# Runge-Kutta Methods

## Runge-Kutta Methods

The aforementioned Euler's method is the simplest single step ODE solving method, but has a fairly large error. The Runge-Kutta methods are more popular due to their improved accuracy, in particular 4th and 5th order methods.

### Outline of the Derivation

The idea behind Runge-Kutta is to perform integration steps using a weighted average of Euler-like steps. The following outline {% cite efferson-numerical-methods %} is not a full derivation of the method, as this requires theorems outside the scope of this course.

#### Second Order Runge-Kutta

We shall start by looking at second order Runge-Kutta methods. We want to solve an ODE of the form

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

on the interval $[x_i, x_{i+1}]$, where $x_{i+1} = x_i + h$, with a given initial condition $y(x = x_i) = y_i$. That is we wish to determine the value of $y(x_{i+1}) = y_{i+1}$. We start by calculating the gradient of $y$ at 2 places:

- The start of the interval: $(x_i, y_i)$
- A point inside the interval, for which we approximate the $y$ value using Euler's method: $(x_i + \alpha h, y_i + \alpha h f(x_i, y_i))$, for some choice of $\alpha$.

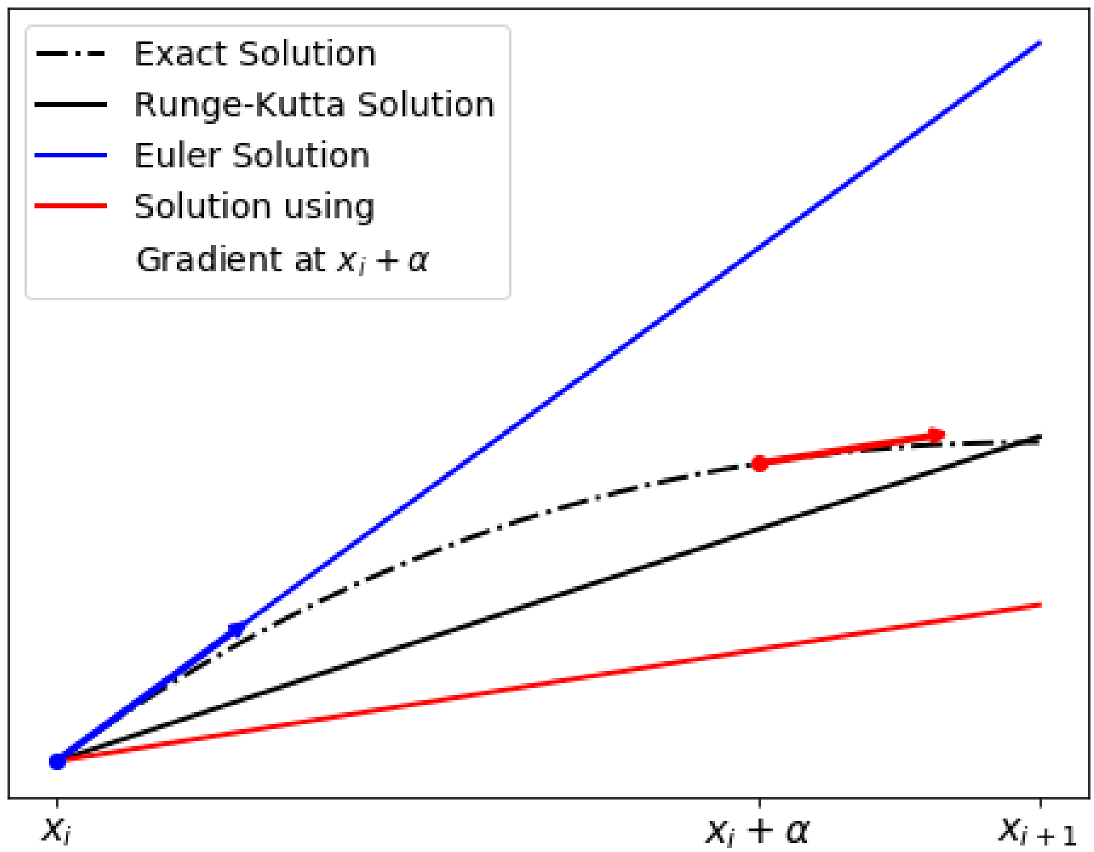
We then approximate the value of $y_{i+1}$ using Euler's method with each of these gradients:

- $y_{i+1} \approx y_i + hf(x_i, y_i)$
- $y_{i+1} \approx y_i + hf(x_i + \alpha h, y_i + \alpha h f(x_i, y_i))$

The final approximation of $y_{i+1}$ is calculated by taking a weighted average of these two approximations:

$$y_{i+1} \approx y_i + c_1 h f(x_i, y_i) + c_2 h f(x_i + \alpha h, y_i + \alpha h f(x_i, y_i))$$

where $c_1 + c_2 = 1$ is required.




Now, how do we go about choosing good values for $c_1$, $c_2$ and $\alpha$? If we Taylor expand the left-hand side of the equation above, and the last term on the right-hand side gives us the relation:

$$\alpha = \frac{1}{2c_2}$$

This still gives us a free choice of one of the parameters. Two popular choices are:

**The trapezoid rule:** $c1 = c2 = \frac{1}{2}$ and $\alpha = 1$, which yields:

$$y_{i+1} = y_i + \tfrac{1}{2}h\left[f(x_i, y_i) + f(x_i + h, y_i + hf(x_i, y_i))\right]$$

**The midpoint rule:** $c1 = 0$, $c2 = 1$ and $\alpha = \frac{1}{2}$, which yields:

$$y_{i+1} = y_i + hf\left(x_i + \tfrac{1}{2}h, y_i + \tfrac{1}{2}hf(x_i, y_i)\right)$$

Both of these methods have an accumulated error of $(h^2)$, as opposed to Euler's method with $(h)$

### Fourth Order Runge-Kutta (RK4)

As mentioned, the more popular Runge-Kutta method is the fourth order (for which we will not cover the derivation):

$$y_{i+1} = y_i + \frac{1}{6}h\ (k_1 + 2k_2 + 2k_3 + k_4)$$

where the $k$ values are the slopes:

$$\begin{aligned} k_1 &= f(x_i, y_i) \\ k_2 &= f\left(x_i + \tfrac{1}{2}h, y_i + \tfrac{1}{2}hk_1\right) \\ k_3 &= f\left(x_i + \tfrac{1}{2}h, y_i + \tfrac{1}{2}hk_2\right) \\ k_4 &= f(x_i + h, y_i + k_3) \end{aligned}$$

$k_1$ is gradient value at the left of the interval. $k_2$ is the gradient at the midpoint of the interval, approximated using $k_1$. The $k_3$ value is the gradient at the midpoint of the interval using $k_2$ to approximate it. $k_4$ is the value of the gradient at the right end of the interval using $k_3$ to approximate it.

This method has an accumulated error of $(h^4)$

#### Worked Example

Consider the ordinary differential equation:

$$\frac{dy}{dx} = \frac{1}{1 + x^2}$$

with the initial condition $y = 1$ at $x = 0$.

