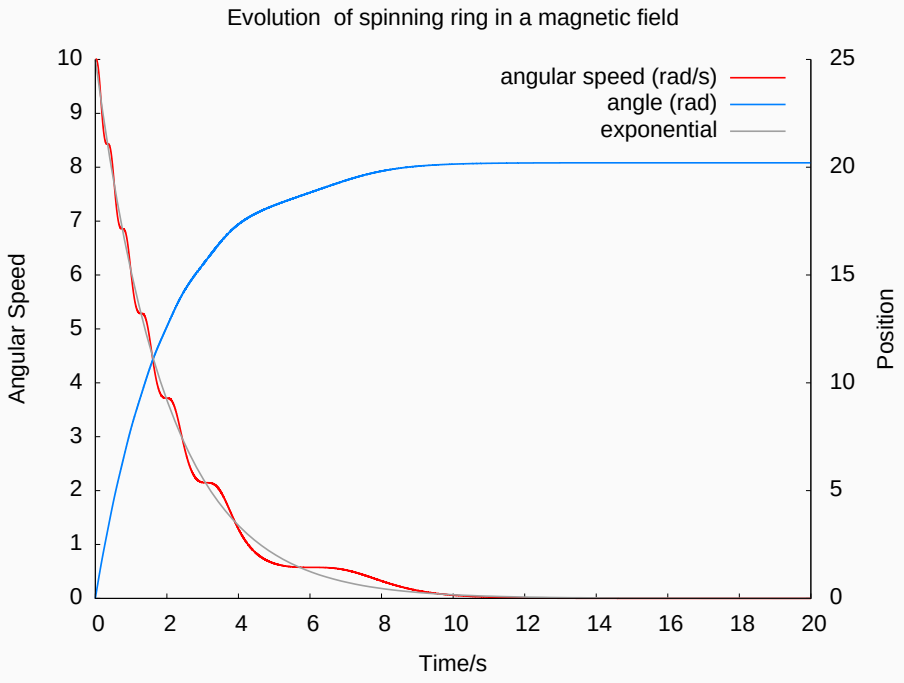
We set

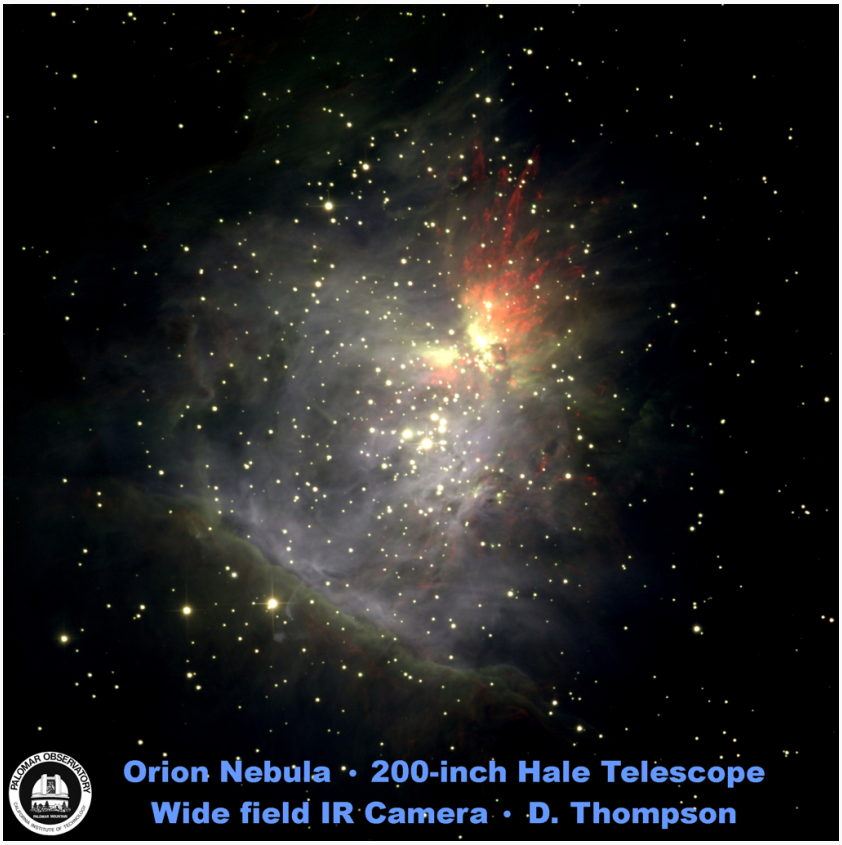
$$Y_0 \equiv \theta, Y_1 \equiv \dot{\theta}$$

to obtain

$$\begin{aligned} \dot{Y_0} &= Y_1 \\ \dot{Y_1} &= -\frac{2}{\tau}\sin^2(Y_0)\,Y_1 \end{aligned}$$

Simple ODE example: Spinning Ring in Magnetic Field





N-body problem example

For N particles moving in 3 dimension, we will need $6N$ coupled first-order ODEs to describe the system.

- $3N$ positions
- $3N$ velocities

For the case of gravity for example:

$$m_i \frac{\mathrm{d}^2 \mathbf{r}_i}{\mathrm{d}t^2} = -\sum_{i \neq j} \frac{G m_i m_j}{r_{ij}^3} \mathbf{r}_{ij}$$

with the particle index running i from 0 to $(N-1)$.

Learn to love 0-based arrays

We transform to a new set of variables, $Y_i, i = 0...(6N-1)$. Define

$$Y_0 = x_0, Y_1 = y_0, Y_2 = z_0$$

$$Y_3 = x_1, Y_4 = y_1, Y_5 = z_1$$

etc for the positions, the final position being stored in Y_{3N-1}. The speeds are then

$$Y_{3N} = \dot{x}_0, Y_{3N+1} = \dot{y}_0, Y_{3N+2} = \dot{z}_0$$

$$Y_{3N+3} = \dot{x}_1, Y_{3N+4} = \dot{y}_1, Y_{3N+5} = \dot{z}_1$$

etc. with the final element being $Y_{6N-1} = \dot{z}_{N-1}$.

We now have $3N$ trivial first order ODEs relating speeds:

$$\begin{aligned} \dot{Y}_0 &= Y_{3N} \\ \dot{Y}_1 &= Y_{3N+1} \\ \dot{Y}_2 &= Y_{3N+2} \dots \\ \dot{Y}_{3N-1} &= Y_{6N-1} \end{aligned}$$

and $3N$ equations specifying the accelerations:

$$\begin{aligned} \dot{Y}_{3N} &= F_{x,0}/m_0 \\ \dot{Y}_{3N+1} &= F_{y,0}/m_0 \\ \dot{Y}_{3N+2} &= F_{z,0}/m_0 \end{aligned}$$

etc

Integrating ODEs: Euler's method

Consider one 1st order ODE

$$\frac{dy}{dx} = f(x, y)$$

with a boundary condition $y(x_0) = y_0$. This is an initial value problem. (More complex problems involve specifying the solution at more than one location and other methods are needed.)

We want to find an approximate solution to this continuous function y, at a set of discrete points x_i which we assume are equally spaced (for now).

