Ordinary differential equations

The set of ordinary differential equations (ODE) can always be reduced to a set of coupled first order differential equations.

For example, Newton's law is usually written by a second order differential equation $m\ddot{\vec{r}} = F[\vec{r}, \dot{\vec{r}}, t]$. In Hamiltonian dynamics, the same problem leads to the set of first order equations $\dot{\vec{p}} = -\frac{\partial H}{\partial q}$ and $\dot{\vec{q}} = \frac{\partial H}{\partial p}$.

We will therefore concentrate on the system of equations

$$\frac{dy_i}{dx} = f_i(x, y_0, y_1, \cdots, y_{N-1}) \tag{1}$$

To solve the problem, the boundary conditions need to be specified. They can be arbitrary complicated - for example a set of nonlinear equations relating values $y_i(x_l)$ and there derivatives $\dot{y}_i(x_l)$ at certain points x_l.

Boundary conditions can be divided into categories

- Initial value problems (all necessary conditions specified at starting point)
- Two point boundary problems (part of the conditions specified at one point x_0 and the rest at x_1).
- More complicated problems

In this chapter, we will concentrate on the Initial value problems and we will show in the *Density functional theory* chapter how to solve The two points boundary problem - by so called **shooting** method.

We will implement these methods

- Runge-Kutta method : general purpose routine
- Numerov's algorithm: $\ddot{y} = f(t)y(t)$ (for Schroedinger equation)
- Verlet algorithm: $\ddot{y} = F[y(t), t]$ (for molecular dynamics because it is more stable and preserves total energy)

In this chapter, we will concentrate on ”most often” general purpose routine. In subsequent chapers we will implement the two other methods.

The simplest method for solving differential equations is Euler's method

$$y_{i+1} = y_i + hf(x_i, y_i) + O(h^2) \tag{2}$$
$$x_{i+1} = x_i + h \tag{3}$$

but it is not advisable to use it in practice.

When solving differential equation, we usually look for a very smooth function $y(x)$ and in order that the step size can be finite and precision loss is not very dramatic, it is recommended to use higher order routine. In addition, Euler's routine is very unstable because it is not "symmetric".

The derivative $f(x_i, y_i)$ is taken at the beginning of the interval $[x_i, x_i + h]$. The stability and precision would increase if one could estimate derivative in the middle of the interval, i.e., $f(x_i + h/2, y(x_i + h/2))$.

The second order Runge-Kutta method implements the above idea

$$k_1 = hf(x_i, y_i) \tag{4}$$
$$y_{i+1} = y_i + hf(x_i + \frac{1}{2}h, y_i + \frac{1}{2}k_1) + O(h^3) \tag{5}$$

It is called second order, because error is of the order of h^3. In general, the error of the n-th order routine is $O(h^{n+1})$.