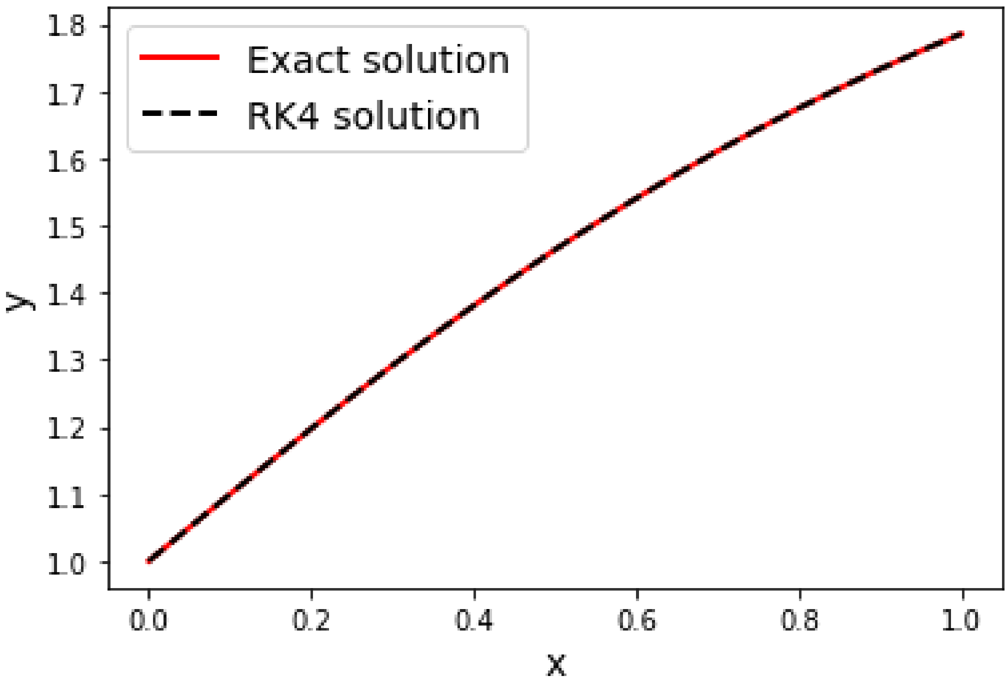
This has the exact solution:

$$y = 1 + \arctan(x)$$

which we can compare are results to.

[7]:
```python
import numpy as np
import matplotlib.pyplot as plt

x0, y0 = 0, 1 #initial conditions
h = 0.05
x_end = 1

#Differential equation
def f(x, y):
    return 1/(1 + x*x)

#Exact solution
```

```python
def y_exact(x):
    return 1 + np.arctan(x)

#Constructing the arrays:
x_arr = np.arange(x0, x_end + h, h) #make sure it goes up to and including x_end

y_arr = np.zeros(x_arr.shape)
y_arr[0] = y0

#Runge-Kutta method
for i,x in enumerate(x_arr[:-1]):

    #k values
    k1 = f(x, y_arr[i])
    k2 = f(x + 0.5*h, y_arr[i] + 0.5*h*k1)
    k3 = f(x + 0.5*h, y_arr[i] + 0.5*h*k2)
    k4 = f(x + h, y_arr[i] + k3)

    #update
    y_arr[i+1] = y_arr[i] + h/6*(k1 + 2*k2 + 2*k3 + k4)


#Plotting the solution
fig, ax = plt.subplots()

ax.plot(x_arr, y_exact(x_arr), '-r', label = 'Exact solution', linewidth = 2)
ax.plot(x_arr, y_arr, '--k', label = 'RK4 solution', linewidth = 2)
ax.set_xlabel('x', fontsize = 14)
ax.set_ylabel('y', fontsize = 14)

ax.legend(fontsize = 14)

plt.show()
```




**High Order ODEs**

As we have discussed in a previous page, higher order ODEs can be reduced to a collection of coupled first order ODEs, for example:

$$\frac{dy_0}{dx} = f_0(x, y_0, y_1, \ldots, y_{n-1}) \tag{1}$$
$$\frac{dy_1}{dx} = f_1(x, y_0, y_1, \ldots, y_{n-1}) \tag{2}$$
$$\frac{dy_2}{dx} = f_2(x, y_0, y_1, \ldots, y_{n-1}) \tag{3}$$
$$\vdots \tag{4}$$
$$\frac{dy_{n-1}}{dx} = f_{n-1}(x, y_0, y_1, \ldots, y_{n-1}) \tag{5}$$

As we have seen, the Euler's method solution for this is fairly simple. For the RK4 method, things are slightly more complicated. We must decide how to calculate the $k$ values.

$$y_{j,i+1} = y_{j,i} + \tfrac{h}{6}(k_{1,j} + 2k_{2,j} + 2k_{3,j} + k_{4,j})$$

Note that the $y_j$ variables are not explicitly dependent on each other, but on the independant variable $x$. Thus we do not have free choice over which $y_j$ values to use when examining another for a particular value of $x$. For any change in $x$, we expect simultenous change in all of the $y_j$. For this reason, when calculating the $k_j$ values for a particular $y_j$, we need to consider the changes in the other $y_l$.