Euler's 18th century algorithm

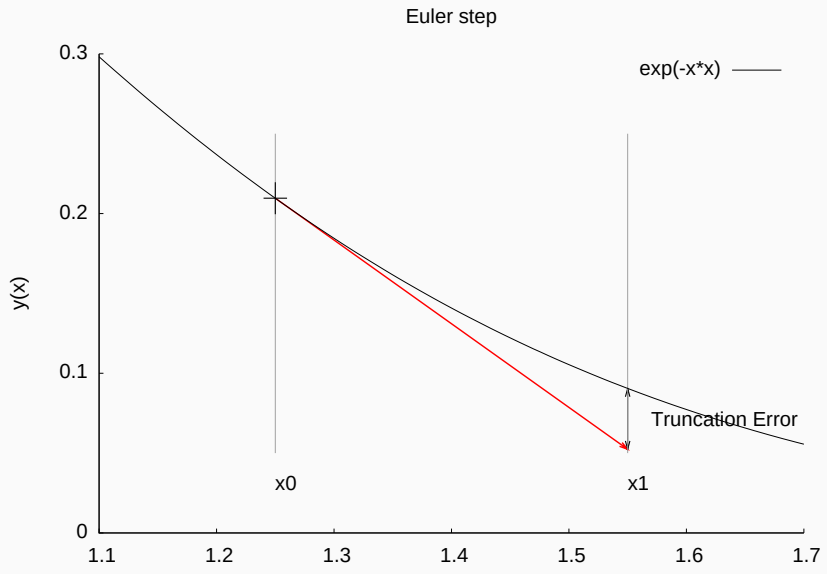
Choose a suitable step size h

- Find the gradient y'_i at the current location x_i
- Set $y_{i+1} = y_i + h\,y'_i$
- Set $x_{i+1} = x_i + h$

Repeat as necessary.

Euler's Algorithm

Illustrated for the ODE: $y' = -2\,x\,y$ (which has a solution $A\exp(-x^2)$)



Truncation Error in Euler's method

We approximate the function as a straight line over the distance h. There is an associated truncation error.

Truncation Error

Error in the estimated value due to finite step size

A Taylor expansion yields

$$y(x_i + h) = y(x_i) + h\,y'(x_i) + \frac{h^2}{2}y''(x_i) + \mathcal{O}(h^3)$$

In Euler's method we truncate the series after the linear term. The truncation error is $\mathcal{O}(h^2)$ in each step.

Truncation Error in Euler's method

- Now if we want to integrate over a range of order unity ($x = 0$ to 1 for example), then we need $\mathcal{O}(h^{-1})$ steps, so the total truncation error is $\mathcal{O}(h)$ if we assume (pessimistically) that the errors accumulate.
- So an accuracy of 1 part in 1 million needs of order a million steps. (Of course this is not true if the function is actually a straight line...)
- Euler's Method is called first order since its error over a finite scale goes as h^1. An n-th order method has a truncation error per step $\mathcal{O}(h^{n+1})$.

- So can we take an infinite number of small steps and get a perfect answer?
- No, because round-off error due to finite precision arithmetic also occurs. At each step we get a round-off error of some value ϵ, which depends on the computer's binary representation of numbers.
- Integrating over a finite range we accumulate a total round-off error $\sim \epsilon/h$, for a total error of

$$E \sim \frac{\epsilon}{h} + h$$

Moral

If we use small steps: round-off error dominates.

If we use large steps: truncation error dominates

- The minimum error occurs for $h \approx \epsilon^{1/2}$ and has a magnitude $E_{\min} \sim \epsilon^{1/2}$.
- The value of ϵ is machine-dependent:

$$\texttt{float} : \epsilon = 1.19 \times 10^{-7}$$

- So in single precision arithmetic, the minimum practicable step size is $h \sim 3 \times 10^{-4}$ yielding an accuracy of about 3×10^{-4}. This is generally not good enough.
- But in double precision, the minimum practicable step is $h \sim 1 \times 10^{-8}$ yielding an accuracy of about 1×10^{-8}. This is much better.

Error in Euler's method: Single Precision

```
// Euler solution to  x'' = -x, with x(0)=1, x'(0)=0
#include <iostream>
#include <cmath>
// Take one Euler step for this problem
// Convention is y[0]=x; y[1]=x' so ODEs are y[0]=-y[1]' and y[1]=y[0]'
void euler_step(float &t, float h, float y[2]) {
  float previous_y0 = y[0];
  float previous_y1 = y[1];
  y[1] += h * (-previous_y0);
  y[0] += h * previous_y1;
  t += h;
  return;
}
```

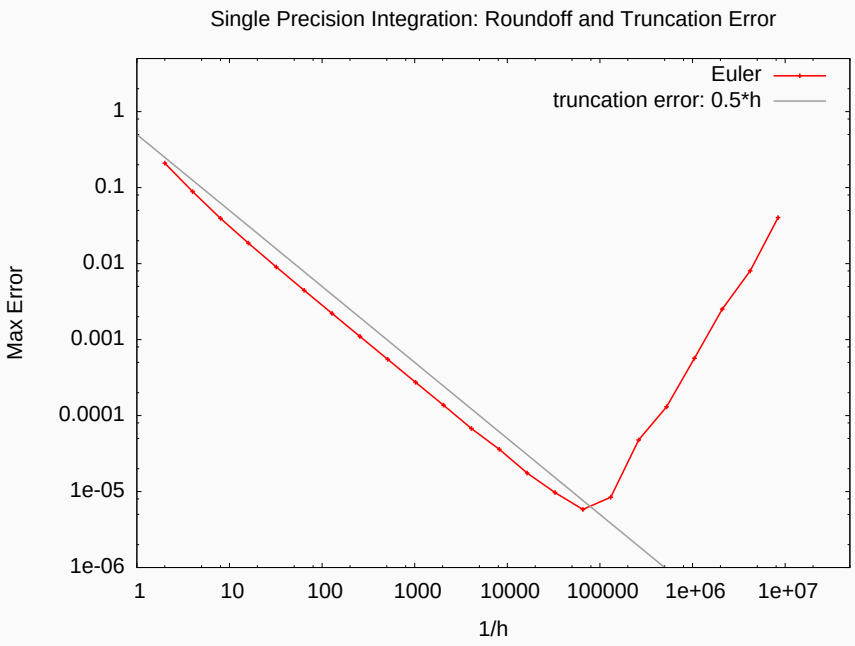


```
// Evaluate the ODE from t=0 to t=t_max using stepsize h
// Returns the maximum error between the result and the analytical solution
float integ(float h, float t_max) {
  // Set up the initial boundary conditions:
  float t = 0;
  float y[2];
  y[0] = 1.0;
  y[1] = 0.0;
  // Advance the solution until t>=t_max, storing max error
  float err_max = 0.0;
  while (t < t_max) {
    euler_step(t, h, y);
    float err = fabs(cos(t) - y[0]);
    if (err > err_max)
      err_max = err;
  }
  return err_max;
}
```

```cpp
int main() {
  for (float h = 0.5; h > 1e-7; h /= 2) {
    float err_euler = integ(h, 1.0);
    std::cout << h << " " << err_euler << " "
              << "\n";
  }
}
```


Errors Accumulation in Euler's method: Single Precision

Euler solution to $\ddot{x} = -x$ with $x(0) = 1, \dot{x}(0) = 0$.