Basic Numerical Algorithms

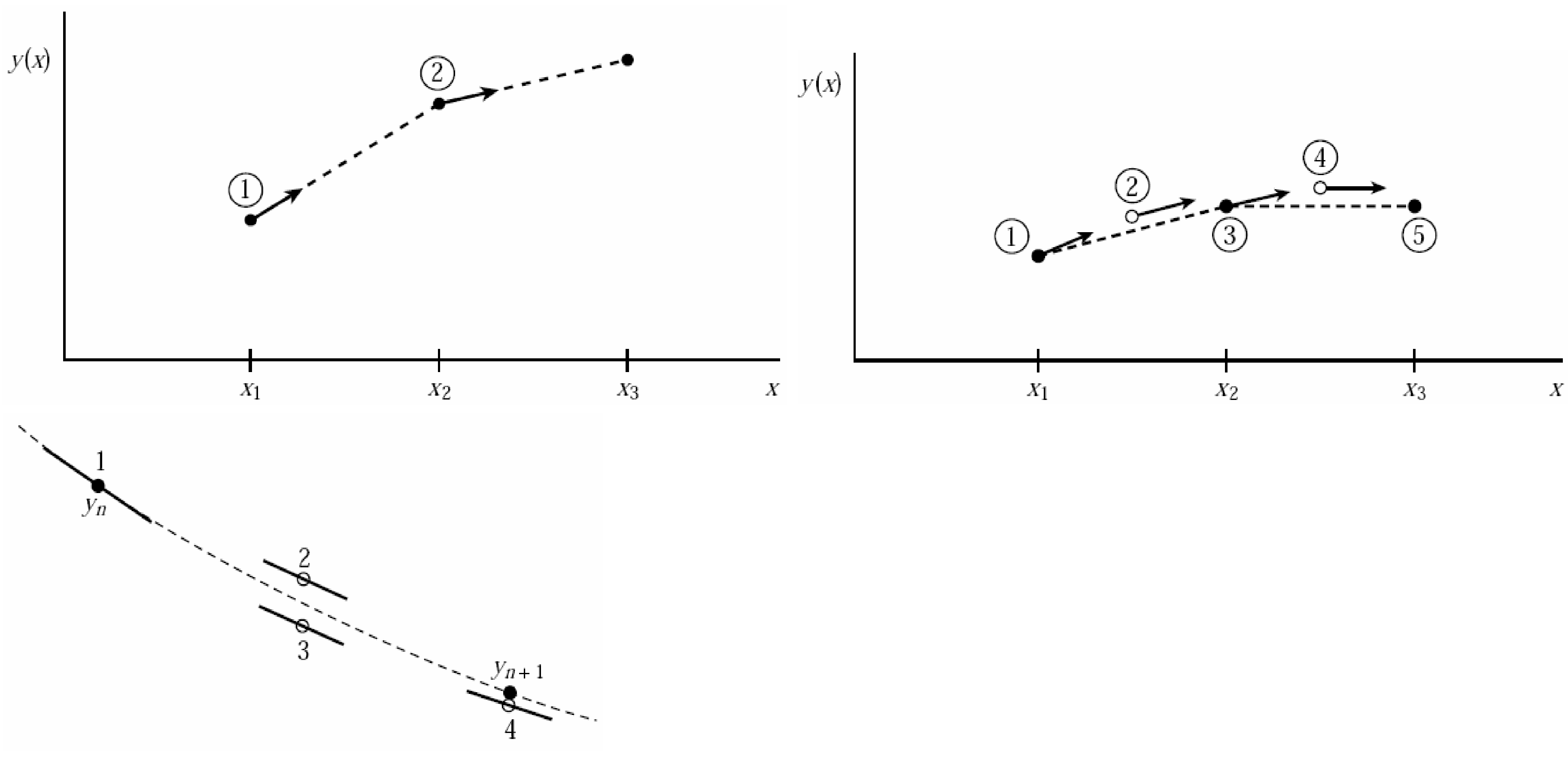


Figure 1: top letf: Euler’s algorithm, top right: Midpoint or Second order Runge-Kutta method bottom: Forth order Runge-Kutta

Most popular is the forth-order Runge Kutta (RK4) method:

$$k_1 = hf(x_i, y_i) \tag{6}$$

$$k_2 = hf(x_i + \frac{1}{2}h, y_i + \frac{1}{2}k_1) \tag{7}$$

$$k_3 = hf(x_i + \frac{1}{2}h, y_i + \frac{1}{2}k_2) \tag{8}$$

$$k_4 = hf(x_i + h, y_i + k_3) \tag{9}$$

$$y_{i+1} = y_i + \frac{1}{6}k_1 + \frac{1}{3}k_2 + \frac{1}{3}k_3 + \frac{1}{6}k_4 + O(h^5) \tag{10}$$

How to understand the method? Looking at the above figure, we see:

- k_1 is the slope at the beginning of the interval;
- k_2 is the slope at the midpoint of the interval, using slope k_1 to determine the value of y at the point $x_i + h/2$ using Euler's method;
- k_3 is again the slope at the midpoint, but now using the slope k_2 to determine the y-value;
- k_4 is the slope at the end of the interval, with its y-value determined using k_3;

- in averaging the four slopes, greater weight is given to the slopes at the midpoint.

The RK4 method is a fourth-order method, meaning that the error per step is on the order of h^5, while the total accumulated error has order h^4. With only four function evaluations, for fourth order accuracy is extremely good.

The code for RK4 is given below:

```
def RK4(x, y, dh, derivs):
    """
    ///////////////// METHOD OF RUNKGE-KUTTA 4-th ORDER   ///////////////////////
    """
    k1 = dh * derivs(y,          x)          # First step : evaluating k1
    k2 = dh * derivs(y + 0.5*k1, x+0.5*dh)   # Second step : evaluating k2
    k3 = dh * derivs(y + 0.5*k2, x+0.5*dh)   # Third step : evaluating k3
    k4 = dh * derivs(y + k3,     x+dh)       # Final step : evaluating k4
    return y + (k1+2.0*(k2+k3)+k4)/6.

def d_simple_pendulum(y, x):
    """ use time units omega*t->t
        d^2 u/dt^2 = - omega^2 u    is written as
        y      = [u(t), du/dt]
        d y/dt = [du/dt, d^2u/dt = -u]
        Exact solution is y[0,1] = [xmax*sin(t), xmax*cos(t)]
    """
    return array([y[1],-y[0]])

t_start=0        # first time
t_stop=200       # elapsed time in dimensionless units
dh=0.1           # time step

ts = linspace(t_start,t_stop, (t_stop-t_start)/dh+1)
dh = ts[1]-ts[0]
y = array([0, 1.]) # Pendulum is initially at zero but has maximum momentum
for i,t in enumerate(ts):
    y = RK4(t, y, dh, d_simple_pendulum)
```

