Runge Kutta (RK) Methods for IVPs

- Recall that our goal is to now find methods (other than those obtained by keeping more terms in the Taylor series) which give more accurate results than forward Euler for our IVP

**Initial Value Problem (IVP)**

\[ \frac{dy}{dt} = f(t, y) \quad t_0 < t < T \]

\[ y(t_0) = y_0 \]

- Recall that to derive higher order Taylor series method we had to repeatedly
differentiate \( f(t, y) \).

- The Runge Kutta (RK) methods are a family of methods that do not require differentiating \( f(t, y) \).

- Recall that in Euler’s Method we simply use the solution at \( t^n \) and \( f \) evaluated here to estimate the solution at \( t^{n+1} \).

- The basic idea in RK is that we will sample \( f \) at several judiciously chosen points in \([t^n, t^{n+1})\) and use this information to more accurately estimate the solution at \( t^{n+1} \).
Midpoint Method

- The simplest RK is the midpoint method where we estimate the slope (i.e., $f$) at the midpoint of $[t^n, t^{n+1}]$ and then use this to estimate the solution at $t^{n+1}$ using Euler’s method.

- To estimate the slope at the midpoint $t^{n+\frac{1}{2}} = t^n + \frac{\Delta t}{2}$ we take $f$ evaluated at $t^{n+\frac{1}{2}}$ and an estimate to the solution at $t^{n+\frac{1}{2}}$, i.e.,

$$f(t^n + \frac{\Delta t}{2}, Y^{n+\frac{1}{2}}).$$

We don’t have $Y^{n+\frac{1}{2}}$ but as an estimate we take an Euler step of length $\frac{\Delta t}{2}$.

- Recall that the solution at $t^{n+1}$ predicted by Euler’s method is

$$Y^{n+1} = Y^n + \Delta tf(t^n, Y^n)$$

so that we approximate $Y^{n+\frac{1}{2}}$ by

$$Y^n + \frac{\Delta t}{2}f(t^n, Y^n).$$
• Now we estimate the solution at $t^{n+1}$ by Euler’s method where we use the approximation to the slope at $t^{n+\frac{1}{2}}$. We obtain

$$Y^{n+1} = Y^n + \Delta t f \left( t^{n+\frac{1}{2}}, Y^n + \frac{\Delta t}{2} f(t^n, Y^n) \right)$$

• The standard way that this method is written is

$$k_1 = \Delta t f(t^n, Y^n)$$

$$k_2 = \Delta t f(t^n + \frac{\Delta t}{2}, Y^n + \frac{1}{2} k_1)$$

$$Y^{n+1} = Y^n + k_2$$
Example

Approximate the solution to the IVP

\[
\frac{dy}{dt} = t + y \quad y(0) = 2
\]
at \( T = 1 \) using 5 equal timesteps in the Midpoint RK method. Here \( f(t, y) = t + y \).

If we follow the steps of the algorithm above we have as an approximation to \( y(\Delta t) = y(0.2) \)

\[
k_1 = \Delta t f(t^0, Y^0) = .2 f(0, 2) = .2(0 + 2) = .4
\]

\[
k_2 = \Delta t f\left(\frac{\Delta t}{2}, Y^0 + \frac{1}{2} k_1\right) = .2 f(.1, 2 + .5(.4)) = .2 f(.1, 2.2) = .2(1.1 + 2.2) = .46
\]

\[
Y^1 = Y^0 + k_2 = 2 + .46 = 2.36
\]

The actual error here is 0.00420827.

Continuing in this manner we get the following table of results:
If we compare this error with the results from Euler’s Method we can see that it is much better. In the plot below the RK and the exact solution lie almost on top of each other whereas Euler’s method starts to deviate quickly.
Deriving a RK Method to Get a Second Order Scheme

- In the Midpoint Method we chose an “easy” point in the interval \([t^n, t^{n+1}]\), i.e., the midpoint \(t^{n+\frac{1}{2}}\). We could derive the error estimate for this method but we are going to do something more general.

- We are going to take the approach that instead of using the midpoint \(t^{n+\frac{1}{2}}\), we are going to find the point in \([t^n, t^{n+1}]\) which gives us the most accurate method.

- Here we are deriving a two-stage RK method since we are using information at \(t^n\) and at another point in our interval \([t^n, t^{n+1}]\).

- This is the way that most useful RK schemes are derived.