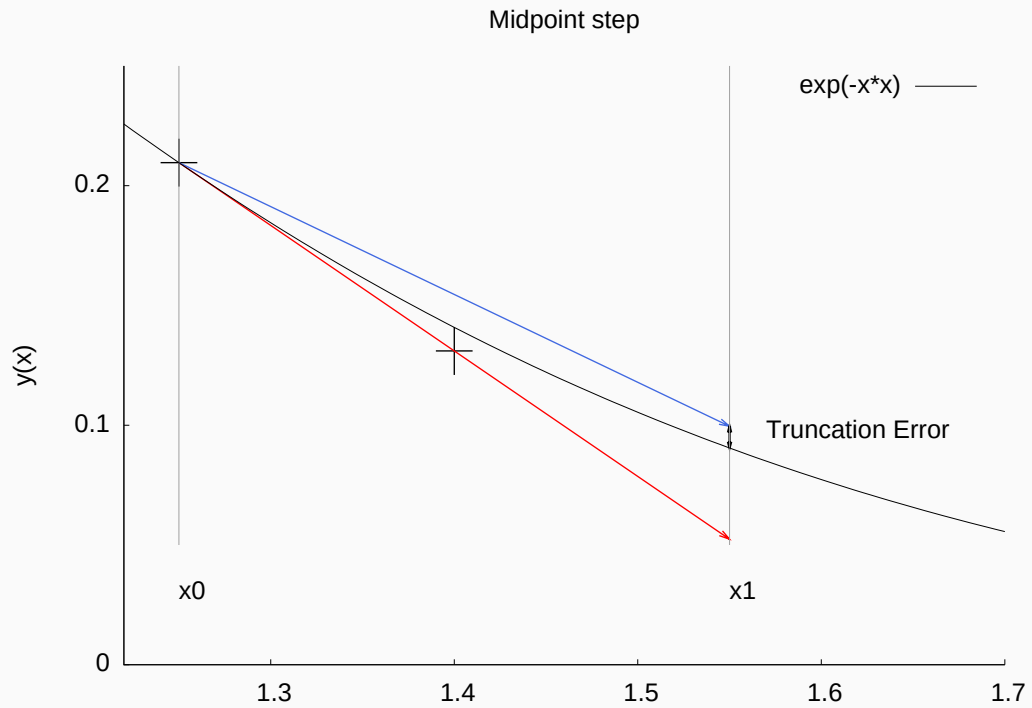
Maximum error over 1 second integration compared to analytic solution.

Higher Order Methods

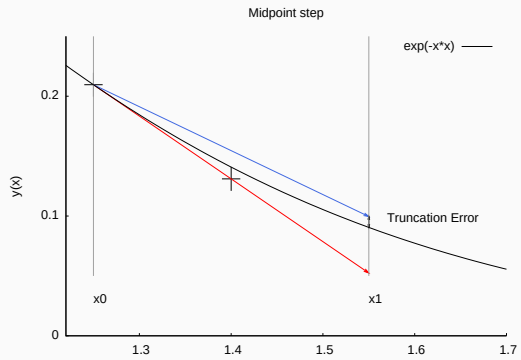
- Euler's method not generally good enough for serious computation, although in some problems it suffices, and with simple tweaks can be a useful workhorse.
- But there exist more accurate and efficient methods. The problem with Euler's method is that it is very asymmetric: it only evaluates the gradient at the beginning of the interval x_i.
- Higher order methods evaluate derivatives at sub-interval points and achieve higher accuracy. For example, the second-order mid-point method would take half a step and evaluate the gradient there.

Midpoint method: $\mathcal{O}(h^2)$

Illustrated for $x'' = -2xy$.



Midpoint method: $\mathcal{O}(h^2)$

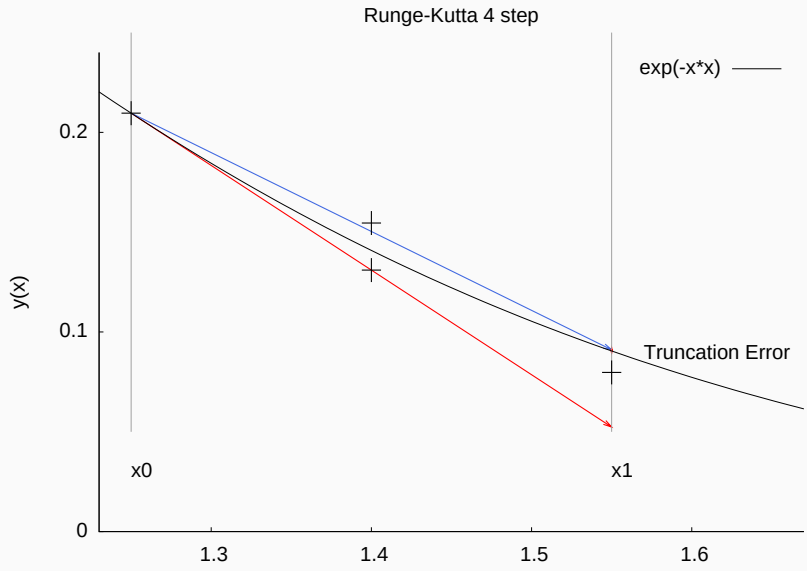


Mathematically, we have:

$$\begin{aligned} k_1 &= h\,y'(x_i, y_i) \\ k_2 &= h\,y'(x_i + h/2, y_i + k_1/2) \\ y_{i+1} &= y_i + k_2 + \mathcal{O}(h^3) \end{aligned}$$

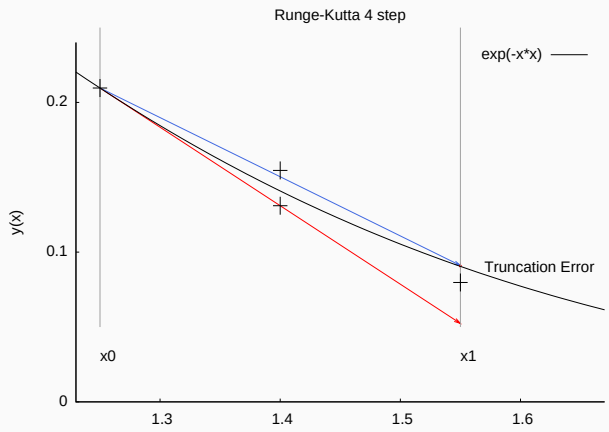
The first order errors cancel. This is called a “2nd Order Runge-Kutta method”. The price to be paid for this higher accuracy is more function evaluations (2) per step. But this is normally worthwhile.

4th order Runge-Kutta



4-th order Runge-Kutta schemes are popular, with evaluations at 4 trial points per step: the start, two midpoints, and one end point.

4th order Runge-Kutta



Mathematically:

$$\begin{aligned} k_1 &= h\,y'(x_i, y_i) \\ k_2 &= h\,y'(x_i + h/2, y_i + k_1/2) \\ k_3 &= h\,y'(x_i + h/2, y_i + k_2/2) \\ k_4 &= h\,y'(x_i + h, y_i + k_3) \end{aligned}$$

which yields a best step of

$$y_{i+1} = y_i + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6} + \mathcal{O}(h^5)$$