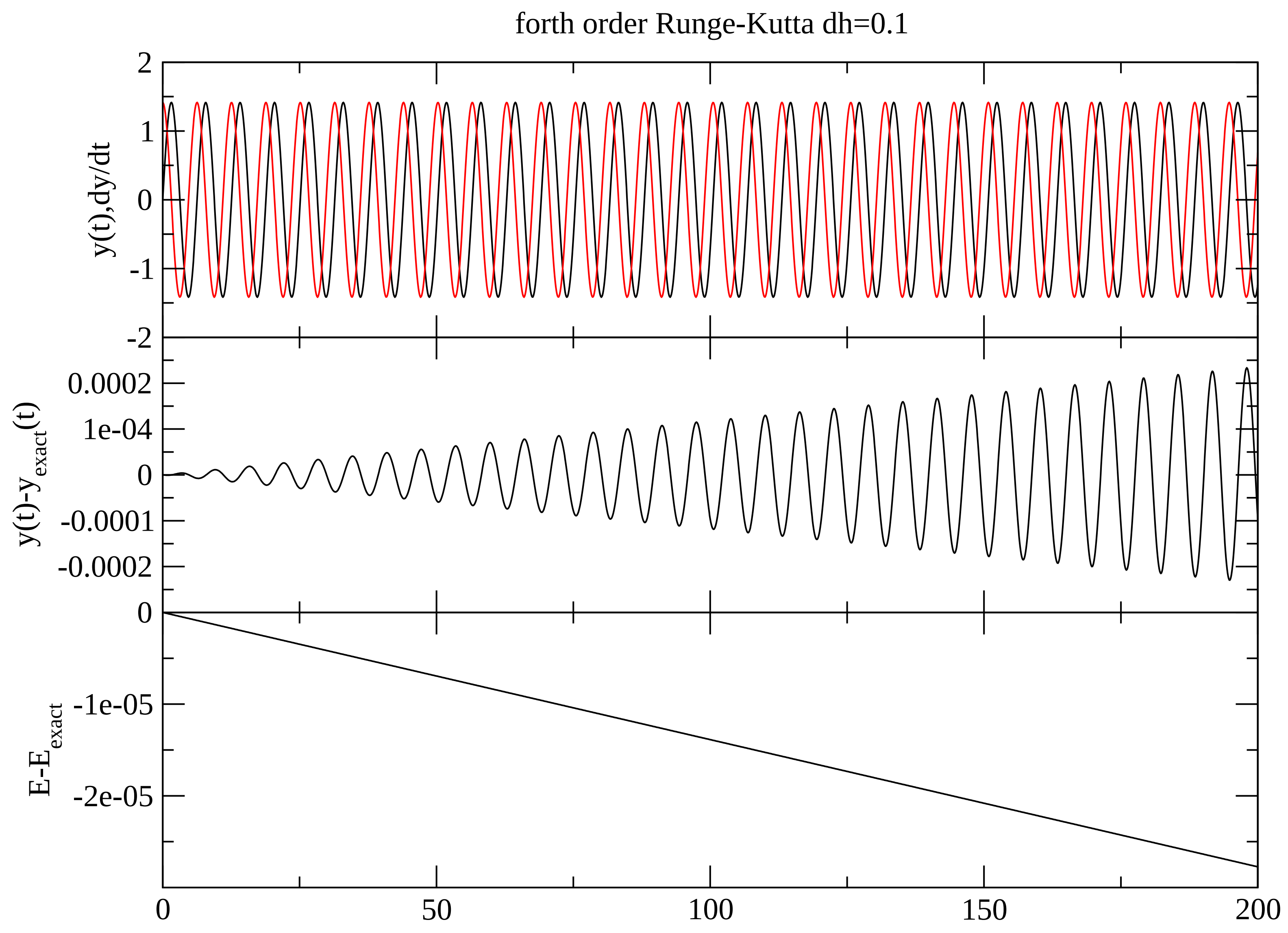


Figure 2: Fixed step Runge Kutta for simple pendulum. The error is increasing linearly with time and energy is being lost linearly with time.

In practice, it is usually better to use adaptive step. In this case, one needs some way of estimating error of each step and decrease the step size if necessary. Two types of algorithms are very popular

- Step doubling
- Embedded methods

Step doubling builds on the fact that when performing half of the step, the error is smaller and by comparing the error of full step and half step, we have good estimation of the error:

$$y(t+2h) = y_1 + (2h)^5\,C \qquad \textit{one big } 2h \textit{ step} \tag{11}$$

$$y(t+2h) = y_2 + 2h^5\,C \qquad \textit{two small } h \textit{ steps} \tag{12}$$

The difference of the two solutions is $0 = y_2 - y_1 - 30h^5C$, which can serve as an error estimate Δ

$$\Delta \equiv y_2 - y_1 \tag{13}$$

We hence have an estimate for the error Δ, and because we know that it is proportional to h^5, we can estimate how much the step should be reduced/increased to reach desired accuracy.

If we make a step h_1 and get an error Δ_1, we can estimate the stepsize which will give the error of the order of Δ_0 is

$$h_0 = h_1 \left| \frac{\Delta_0}{\Delta_1} \right|^{1/5} \tag{14}$$

The solution at current step can even be ”improved” to fifth order by evaluating

$$y(x + 2h) = y_2 + \frac{\Delta}{15} + O(h^6). \tag{15}$$

Embedded Runge-Kutta methods

For 4-th order RK method one needs 4 function evaluations, for higher order accuracy (of order M) one typically needs more function evaluations, namely, M+2.

The method due to Fehldberg needs 6 function evaluations for 5-th order accuracy. In addition, one can use the same 6 function values to get 4-th order accuracy. The difference can therefore be used as an error estimate.