- Recall that the midpoint rule was written as

\[
k_1 = \Delta t f(t^n, Y^n)
\]

\[
k_2 = \Delta t f(t^n + \frac{\Delta t}{2}, Y^n + \frac{1}{2}k_1)
\]
\[ Y^{n+1} = Y^n + k_2 \]

Instead of using the midpoint we will use an arbitrary point \( t^n + \alpha \Delta t \), for \( 0 \leq \alpha \leq 1 \). Likewise instead of evaluating \( f \) in the second step at \( y = Y^n + \frac{1}{2} k_1 \) we will use \( Y^n + \beta k_1 \). Moreover instead of setting \( Y^{n+1} = Y^n + 0 \cdot k_1 + 1 \cdot k_2 \) we will use the general expression \( Y^{n+1} = Y^n + a k_1 + b k_2 \). We have

\[
\begin{align*}
k_1 &= \Delta t f(t^n, Y^n) \\
k_2 &= \Delta t f(t^n + \alpha \Delta t, Y^n + \beta k_1) \\
Y^{n+1} &= Y^n + a k_1 + b k_2
\end{align*}
\]

- Our goal is the find \( \alpha, \beta, a, b \) so that the method is as accurate as we can achieve; in this case it is second order.

- To do this we return to our Taylor series expansion with remainder for \( y(t^n + \Delta t) \)

\[
y(t^{n+1}) = y(t^n) + y'(t^n) \Delta t + y''(t^n) \frac{\Delta t^2}{2} + y'''(\xi_n) \frac{\Delta t^3}{6}
\]

- Now we want to take this expansion and subtract our expansion for \( Y^{n+1} \) to get the highest power of \( \Delta t \), i.e., the highest order method possible with only using information from two points.
To do this we would like to relate the derivatives of $y(t)$ to $f(t, y)$. Clearly $y'(t) = f(t, y)$ but what about $y''(t)$? We know that

$$y''(t) = \frac{d}{dt} y'(t) = \frac{d}{dt} f(t, y)$$

and since $f$ is a function of both $t$ and $y(t)$ we use the chain rule to get

$$y''(t) = \frac{\partial f}{\partial t} \frac{\partial}{\partial t} + \frac{\partial f}{\partial y} \frac{dy}{dt} = f_t + f_y f$$

Thus we can write our Taylor series expansion as

$$y(t^{n+1}) = y(t^n) + f \Delta t + (f_t + f_y f) \frac{\Delta t^2}{2} + \mathcal{O}(\Delta t^3)$$

where we have left off the fact that $f$ and $f_t + f_y f$ are explicitly evaluated at $t^n, y(t^n)$ for brevity.

To make our truncation error second order we have to have

$$y(t^{n+1}) - Y^{n+1} = \mathcal{O}(\Delta t^3)$$

where $Y^{n+1}$ is computed using the exact solution $y(t^n)$. Plugging $y(t^n)$ into our RK for $Y^n$ we have

$$Y^{n+1} = y(t^n) + ak_1 + bk_2$$
where now \( k_1 = \Delta t f(t^n, y(t^n)) \) and
\[
k_2 = \Delta t f \left( t^n + \alpha \Delta t, y(t^n) + \beta k_1 \right)
\]
Now \( k_1 \) is in a form we can subtract from our Taylor series but \( k_2 \) is not. So we expand \( f \left( t^n + \alpha \Delta t, y(t^n) + \beta k_1 \right) \) in a Taylor series in each component. We can either do this twice, the first time holding the second component first and then holding the first component fixed or we can use a Taylor series expansion for two independent variables. We get
\[
f(t^n + \alpha \Delta t, z) = f(t^n, z) + \alpha \Delta t f_t(t^n, z) + \mathcal{O}(\Delta t^2)
\]
Here we have set \( z = y(t^n) + \beta k_1 \) for shorthand notation. Note that this term \( f(t^n + \alpha \Delta t, z) \) is multiplied by \( \Delta t \) in the definition of \( k_2 \) so we only need to keep terms through \( \Delta t^2 \). Now we expand each of these terms in the second argument \( z \). We have
\[
f(t^n, y(t^n) + \beta k_1) = f(t^n, y(t^n)) + \beta k_1 f_y(t^n, y(t^n)) + \mathcal{O}(\Delta t^2)
\]
and
\[
\alpha \Delta t f_t(t^n, y(t^n) + \beta k_1) = \alpha \Delta t f_t(t^n, y(t^n)) + \mathcal{O}(\Delta t^2)
\]
Again we only kept terms through $O(\Delta t^2)$ since the whole expansion is multiplied by $\Delta t$.