4th order Runge-Kutta

The total error E in an n'th order method is

$$E = \frac{\epsilon}{h} + h^n$$

for which h is minimised for

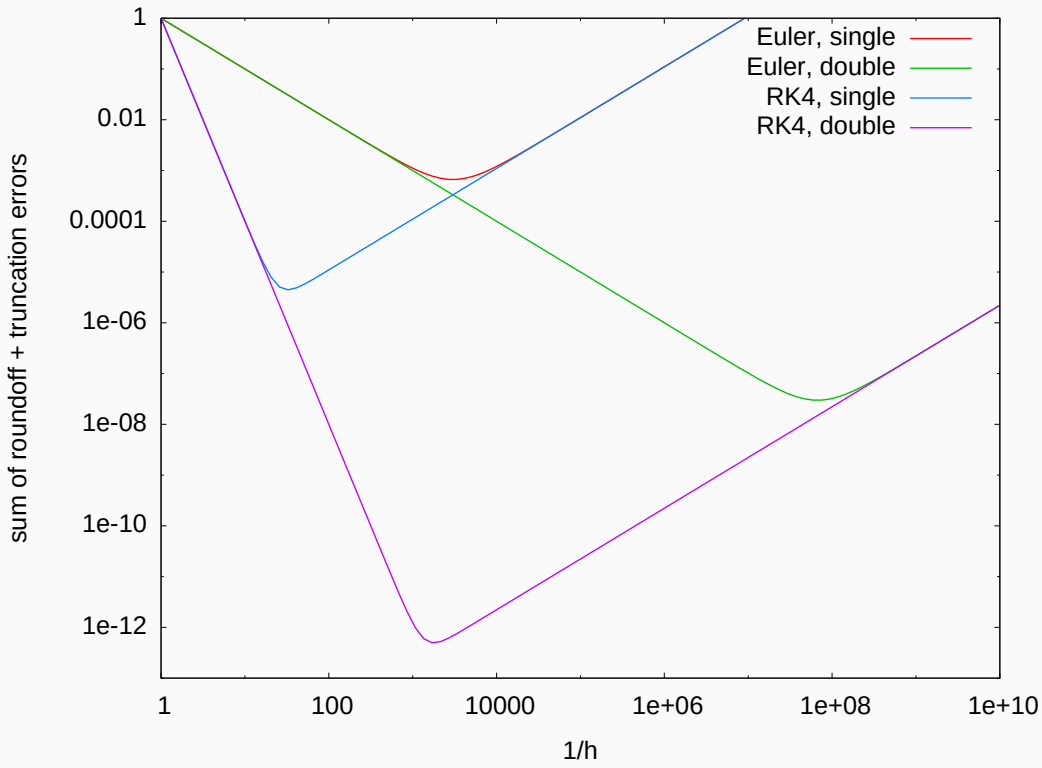
$$h_{\min} = \left(\frac{\epsilon}{n}\right)^{1/(n+1)}$$

The minimum error is then

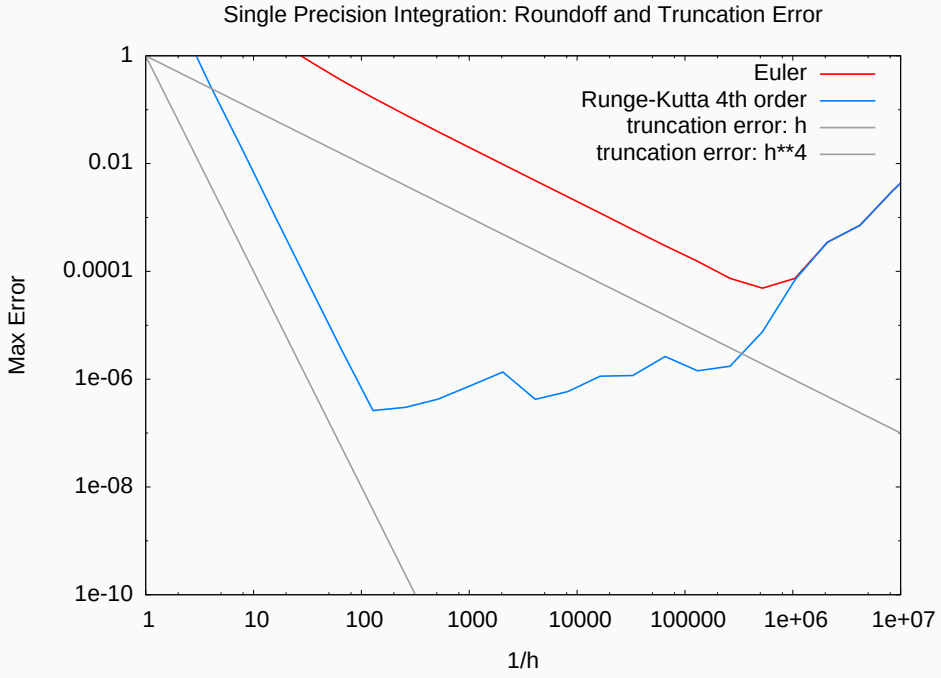
$$E_{\min} = \left(\frac{\epsilon}{n}\right)^{n/(n+1)}$$

For double precision arithmetic we get a sensible step size of $h_{\min} \sim 6 \times 10^{-4}$ and a corresponding error of $E_{\min} \approx 6 \times 10^{-13}$. (Compare this with the best values of $h \sim 10^{-8}$ and $E \sim 10^{-8}$ we obtain for Euler's method.)

Theoretical Errors in R-K integrations



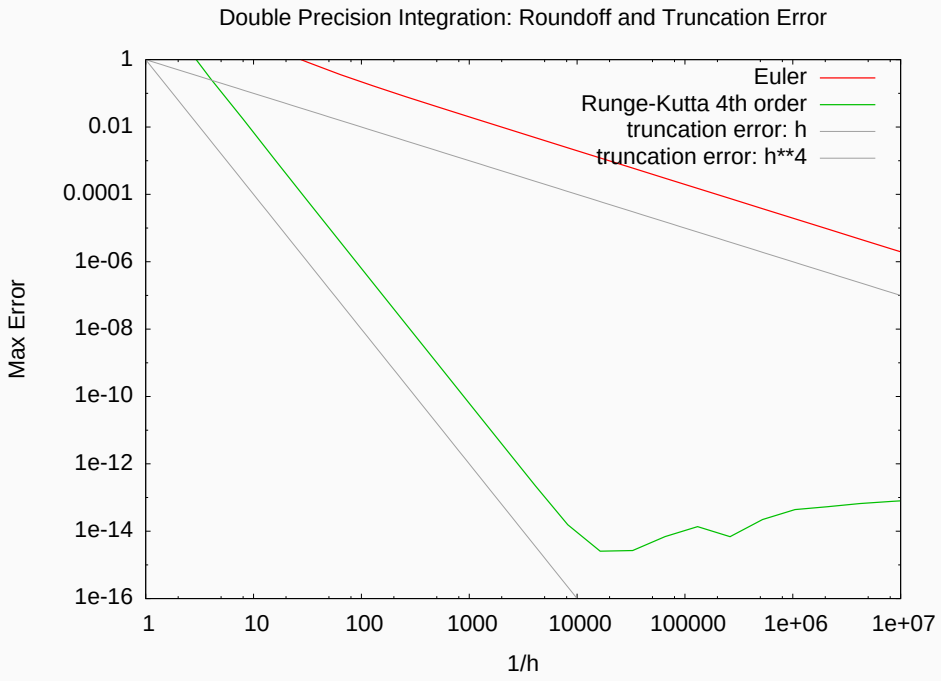
Real Errors in Runge-Kutta schemes: single-precision



Superiority of Runge-Kutta 4th order scheme is obvious.