The embedded fifth-order RK formulas are

$$k_0 = hf(x_i, y_i) \tag{16}$$

$$k_1 = hf(x_i + a_1 h, y_i + b_{10} k_0) \tag{17}$$

$$\cdots \tag{18}$$

$$k_5 = hf(x_i + a_5 h, y_i + b_{50} k_0 + \cdots + b_{54} k_4) \tag{19}$$

$$y_{i+1} = y_i + c_0 k_0 + c_1 k_1 + \cdots + c_5 k_5 + O(h^6) \tag{20}$$

$$y'_{i+1} = y_i + c'_0 k_0 + c'_1 k_1 + \cdots + c'_5 k_5 + O(h^5) \tag{21}$$

$$\Delta = y_{i+1} - y'_{i+1} = \sum_{i=0}^{5} (c_i - c'_i) k_i \tag{22}$$

where coefficients are

i	a_i	c_i	c'_i	b_{i0}	b_{i1}	b_{i2}	b_{i3}	b_{i4}	b_{i5}
0		$\frac{37}{378}$	$\frac{2825}{27648}$						
1	$\frac{1}{5}$	0	0	$\frac{1}{5}$					
2	$\frac{3}{10}$	$\frac{250}{621}$	$\frac{18575}{48384}$	$\frac{3}{40}$	$\frac{9}{40}$				
3	$\frac{3}{5}$	$\frac{125}{594}$	$\frac{13525}{55296}$	$\frac{3}{10}$	$-\frac{9}{10}$	$\frac{6}{5}$			
4	1	0	$\frac{277}{14336}$	$-\frac{11}{54}$	$\frac{5}{2}$	$-\frac{70}{27}$	$\frac{35}{27}$		
5	$\frac{7}{8}$	$\frac{512}{1771}$	$\frac{1}{4}$	$\frac{1631}{55296}$	$\frac{175}{512}$	$\frac{575}{13824}$	$\frac{44275}{110592}$	$\frac{253}{4096}$	

(23)

The error estimate we have is for the forth-order value y'_{i+1} and is proportional to h^5 or $\Delta = Ch^5$. If we made a step h_1 and got the error Δ_1, we can figure out what the step needs to be, to get the error of the order of Δ_0

$$h_0 = h_1 \left|\frac{\Delta_0}{\Delta_1}\right|^{1/5} \tag{24}$$

Although the error estimate is for the forth-order value, we obviously accept fifth order estimate y_{i+1}.

The above formula can be used in two ways

- If obtained error Δ_1 (by taking step h_1) is smaller than desired accuracy Δ_0, we can increase step to $h_0 = h_1 \left|\frac{\Delta_0}{\Delta_1}\right|^{1/5}$ when taking the next step.
- If the obtained error is to big, we have to backtrack and take the same step again by choosing smaller step $h_0 = h_1 \left|\frac{\Delta_0}{\Delta_1}\right|^{1/5}$.

Backtracking is "expensive" bacuse we throw away 5-6 function evaluation! To play it save, we rather increases the step a little less that we should and decrease slightly more than we could

$$h_0 = \begin{cases} S h_1 \left|\frac{\Delta_0}{\Delta_1}\right|^{0.2} & \Delta_0 > \Delta_1 \\ S h_1 \left|\frac{\Delta_0}{\Delta_1}\right|^{0.25} & \Delta_0 < \Delta_1 \end{cases} \tag{25}$$

where $S \approx 0.9$ is safety factor.

We need routine to solve a system of coupled equations and therefore y_i is a vector of values, however, we treated Δ as a number. In the code, we take a number Δ to be the largest component of vector Δ_m since error of all components needs to be kept below desired accuracy.

Many times, the components (corresponding to different equations) differ dramatically in value. In many cases, we want to multiply different components with different factors when evaluating error

$$\Delta = \max(\Delta_m/\mathrm{yscal}_m) \tag{26}$$

A good choice for scaling factors is

$$\mathrm{yscal}_m = |y_m| + |f_m h| + 10^{-3} \tag{27}$$

where y_m is m-th component of the vector at each step i and f_m is the derivative at the same step. This ensures that the relative error is bounded rather than absolute.