$$
\begin{array}{llllllll}
k_{1,j} & = f_j(x_i, & y_{0,i}\ , & \ldots, & y_{j,i}\ , & \ldots, & y_{n-1,i}\ ) \\
k_{2,j} & = f_j\big(x_i + \frac{1}{2}h, & y_{0,i}\ + \frac{1}{2}k_{1,0}\ , & \ldots, & y_{j,i}\ + \frac{1}{2}k_{1,j}, & \ldots, & y_{n-1,i}\ + \frac{1}{2}k_{1,n-1}\ \big) \\
k_{3,j} & = f_j\big(x_i + \frac{1}{2}h, & y_{0,i}\ + \frac{1}{2}k_{2,0}\ , & \ldots, & y_{j,i}\ + \frac{1}{2}k_{2,j}, & \ldots, & y_{n-1,i}\ + \frac{1}{2}k_{2,n-1}\ \big) \\
k_{4,j} & = f_j(x_i + h, & y_{0,i}\ + k_{3,0}\ , & \ldots, & y_{j,i}\ + k_{3,j}, & \ldots, & y_{n-1,i}\ + k_{3,n-1})
\end{array}
$$

This looks more complicated then it is to apply in practice. All we need to do is vectorize the solution, as on the previous page. We can represent all the $y_j$ as a vector $\vec{y}$, i.e

$$
\vec{y} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{pmatrix}
$$

the ODE can thus be represented as:

$$
\frac{d\vec{y}}{dx} = \vec{f}(x, \vec{y}) = \begin{pmatrix} f_0(x, \vec{y}) \\ f_1(x, \vec{y}) \\ \vdots \\ f_{n-1}(x, \vec{y}) \end{pmatrix}
$$

and an update step as:

$$
\vec{y}_{i+1} = \vec{y}_i + \tfrac{1}{6}h(\vec{k_1} + 2\vec{k_2} + 2\vec{k_3} + \vec{k_4})
$$

where:

$$
\vec{k_m} = \begin{pmatrix} k_{0,m} \\ k_{1,m} \\ \vdots \\ k_{n-1,m} \end{pmatrix}
$$

Note that we can write:

$$
\begin{pmatrix} y_{0,i} & + \frac{1}{2}hk_{1,0} \\ & \vdots \\ y_{j,i} & + \frac{1}{2}hk_{1,j} \\ & \ldots \\ y_{n-1,i} & + \frac{1}{2}hk_{1,n-1} \end{pmatrix} = \vec{y}_i + \tfrac{1}{2}h\vec{k1}
$$

with this in mind, we can simply write the $k$ values as:

$$
\begin{aligned}
\vec{k_1} &= \vec{f}(x_i, \vec{y_i}) \\
\vec{k_2} &= \vec{f}\left(x_i + \tfrac{1}{2}h, \vec{y_i} + \tfrac{1}{2}h\ \vec{k_1}\right) \\
\vec{k_3} &= \vec{f}\left(x_i + \tfrac{1}{2}h, \vec{y_i} + \tfrac{1}{2}h\ \vec{k_2}\right) \\
\vec{k_4} &= \vec{f}\left(x_i + h, \vec{y_i} + h\ \vec{k_3}\right)
\end{aligned}
$$

**Worked Example** Consider the third order differential equation:

$$\frac{d^4 y}{dx^4} = -12xy - 4x^2 \frac{dy}{dx}$$

with the initial conditions: $y(x=0) = 0$, $y'(0) = 0$ and $y''(0) = 2$.

This has an exact solution of:

$$y(x) = e^{-x^2}$$

which we shall use to test our numerical result.