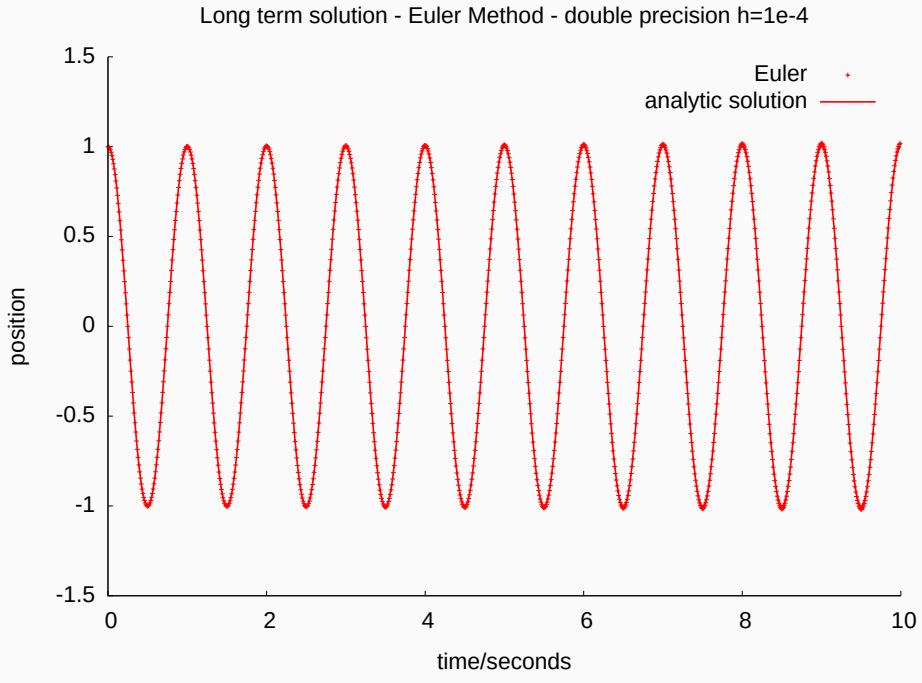
(Complete code to generate these data and the plot is in `ode1.cc` and `ode1.gp`)

Real Errors in Runge-Kutta schemes: double precision



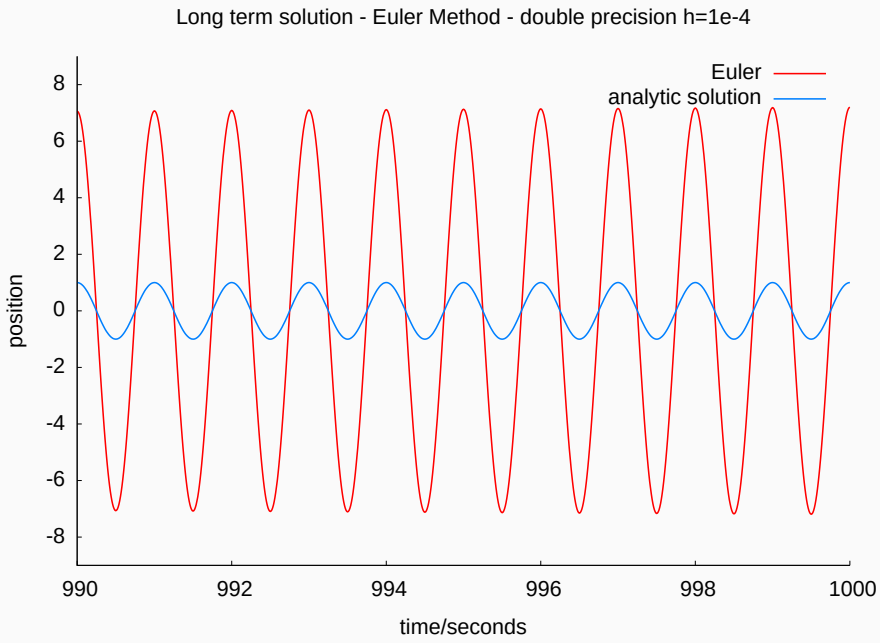
Superiority of double precision is clear. For this problem $h < 10^{-4}$ is reasonable

Long term solution using Euler. $\ddot{x} = -4\pi^2 x$



But Euler's method appears to be doing a good job for the first 10 periods (seconds)...maybe it's crude but effective. Let's keep integrating...

Long term solution using Euler



Amplitude $\sim$ 8 times too large after 1000 oscillations.

Origin of the Numerical Instability in Euler's Method

The scheme we are using updates position and velocity according to:

$$\begin{aligned} x_{i+1} &= x_i + \dot{x}_i \, \Delta t \\ \dot{x}_{i+1} &= \dot{x}_i + \ddot{x}_i \, \Delta t \end{aligned}$$

The total energy is just

$$E_i = \frac{1}{2} m \, \dot{x}_i^2 + \frac{1}{2} k \, x_i^2$$

So that after a little work we find

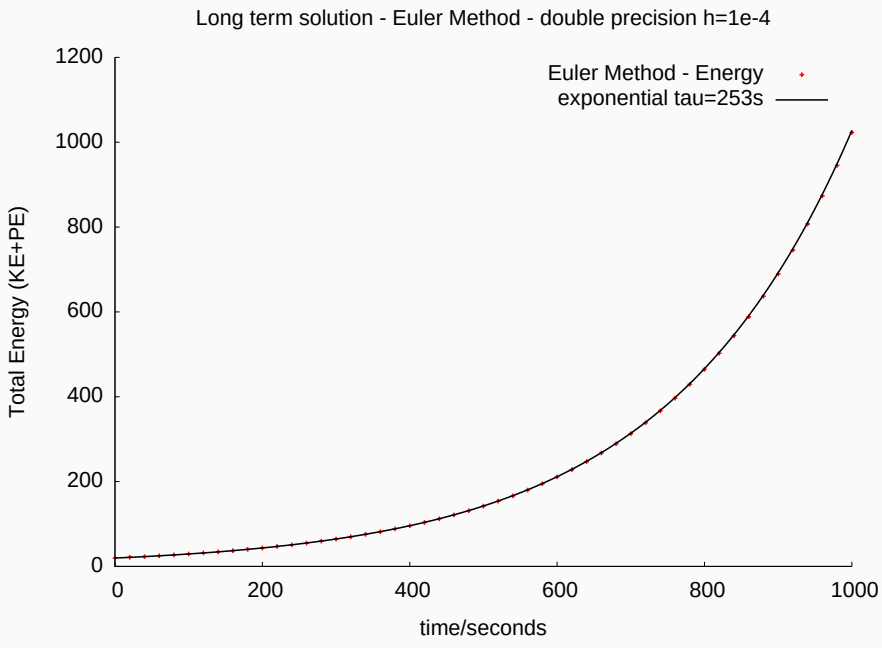
$$E_{i+1} = E_i + \left(\frac{1}{2} m \, \ddot{x}_i^2 + \frac{1}{2} k \, \dot{x}_i^2 \right) (\Delta t)^2$$