Below is the Python implementation for the RK5 algorithm. Note that if speed is desired, the code should be rewritten in C++ or fortran. (See source code directory for C++ implementation).

```
def RK5_Try(x, y, dydx, dh, derivs):
    """
    ///////// METHOD OF RUNGE-KUTTA 5-th ORDER, ADAPTIVE STEP  /////////////
    Given values for variables y[...] and their derivatives dydx[...] known at x, use
    the fifth-order Cash-Karp Runge-Kutta method to advance the solution over an interval dh
    and return the incremented variables as yout[..]. Also return an estimate of the local
    truncation error in yout using the embedded fourth-order method. The user supplies the routine
    dydx = derivs(y,x), which returns derivatives dydx at x.
    """
    #           a0  a1   a2   a3   a4   a5
    ai = array([0, 0.2, 0.3, 0.6, 1.0, 0.875])
    #            c0          c1   c2            c3            c4   c5
    ci = array([37.0/378.0, 0.0, 250.0/621.0, 125.0/594.0, 0.0, 512.0/1771.0])
    #              c0-d0                c1-d1  c2-d2                c3-d3                      c4-d4               c5-d5
    dci = array([ci[0]-2825.0/27648.0, 0.0, ci[2]-18575.0/48384.0, ci[3]-13525.0/55296.0, -277.00/14336.0, ci[5]-0.25])
    bs = array([[0.0,            0.0,         0.0,           0.0,              0.0,          0.0],
                [0.2,            0.0,         0.0,           0.0,              0.0,          0.0],
                [3.0/40.0,       9.0/40.0,    0.0,           0.0,              0.0,          0.0],
                [0.3,           -0.9,         1.2,           0.0,              0.0,          0.0],
                [-11.0/54.0,     2.5,        -70.0/27.0,     35.0/27.0,        0.0,          0.0],
                [1631.0/55296.0, 175.0/512.0, 575.0/13824.0, 44275.0/110592.0, 253.0/4096.0, 0.0]])

    k0 = dh*dydx                                                    # first step
    k1 = dh*derivs(y + bs[1,0]*k0, x + ai[1]*dh)                    # Second step.
    k2 = dh*derivs(y + bs[2,0]*k0+bs[2,1]*k1, x + ai[2]*dh)         # Third step.
    k3 = dh*derivs(y + bs[3,0]*k0+bs[3,1]*k1+bs[3,2]*k2, x + ai[3]*dh) # Fourth step.
    k4 = dh*derivs(y + bs[4,0]*k0+bs[4,1]*k1+bs[4,2]*k2+bs[4,3]*k3, x + ai[4]*dh) # Fifth step.
    k5 = dh*derivs(y + bs[5,0]*k0+bs[5,1]*k1+bs[5,2]*k2+bs[5,3]*k3+bs[5,4]*k4, x + ai[5]*dh) # Sixth step.
    # Accumulate increments with proper weights.
    yout = y + ci[0]*k0+ci[2]*k2+ci[3]*k3+ci[5]*k5
    # Estimate error as difference between fourth and fifth order methods.
    yerr = dci[0]*k0+dci[2]*k2+dci[3]*k3+dci[4]*k4+dci[5]*k5
    return (yout, yerr)
```

```
def RK5_Step(x, y, dhtry, derivs, accuracy):
    """ Fifth-order Runge-Kutta step with monitoring of local truncation error to ensure accuracy and adjust stepsize.
      Input is
         x                  -- independent variable
         y[...], dydx[...]  -- the dependent variable vector and its derivative at the starting value of the independent
         dhtry              -- the stepsize to be attempted
         derivs             -- the user-supplied routine that computes the right-hand side derivatives.
       Output:
         x_new, y_new
         dh_next            -- the estimated next stepsize
    """
    dydx = derivs(y, x)  # Calculates derivatives for the new step
    # good way of determining desired accuracy
    yscal = abs(y[:]) + abs(dydx[:]*dhtry) + 1e-3

    dh = dhtry   # Set stepsize to the initial trial value.
    while True:  # infinite loop
        (y_new, yerr) = RK5_Try(x, y, dydx, dh, derivs)  # Take a step.
        errmax = max( abs(yerr/yscal) )/accuracy       # maximum error scaled to required tolerance
        if (errmax <= 1.0): break                      # Step succeeded. Compute size of next step.
        dh_new = 0.9*dh/errmax**0.25                   # Truncation error too large, reduce stepsize.
        if abs(dh_new) < 0.1*abs(dh): # if step might get too small
          dh_new = 0.1*dh             # take at most 10-times smaller step
        dh = dh_new
        if ( x+dh == x): print "ERROR: stepsize underflow in RKStep"

    if errmax < 2.e-4:     # Step is way too small
        dh_next = 5.0*dh   # Increase it 5-times
    else:                  # Step was too small, but of correct order of magnitude
        dh_next = 0.9*dh/errmax**0.2  # Step is too small, increase it next time with the delta^1/5. power

    return (x+dh, y_new, dh_next)
```

```
if __name__ == '__main__':
  t_start=0        # first time
  t_stop=200       # elapsed time in dimensionless units
  dh=0.1           # initial time step
  accuracy=1e-7
  y = array([0, 1.0]) # Pendulum is initially at zero but has maximum momentum

  tc = t_start
  while (tc<=t_stop):
    (tc, y, dh) = RK5_Step(tc, y, dh, d_simple_pendulum, accuracy)
```