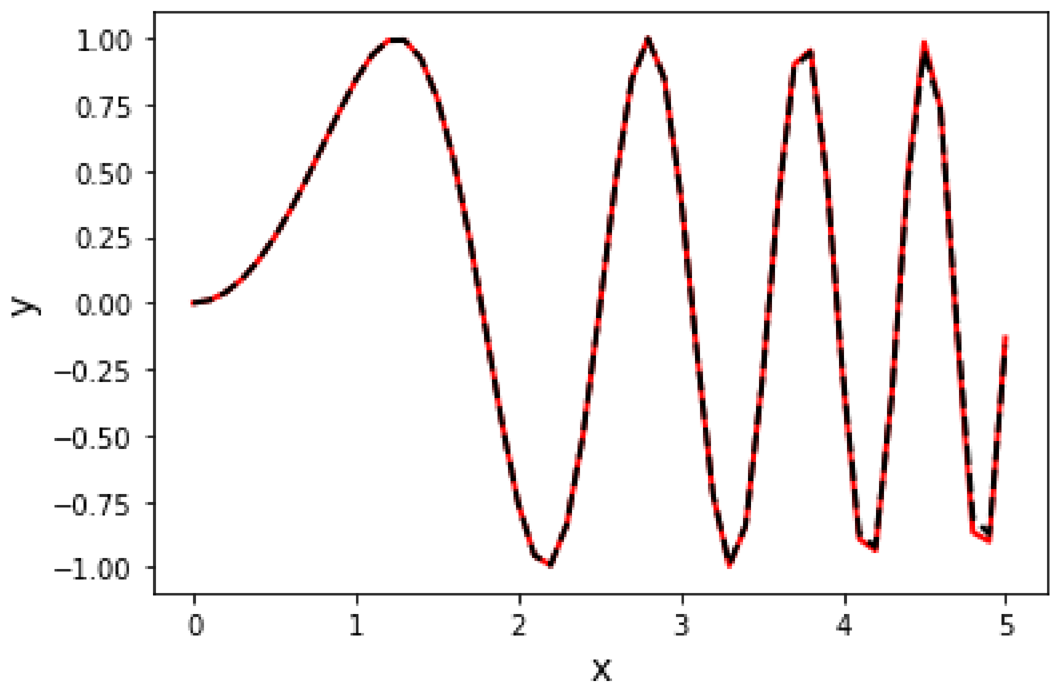
We shall solve this up to $x = 5$ with steps of size $h = 0.1$.

First we reduce this to a system of first order equations by introducing the variables $y_0(x) = y(x)$, $y_1(x) = y'(x)$ and $y_2(x) = y''(x)$:

$$
\begin{aligned}
\frac{dy_0}{dx} &= y_1 \\
\frac{dy_1}{dx} &= y_2 \\
\frac{dy_2}{dx} &= -12xy_0 - 4x^2 y_1
\end{aligned}
$$

[25]:
```
import numpy as np
import matplotlib.pyplot as plt

x0, y0 = 0, [0, 0, 2] #initial conditions
h = 0.1
x_end = 5

def f(x, y):
    #Important! This must return an array!
    return np.array([
        y[1],
        y[2],
        -12*x*y[0] - 4*x*x*y[1]
    ])
```

```
def y_exact(x):
    return np.sin(x*x)

#Constructing the arrays:
x_arr = np.arange(x0, x_end + h, h) #make sure it goes up to and including x_end

y_arr = np.zeros((x_arr.size, len(y0)))
y_arr[0, :] = y0

#Runge-Kutta method
for i,x in enumerate(x_arr[:-1]):
    y = y_arr[i,:]

    #k values
    k1 = f(x, y)
    k2 = f(x + 0.5*h, y + 0.5*h*k1)
    k3 = f(x + 0.5*h, y + 0.5*h*k2)
    k4 = f(x + h, y + h*k3)

    #update
    y_arr[i+1, :] = y + h/6*(k1 + 2*k2 + 2*k3 + k4)


#Plotting the solution
fig, ax = plt.subplots()

ax.plot(x_arr, y_exact(x_arr), 'r-', linewidth = 2)
ax.plot(x_arr, y_arr[:, 0], 'k--', linewidth = 2)
ax.set_xlabel('x', fontsize = 14)
ax.set_ylabel('y', fontsize = 14)

plt.show()
```




## References

{% bibliography –cited %}