We can write this as

$$\frac{\Delta E}{\Delta t} = E\frac{k\Delta t}{m}$$

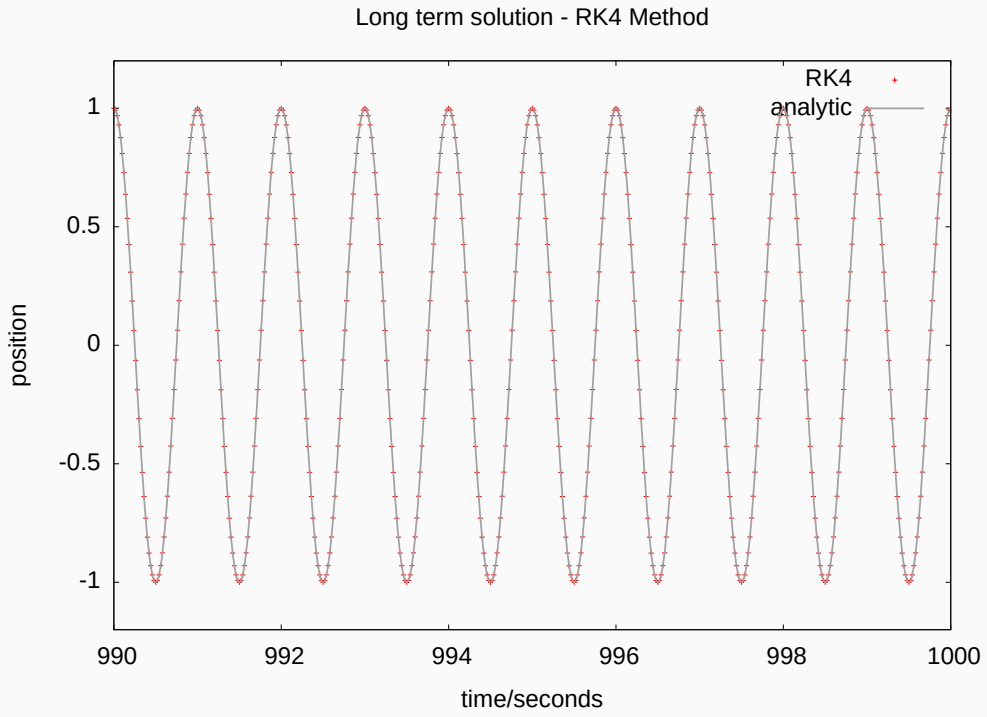
- We see that E will increase on each step by an amount proportional to E: we get exponential growth with a timescale $\tau = m/(k\Delta t)$. Even if we make the step smaller, the energy will still grow without limit.
- The method is intrinsically unstable for this oscillatory equation.
- But the same does not necessarily apply to other equations.

Long term solution using Euler



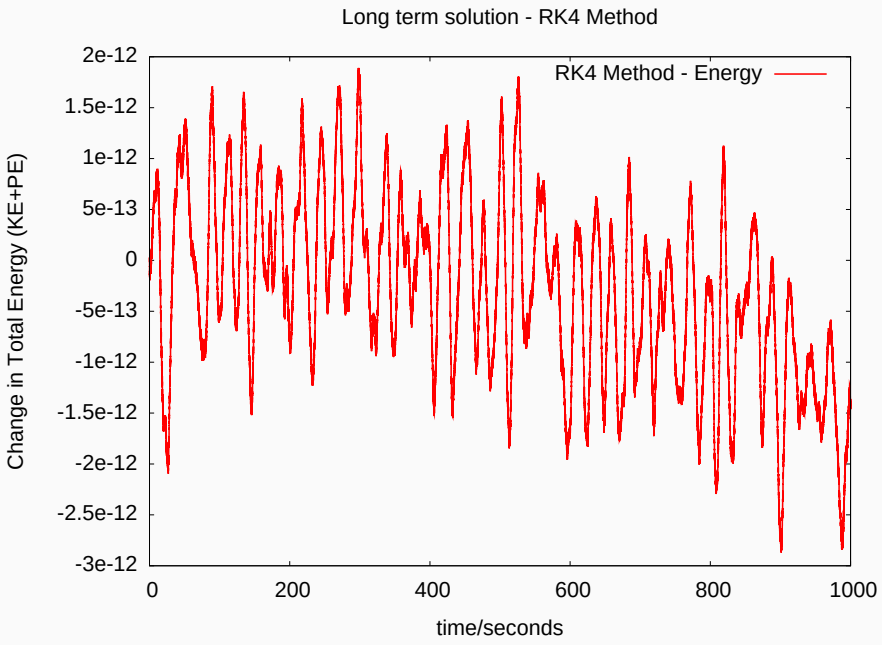
Initial energy = $2\pi^2$.
Exponential growth timescale $\tau = m/(k\Delta t) = 1/(4\pi^2\Delta t) = 253$s.

Long term solution using Runge-Kutta 4th order



This looks a lot better...

Long term solution using RK4



The energy stays constant to 1 part in 10^{12} again using $h = 10^{-4}$. True even after 10^6 oscillations.

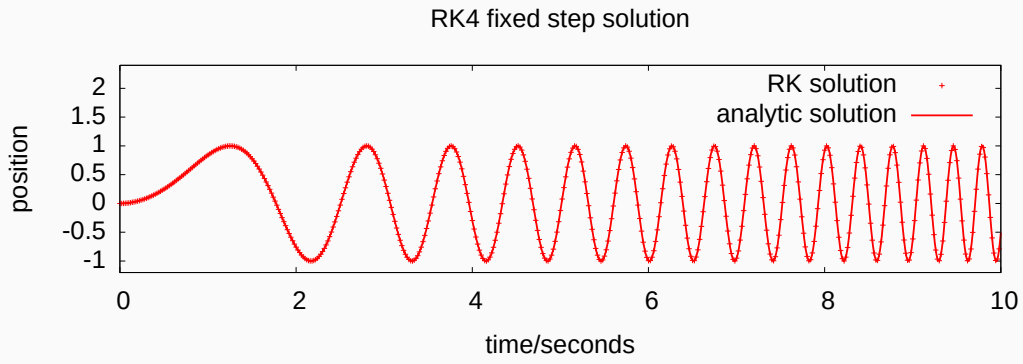
Adaptive Stepping

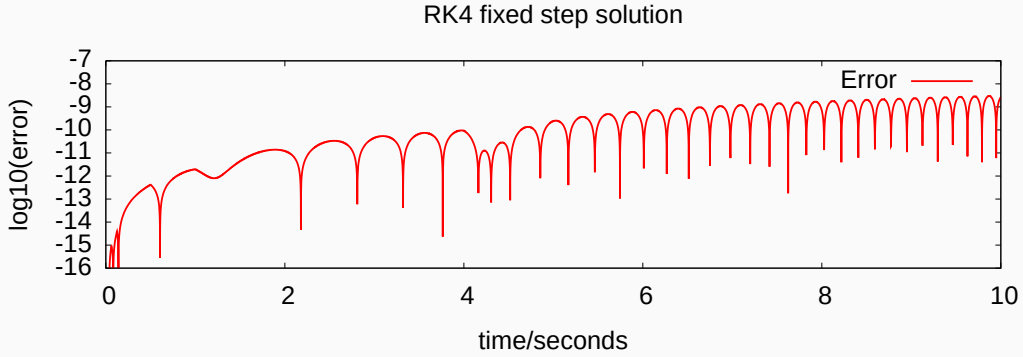
- In the simple SHM problem studied earlier, the period is constant, and using a constant step size h to advance the integration made sense.
- But for many ODEs, the scale over which the solution changes appreciably is not constant. For example

$$\frac{\mathrm{d}^2x}{\mathrm{d}t^2} = -4\,t^2\,\sin(t^2) + 2\,\cos(t^2)$$

  has a solution $x = \sin(t^2)$ which changes progressively more quickly as t increases.
- A physical example might be the close approach of two bodies interacting via an inverse square law potential — stars scattering off each other in a cluster, or the case of Rutherford scattering.
- In such cases, a constant step size is not sensible. To get an accurate answer we need to run with a very small step size which is very inefficient.