Basic Numerical Algorithms

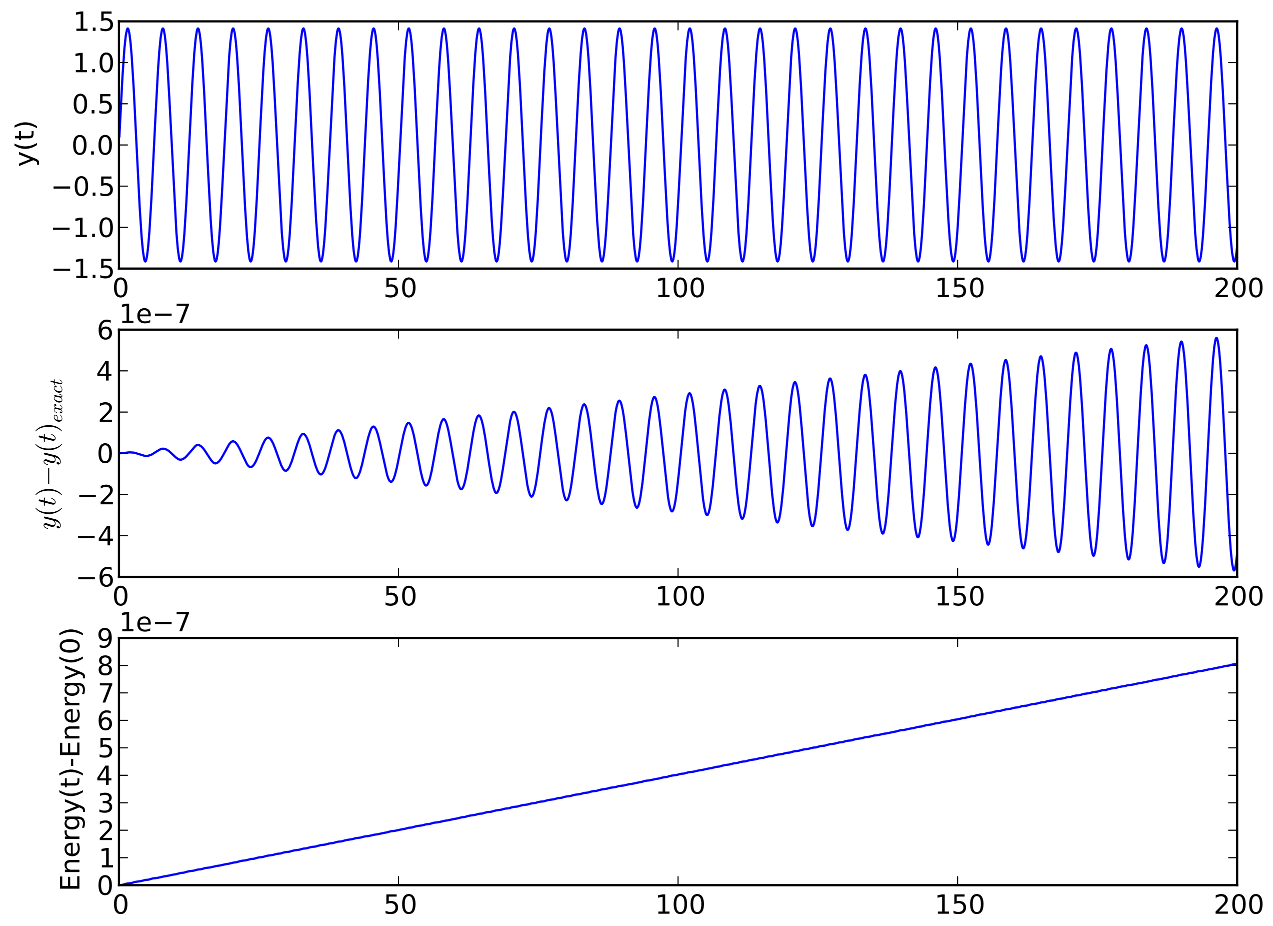
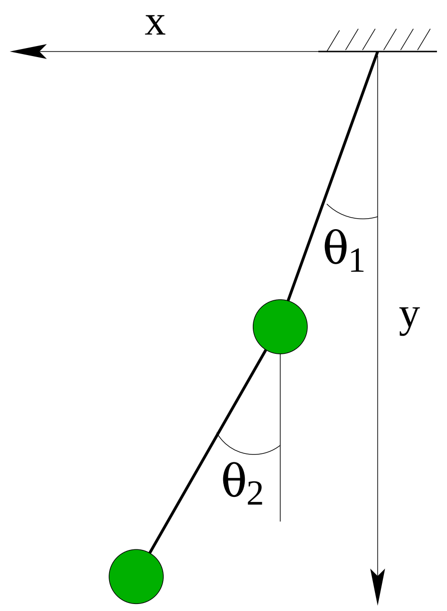


Figure 3: Variable Runge Kutta step for simple pendulum. Precision is here set to 10^{-8}. The error is increasing linearly, but is very small (10^{-6}) after t=200.

A somewhat more interesting example is double pendulum. It is well known that double pendulum is chaotic for not too small energy and the Poincare plots in chaotic regime show nice patterns.