- Combining these we get the following expression for $k_2$

$$k_2 = \Delta t \left[ f(t^n, y(t^n)) + \beta k_1 f_y(t^n, y(t^n)) + \alpha \Delta t f_t(t^n, y(t^n)) \right] + O(\Delta t^3)$$

- We are now ready to subtract our expansions for $Y^{n+1}$ and $y(t^{n+1})$. Recapping, we have the following where $f$ and its derivatives are all evaluated at $(t^n, y(t^n))$ (I have left this off for clarity of exposition)

$$Y^{n+1} = y(t^n) + a k_1 + b \Delta t \left[ f + \beta k_1 f_y + \alpha \Delta t f_t \right] + O(\Delta t^3)$$

$$= y(t^n) + a \Delta t f + b \Delta t \left[ f + \beta \Delta t f f_y + \alpha \Delta t f_t \right] + O(\Delta t^3)$$

and

$$y(t^{n+1}) = y(t^n) + f \Delta t + (f_t + f y f) \frac{\Delta t^2}{2} + O(\Delta t^3)$$

Subtracting yields

$$y(t^{n+1}) - Y^{n+1} = f \Delta t (1-a-b) + \Delta t^2 f_t \left( \frac{1}{2} - b \alpha \right) + f f_y \Delta t^2 \left( \frac{1}{2} - b \beta \right) + O(\Delta t^3)$$
So to make \( y(t^{n+1}) - Y^{n+1} = O(\Delta t^3) \) we need all the terms involving lower powers of \( \Delta t \) to disappear, i.e., we need

\[
a + b = 1, \quad 2\alpha b = 1 \quad 2\beta b = 1
\]

- These equations are under-determined and have an infinite number of solutions.
- For the midpoint rule we have \( a = 0, b = 1, \alpha = \frac{1}{2}, \beta = \frac{1}{2} \) which satisfy the equations. Thus the midpoint rule is a second order method.
- However, the usual choice is \( a = b = \frac{1}{2} \) and \( \alpha = \beta = 1 \).
- We take our second order scheme as follows.
Classical Second Order Runge Kutta Method

\[ k_1 = \Delta t f(t^n, Y^n) \]

\[ k_2 = \Delta t f(t^{n+1}, Y^n + k_1) \]

\[ Y^{n+1} = Y^n + \frac{1}{2}(k_1 + k_2) \]
Higher Order Runge Kutta Methods

- In the same way we can derive higher order schemes.
- For a third order method we have

**General Third Order RK:**

\[ k_1 = h(f(t^n, Y^n)) \]
\[ k_2 = hf(t^n + c_2 \Delta t, Y^n + a_{21}k_1) \]
\[ k_3 = hf(t^n + c_3 \Delta t, Y^n + a_{31}k_1 + a_{32}k_2) \]
\[ Y^{n+1} = Y^n + w_1k_1 + w_2k_2 + w_3k_3 \]
where we have adopted more general notation.

- Once again when we try to determine the coefficients which provide the highest degree of accuracy we get more than one solution for our parameters.
- Here are some popular third order RK methods.

**Classical Third Order RK:**

\[
\begin{align*}
c_2 &= \frac{1}{2} \\
c_3 &= 1 \\
a_{21} &= \frac{1}{2} \\
a_{31} &= -1 \\
a_{32} &= 2 \\
w_1 &= \frac{1}{6} \\
w_2 &= \frac{4}{6} \\
w_3 &= \frac{1}{6}
\end{align*}
\]
Heun’s Third Order RK:

\[ c_2 = \frac{1}{3}, \quad c_3 = \frac{2}{3} \]

\[ a_{21} = \frac{1}{3}, \quad a_{31} = 0, \quad a_{32} = \frac{2}{3} \]

\[ w_1 = \frac{1}{4}, \quad w_2 = 0, \quad w_3 = \frac{3}{4} \]
General Fourth Order RK:

\[ k_1 = h(f(t^n, Y^n)) \]

\[ k_2 = hf(t^n + c_2\Delta t, Y^n + a_{21}k_1) \]

\[ k_3 = hf(t^n + c_3\Delta t, Y^n + a_{31}k_1 + a_{32}k_2) \]

\[ k_4 = hf(t^n + c_4\Delta t, Y^n + a_{41}k_1 + a_{42}k_2 + a_{43}k_3) \]

\[ Y^{n+1} = Y^n + w_1k_1 + w_2k_2 + w_3k_3 + w_4k_4 \]