Look how the error evolves if we use a 4th order Runge-Kutta method with fixed step size of 10^{-5}:





Adaptive Step Methods try to find optimal step lengths for a given problem. One possible way of doing this is to

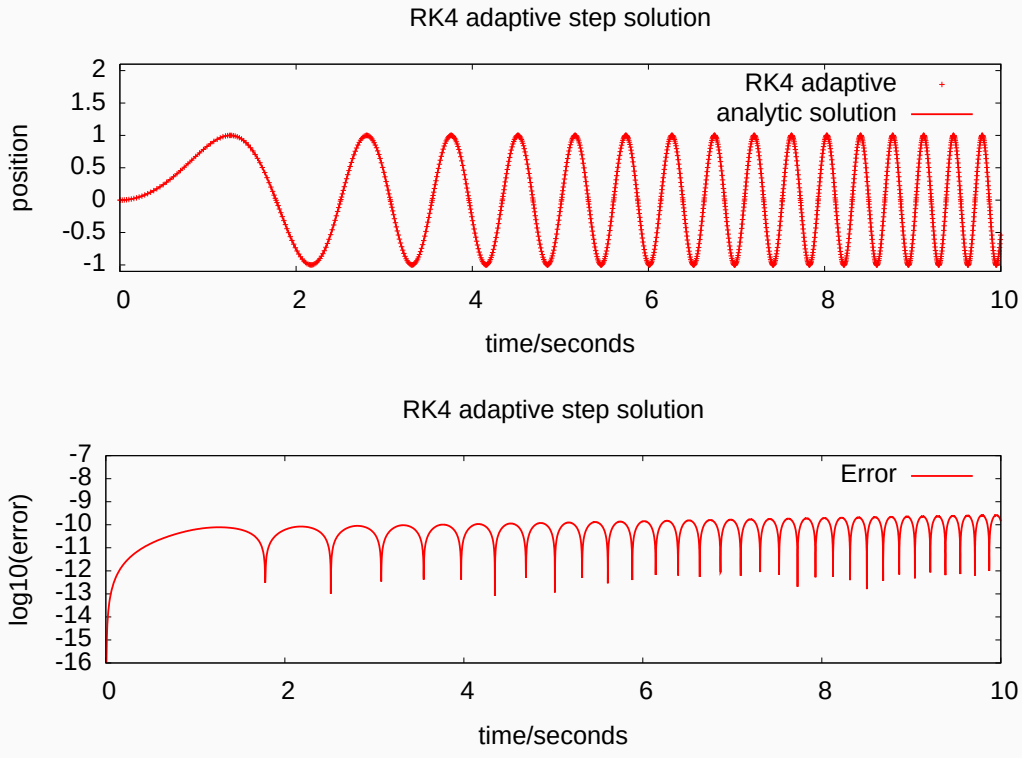
- Take a trial step of size h
- Take two trial steps of length $h/2$

The difference between these two solutions is an estimate of the truncation error in one step E.

For a 4th order Runge-Kutta scheme, the truncation error goes as h^5, so we can estimate the desired step length using

$$h_{\text{pred}} = h \left| \frac{E_0}{E} \right|^{1/5} \tag{1}$$

where E_0 is the desired error.



Adaptive step solution is more accurate and took only 6000 steps; the fixed-step solution took a million. So even though there were more function evaluations per step, it was more efficient.

Other Topics

There is a lot we have not had time to cover:

1. More sophisticated adaptive stepping methods: you will see the method of "Burlischer-Stoer" recommended
2. Shooting methods: where we have boundary conditions at different points.
3. Further theory on stability of various schemes
4. Partial Differential Equations

Beond the level of this course, and not needed for the problems we cover. Look in "Numerical Recipes" and references therein if you want to become an expert.

ODEs: Lessons Learnt

1. Single precision (32-bit) arithmetic is not good enough for serious numerical work.
2. But double precision (64-bit) arithmetic is very good. We got better than 1 part in 10^{14} accuracy with our RK4 integration for our simple test problem.
3. Test your code against simple cases where analytic solutions exist. Run codes over long periods to test for stability. Have you discovered new physics or a numerical issue?
4. If you have no analytic solution for comparison, measure conserved quantities (energy, angular momentum, momentum) to see how well they are conserved in your simulation.
5. Writing your own integrators is instructive, but many years of experience have gone into library routines, so once you understand the basics, use them.
6. There are few hard and fast rules in solving ODEs. Every problem is different, but Runge-Kutta 4th order with adaptive steps is often a reliable algorithm.

ODEs with the Gnu Science Library: ode_ring1.cc

```
// GSL solution to the ODE for a spinning ring in a magnetic field
// d^2 theta/dt^2 = - (2/tau) * sin^2(theta) * d theta/dt
#include <iostream>
#include <cmath>
#include <gsl/gsl_errno.h>
#include <gsl/gsl_odeiv.h>
// Evaluate the derivatives: we work in the transformed variables
// y[0] = theta, y[1] = d(theta)/dt
int calc_derivs(double t, const double y[], double dydx[],
                void *params) {
  // Extract the parameters from the pointer param*.  In this case
  // there is only one - the value tau in the differential equation.
  double tau = *(double *)(params);
  dydx[0] = y[1];
  dydx[1] = -(2.0 / tau) * pow(sin(y[0]), 2) * y[1];
  return GSL_SUCCESS;
}
```

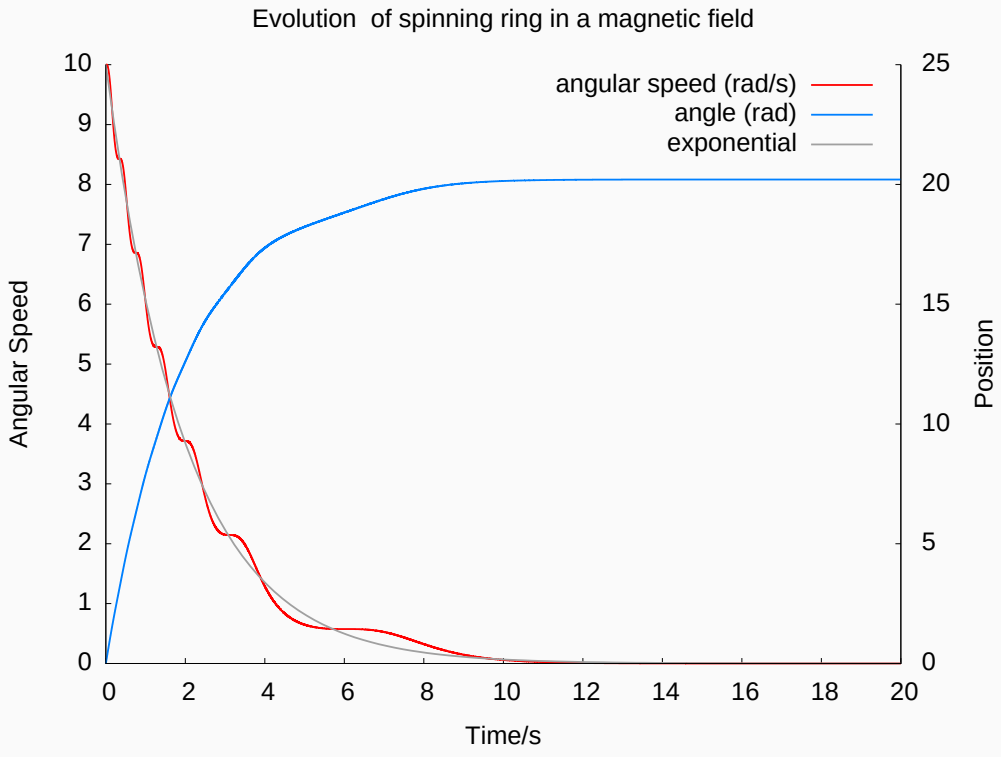

ODEs with the Gnu Science Library: ode_ring1.cc (cont'd)

```
int main() {
  // Initial conditions:
  double tau = 2.0;
  const int n_equations = 2;
  double y[n_equations] = {0, 10};
  double t = 0.0;
  // Create a stepping function:
  gsl_odeiv_step *gsl_step =
      gsl_odeiv_step_alloc(gsl_odeiv_step_rk4, n_equations);
  // Adaptive step control: let's use fixed steps here:
  gsl_odeiv_control *gsl_control = NULL;
  // Create an evolution function:
  gsl_odeiv_evolve *gsl_evolve = gsl_odeiv_evolve_alloc(n_equations);
  // Set up the system needed by GSL: The 4th arg is a pointer
  // to any parameters needed by the evaluator. The 2nd arg
  // points to the jacobian function if needed (it's not needed here).
  gsl_odeiv_system gsl_sys = {calc_derivs, NULL, n_equations, &tau};
  double t_max = 20.0;
  double h = 1e-3;
```