$$\vec{r}_1 = (l\sin\theta_1, l\cos\theta_1) \tag{28}$$

$$\vec{r}_2 = (l(\sin\theta_1 + \sin\theta_2), l(\cos\theta_1 + \cos\theta_2)) \tag{29}$$

$$T = \frac{1}{2}m\dot{\vec{r}}_1^2 + \frac{1}{2}m\dot{\vec{r}}_2^2 \tag{30}$$

$$V = 3mgl - m\vec{g}\vec{r}_1 - m\vec{g}\vec{r}_2 \tag{31}$$

$$L = \frac{1}{2}ml^2[2\dot{\theta}_1^2 + \dot{\theta}_2^2 + 2\cos(\theta_1-\theta_2)\dot{\theta}_1\dot{\theta}_2] - mgl[3 - 2\cos\theta_1 - \cos\theta_2] \tag{32}$$

$$p_1 = \frac{\partial L}{\partial\dot{\theta}_1} = ml^2[2\dot{\theta}_1 + \cos(\theta_1-\theta_2)\dot{\theta}_2] \tag{33}$$

$$p_2 = \frac{\partial L}{\partial\dot{\theta}_2} = ml^2[\dot{\theta}_2 + \cos(\theta_1-\theta_2)\dot{\theta}_1] \tag{34}$$

$$H = \frac{1}{2ml^2}\frac{[p_1^2 + 2p_2^2 - 2p_1p_2\cos(\theta_1-\theta_2)]}{1+\sin^2(\theta_1-\theta_2)} + mgl[3 - 2\cos\theta_1 - \cos\theta_2] \tag{35}$$

(36)

$$\dot\theta_1 = \frac{\partial H}{\partial p_1} = \frac{p_1 - p_2\cos(\theta_1-\theta_2)}{ml^2[1+\sin^2(\theta_1-\theta_2)]} \tag{37}$$

$$\dot\theta_2 = \frac{\partial H}{\partial p_2} = \frac{2p_2 - p_1\cos(\theta_1-\theta_2)}{ml^2[1+\sin^2(\theta_1-\theta_2)]} \tag{38}$$

$$\dot p_1 = -\frac{\partial H}{\partial \theta_1} = -2mgl\sin\theta_1 - C_1 + C_2 \tag{39}$$

$$\dot p_2 = -\frac{\partial H}{\partial \theta_2} = -mgl\sin\theta_2 + C_1 - C_2 \tag{40}$$

$$C_1 = \frac{p_1p_2\sin(\theta_1-\theta_2)}{ml^2[1+\sin^2(\theta_1-\theta_2)]} \tag{41}$$

$$C_2 = \frac{[p_1^2 + 2p_2^2 - 2p_1p_2\cos(\theta_1-\theta_2)]\sin(2(\theta_1-\theta_2))}{2ml^2[1+\sin^2(\theta_1-\theta_2)]^2} \tag{42}$$

$$\widetilde{p} = \frac{p}{ml^2\omega_0} \tag{43}$$

$$\widetilde{t} = t\omega_0 \tag{44}$$

$$\omega_0^2 = \frac{g}{l} \tag{45}$$

$$\widetilde{p} \to p; \widetilde{t} \to t \tag{46}$$

$$\dot{\theta}_1 = \frac{p_1 - p_2\cos(\theta_1-\theta_2)}{1+\sin^2(\theta_1-\theta_2)} \tag{47}$$

$$\dot{\theta}_2 = \frac{2p_2 - p_1\cos(\theta_1-\theta_2)}{1+\sin^2(\theta_1-\theta_2)} \tag{48}$$

$$\dot{p}_1 = -2\sin\theta_1 - C_1 + C_2 \tag{49}$$

$$\dot{p}_2 = -\sin\theta_2 + C_1 - C_2 \tag{50}$$

$$C_1 = \frac{p_1p_2\sin(\theta_1-\theta_2)}{1+\sin^2(\theta_1-\theta_2)} \tag{51}$$

$$C_2 = \frac{[p_1^2+2p_2^2-2p_1p_2\cos(\theta_1-\theta_2)]\sin(2(\theta_1-\theta_2))}{2[1+\sin^2(\theta_1-\theta_2)]^2} \tag{52}$$

The class which can be given to integration routine is

```
def d_DoublePendulum(y, x):
    """
     theta1 -> y[0]
     theta2 -> y[1]
     p1     -> y[2]
     p2     -> y[3]
   """
    t1,t2 = y[0],y[1]
    p1,p2 = y[2],y[3]
    cs = cos(t1-t2)
    ss = sin(t1-t2)
    tt = 1./(1+ss**2)
    c1 = p1*p2*ss*tt
    c2 = (p1**2+2*p2**2-2*p1*p2*cs)*cs*ss*tt**2
    return array([ (p1-p2*cs)*tt,  (2*p2-p1*cs)*tt, -2*sin(t1)-c1+c2, -sin(t2)+c1-c2])

def Energy_DoublePendulum(y):
    t1,t2 = y[0],y[1]
    p1,p2 = y[2],y[3]
    cs = cos(t1-t2)
    ss = sin(t1-t2)
    tt = 1./(1+ss**2)
    return 0.5*(p1**2 + 2*p2**2 - 2*p1*p2*cs)*tt + (3.-2.*cos(t1)-cos(t2))
```

Poincare map From Wikipedia

In mathematics of dynamical systems, a Poincare map or Poincare section, named after Henri Poincare, is the intersection of a trajectory which moves periodically (or quasi-periodically, or chaotically), in a space of at least three dimensions, with a transversal hypersurface of one fewer dimension. More precisely, one considers a trajectory with initial conditions on the hyperplane and observes the point at which this trajectory returns to the hyperplane. The Poincare section refers to the hyperplane, and the Poincare map refers to the map of points in the hyperplane induced by the intersections.