- One of the most popular fourth order RK methods is
Classical Fourth Order RK:

\[ c_2 = \frac{1}{2}, \quad c_3 = \frac{1}{2}, \quad c_4 = 1 \]

\[ a_{21} = \frac{1}{2}, \quad a_{31} = 0, \quad a_{32} = \frac{1}{2} \]

\[ a_{41} = 0, \quad a_{42} = 0, \quad a_{43} = 1 \]

\[ w_1 = \frac{1}{6}, \quad w_2 = \frac{1}{3}, \quad w_3 = \frac{1}{3}, \quad w_4 = \frac{1}{6} \]
Implementing RK Methods

- RK methods should be easy to implement. We simply implement the formulas in each case.
- We will write a module containing all of our methods so they will be compartmentalized.
- In writing the functions, we could use the same strategy that we did for nonlinear equations; that is, we could write a function for each method so that it will simply advance the solution one time step $\Delta t$. Then we could have a main program which sets the method to use, the initial condition, the right hand side, the final time, etc. and loops over the number of time steps.
- Another approach is to take the strategy that each function for a particular method provides the solution at some final time that is input.
- This will require a little different structure of our program but is easily doable and will provide a different viewpoint.
In order to compute the approximation at the final time, we need to input $y_0$, $n$ (or $\Delta t$) and the final time.

Our function will also need access to $f(t, y)$, i.e., the right hand side of our IVP.

Where should we put this function? We can no longer put it in the main program because how will the function in the module have access to it?

We don’t want to put it in the module containing our routines because we don’t want to modify it.

One option would be to put it in a separate file and we could just use an `include` statement to add it to the end of our module file. Another option is to make a separate module and add a `use` statement.
Function for Second Order RK

- We have a calling statement such as
  ```
  function rk_second ( y_init, n_steps, t_init, t_final ) &
  result(y_final)
  ```
  where all of the arguments should be self-explanatory. I have added the input of the initial time in case we have a nonzero start time.

- **Warning** There is one thing which can affect your error which you need to be careful of. Since a number may not be represented exactly on the computer, when we take the correct number of steps to get to the final time, we may actually be slightly off. For example, if we were only using 2 digit arithmetic and we took 3 steps of length 0.33, then we reach 0.99 instead of 1.0 So when we compute the exact solution at the final time of 1.0 and the approximation, they are actually at slightly different times.

- What I do to make sure I am actually at the exact final time is to add the statements
\[
t = t + dt \\
if ( \text{abs}( t - t_{\text{final}} ) \leq tol ) \text{ then} \\
\quad dt = t_{\text{final}} - (t - dt) \\
\quad t = t_{\text{final}} \\
end \text{ if}
\]

- After the declarations, the code could be written as the following. Note that I have written the output to file for plotting, etc.

\[
dt = ( t_{\text{final}} - t_{\text{init}} ) / \text{dfloat } (n_{\text{steps}}) \\
t = t_{\text{init}} - dt \\
yold = y_{\text{init}} \\
tol = 1.\text{e-5} \\
open ( \text{unit} = 5 , \text{file} = '\text{out}_\text{rk2.txt}' ) \\
write(5, *) t_{\text{init}}, y_{\text{init}} \\
do k = 1, n_{\text{steps}}\]
\[ t = t + dt \]

if \( \text{abs}( t - t_{\text{final}}) \leq \text{tol} \) then

\[ dt = t_{\text{final}} - (t - dt) \]

\[ t = t_{\text{final}} \]

end if

\[ \text{kone} = dt * \text{rhs}(t, y_{\text{old}}) \]

\[ \text{ktwo} = dt * \text{rhs}(t + dt, y_{\text{old}} + \text{kone}) \]

\[ y_{\text{new}} = y_{\text{old}} + (\text{kone} + \text{ktwo}) / \text{two} \]

write(5, *) t, y_{\text{new}}

\[ y_{\text{old}} = y_{\text{new}} \]

end do

\[ y_{\text{final}} = y_{\text{new}} \]
close ( unit = 5 )

At the end of the module I have added an \texttt{include} statement to include the file \texttt{rhs.f90}
Rate of convergence