ODEs with the Gnu Science Library: ode_ring1.cc (cont'd)

```
  // Main loop: advance solution until t_max reached.
  while (t < t_max) {
    std::cout << t << " " << y[0] << " " << y[1] << "\n";
    int status = gsl_odeiv_evolve_apply(gsl_evolve, gsl_control,
                                        gsl_step, &gsl_sys, &t,
                                        t_max, &h, y);
    if (status != GSL_SUCCESS) break;
  }
  // Tidy up the GSL objects for neatness:
  gsl_odeiv_evolve_free(gsl_evolve);
  gsl_odeiv_step_free(gsl_step);
  return 0;
}
```


Simple ODE example: GSL result



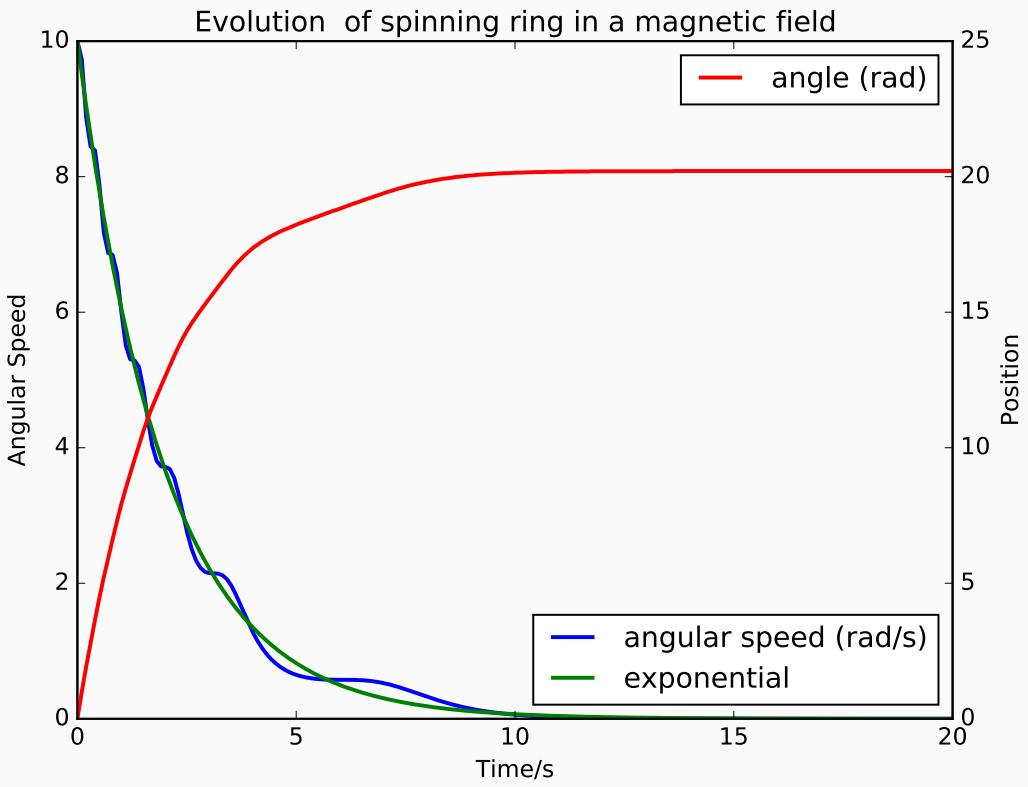
Simple ODE example: scipy code

```
# Solve the ODE for a conducting ring spinning in a magnetic field.
import numpy as np
import scipy.integrate

# This function evaluates the derivatives for the equation
# d^2 theta/dt^2 = - (2/tau) * sin^2(theta) * d theta/dt
# We work in the transformed variables y[0] = theta, y[1] = d(theta)/dt
def derivatives(y,t,tau):
    return [y[1], -(2.0/tau)*np.sin(y[0])**2*y[1]]

# Main code starts here
t=np.linspace(0.0, 20.0, 200)
y0=[0.0, 10.0]
tau=2.0
y=scipy.integrate.odeint(derivatives,y0,t,args=(tau,))
for i in range(len(y)):
    print("{:8.4g} {:8.4g} {:8.4g}".format(t[i],y[i,0],y[i,1]))
```


Simple ODE example: scipy result



Simple ODE example: gnuplot code

```
reset
set term png size 1200,900 font "LiberationSans-Regular" 22
set output 'ode_ring1.png'
set xlabel 'Time/s'
set ylabel 'Angular Speed'
set y2label 'Position'
set y2tics
set title 'Evolution  of spinning ring in a magnetic field'
f='ode_ring1.out'
set border 11
set tics nomirror
tau=2.0
theory(x)=10*exp(-x/tau)
plot f using 1:3 w l lc 8 lw 3 title 'angular speed (rad/s)',\
     f u 1:2 axes x1y2 w l lc -1 lw 3 title 'angle (rad)',\
     theory(x) w l lc 1 lw 2 title 'exponential'
```


Simple ODE example: pyplot code

```
import matplotlib.pyplot as plt
import numpy as np
# Load the data
x,y,dydx=np.loadtxt("ode_ring1_scipy.out",unpack=True)
# Plot angular speed and analytical approximation
fig, ax1 = plt.subplots()
ax1.plot(x,dydx,lw=2,label="angular speed (rad/s)")
ax1.plot(x,10*np.exp(-x/2.0),lw=2,label="exponential")
ax1.set_xlabel("Time/s")
ax1.set_ylabel("Angular Speed")
ax1.set_title("Evolution  of spinning ring in a magnetic field")
ax1.legend(loc="lower right")
# Angular position plotted on same plot with second set of axes
ax2=ax1.twinx()
ax2.plot(x,y,lw=2,label="angle (rad)",color="red")
ax2.set_ylabel("Position")
ax2.legend(loc="upper right")
plt.savefig("ode_ring1_scipy.pdf",bbox_inches="tight",transparent=True)
```

Partial Differential Equations (PDEs)

We spent a lot of time looking at ODEs. We note that the GNU Science Library has extensive support for ODE solution. But there is nothing for PDEs. Why is this?

PDEs come in many different types and there are many methods of solution. Specialised techniques are created for the different types.

Physical applications include

- Wave equation: $\frac{\partial^2 \psi}{\partial t^2} = c^2 \frac{\partial^2 \psi}{\partial x^2}$
- Navier-Stokes equation for fluids
- Laplace's equation
- Diffusion equation

Solution methods include

- Finite Differences
- Fourier methods

Often developed specifically for a particular type of problem.