Basic Numerical Algorithms

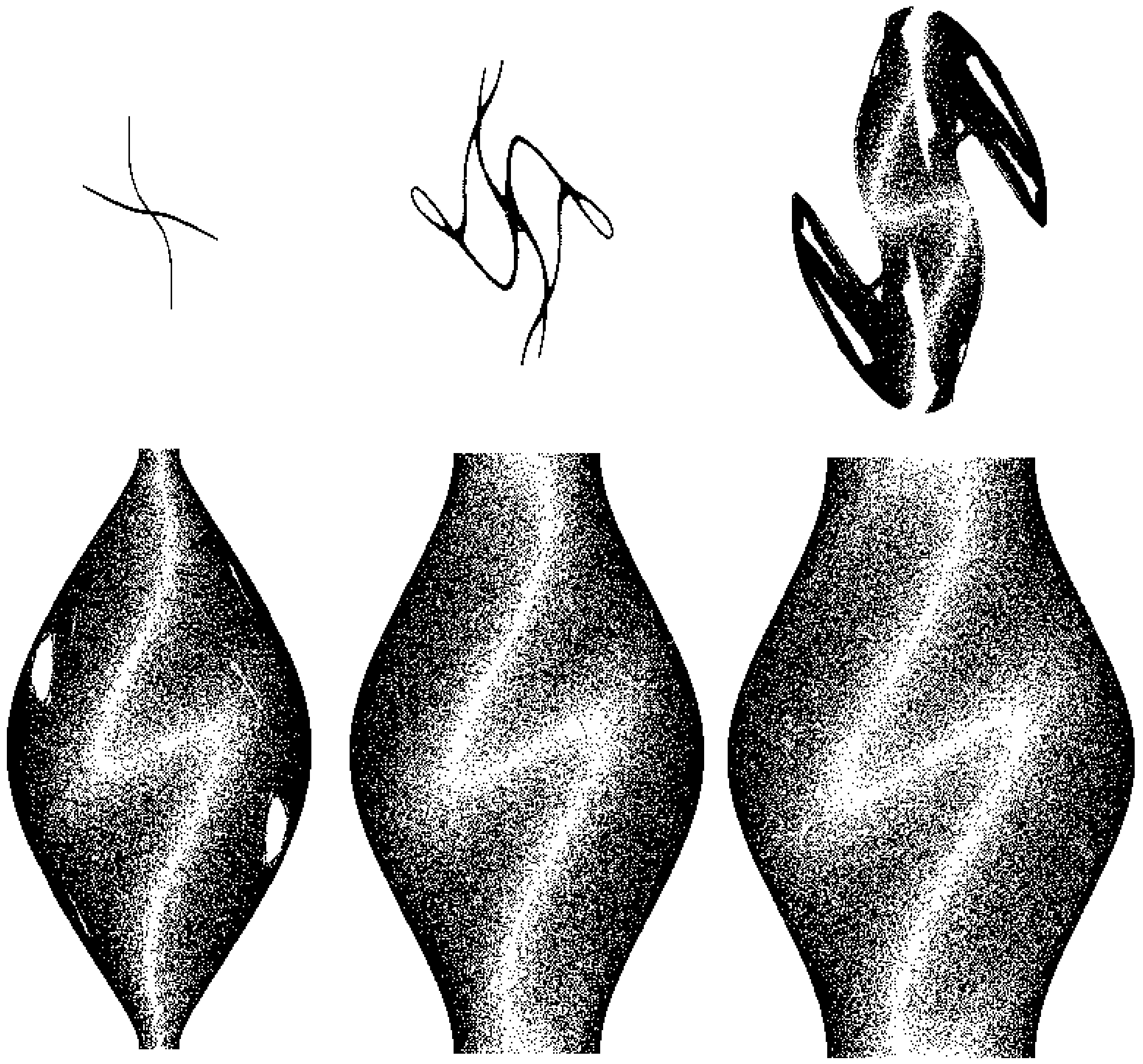


Figure 4: Poincare plots using θ_1 and θ_2 as variables. The point is plotted when $p_{\theta_1} = 0$. Energies used to get the above plots are 5,10,15,20,25,30.

Homeworks

1 Simulate the motion of Earth in the solar system as a two body problem (taking into account only the Sun and Earth). Write the equation

$$m\ddot{\vec{r}} = -GmM\frac{\vec{r}}{r^3}$$

in dimensionless units or atronomical units (AU).

2 Simulate the three body problem and check how strong is the influence of Jupiter on motion of Earth.

3 Plot trajectories of Earth in case Jupiter's mass is 1000 times bigger than its actual mass.

4 Verify the existance of Kirkwood gaps. Simulate motion of Jupiter together with asteroids close to 2/1 Kirkwood gap with the following initial conditions

$Object$	$Radius(AU)$	$Velocity(AU/yr)$
$Jupiter$	5.2	2.755
$Asteroid1$	3.0	3.628
$Asteroid2$	3.276	3.471
$Asteroid3$	3.7	3.267

(53)

astronomical units (AU) are:

- length is meassured in units of distance between Earth and Sun $\approx 1.5\,10^{11}$
- time can be meassured in years