- Now we want to compute the rate of convergence to convince ourselves that it is second order.
- Recall that we expect \(|y(t^n) - Y^n| = \mathcal{O}(\Delta t^2)|. We would like to verify this numerically so we can have some confidence that the method is working correctly.
- We do this in a similar manner to the way we computed the rate of convergence for Newton’s Method. If we let \(E_i = |y(t^n) - Y^n|\) computed with \(\Delta t_i\) then we have

\[
E_1 = C\Delta t_1^r \quad E_2 = C\Delta t_2^r
\]

where \(r\) is the numerical rate of convergence which we hope will be two.
- Solving these equations for \(r\) we get

\[
C = \frac{E_1}{\Delta t_1^r} = \frac{E_2}{\Delta t_2^r} \quad \Rightarrow \quad \frac{E_1}{E_2} = \left[\frac{\Delta t_1}{\Delta t_2}\right]^r \quad \Rightarrow \quad r = \ln\left[\frac{E_1}{E_2}\right] / \ln\left[\frac{\Delta t_1}{\Delta t_2}\right]
\]
and thus if we cut $\Delta t$ in half each time we get

$$r = \frac{\ln \left[ \frac{E_1}{E_2} \right]}{\ln 2}$$
Runge Kutta Methods for Solving Systems of IVPs

- The concept of generalizing RK methods for solving systems of IVPs is similar to what we did for Euler’s method.

- Consider the system of two IVPs

  \[ u'(t) = f(t, u, v) \quad v'(t) = g(t, u, v) \]

  where

  \[ u(t^0) = u_0, \quad v(t^0) = v_0. \]

- Since each equation has a different right hand side, we have to compute the \( k_i \) for each equation.

- Suppose that we are using the midpoint rule which is a second order scheme. Recall that here we have for a single IVP

  \[ k_1 = \Delta t f(t^n, Y^n) \quad k_2 = \Delta t f(t^n + \frac{1}{2}, Y^n + \frac{1}{2} k_1), \quad Y^{n+1} = Y^n + k_2 \]
To distinguish the two sets of coefficients let's call the terms for $u$, $k_i$ and those for $v$, $m_i$.

Let's look at how we compute $U^1, V^1$. We first compute $k_1$ and $m_1$

$$k_1 = \Delta t f(t^0, U^0, V^0), \quad m_1 = \Delta t g(t^0, U^0, V^0)$$

Now we compute $k_2, m_2$

$$k_2 = \Delta t f(t^0 + \frac{\Delta t}{2}, U^0 + \frac{1}{2} k_1, V^0 + \frac{1}{2} m_1), \quad m_2 = \Delta t g(t^0 + \frac{\Delta t}{2}, U^0 + \frac{1}{2} k_1, V^0 + \frac{1}{2} m_1),$$

Note that we can not compute all the $k_i$'s and then all the $m_i$'s. Why?

Finally we compute the solution

$$U^1 = U^0 + k_2 \quad V^1 = V^0 + m_2$$

Example. Consider the system

$$u' = v \quad v' = -\frac{2}{t}v$$

$$u(1) = 10 \quad v(1) = 1$$

Let's do one step of length 0.2 using the midpoint RK scheme.
For $U^1$ we have

$$k_1 = .2 f(1, U^0, V^0) = .2 * V^0 = .2 \quad \text{since } f = v$$

$$m_1 = .2 g(1, U^0, V^0) = .2 * \left( \frac{-2}{1} \right) * V^0 = -0.4 \quad \text{since } g = -\frac{2}{t}v$$

$$k_2 = .2 f(1.1, U^0 + \frac{k_1}{2}, V^0 + \frac{m_1}{2}) = .2(1 + (-.4)/2) = 0.16$$

$$m_2 = .2 f(1.1, U^0 + \frac{k_1}{2}, V^0 + \frac{m_1}{2}) = .2(\frac{-2}{1.1}) .8 = -0.2909$$

$$U^1 = U^0 + k_2 = 10 + 0.16 = 10.16$$

$$V^1 = V^0 + m_2 = 1 - 0.2909 = 0.709091$$
Classwork

• Add routines to your module `ivp_solvers` to incorporate the classical second order RK method. Assume that your function inputs the initial conditions, the number of steps, the initial and final times and returns the final value of the solution. Compile this; use a separate file for your $f(t, y)$ and use an `include` statement.

• Test the code on our example $y' = y + t$, $y(0) = 2$ whose exact solution is $y(t) = 3e^t - t - 1$. Run the code for $n = 10, 20, 40$ points and compute the error.