CHAPTER

17
Initial Value Problems

OUTLINE

17.1 Forward Euler 302
17.2 Backward Euler 305
17.3 Crank–Nicolson (Trapezoid Rule) 308
17.3.1 Comparison of the Methods 310
17.4 Stability 311
17.5 Fourth-Order Runge–Kutta Method 313
17.5.1 Stability for RK4 314
17.6 Systems of Differential Equations 315
17.6.1 Stability for Systems 320
17.6.2 Crank–Nicolson for Systems 321
17.6.3 RK4 for Systems 323
17.7 Point-Reactor Kinetics Equations 324
17.7.1 Rod-Drop 325
17.7.2 Linear Reactivity Ramp 327
Coda 329
Further Reading 329
Problems 329
Short Exercises 329
Programming Projects 330
1. Point Reactor Kinetics 330
2. Iodine Ingestion 331

Maude: *Lord. You can imagine where it goes from here.*

Dude: *He fixes the cable?*

***–from the film* The Big Lebowski**

CHAPTER POINTS

- Initial value problems require the application of an integration rule to update dependent variables.
- Explicit methods use information from previous and current times to march forward in time. These methods have a limited step size.
- Implicit methods define the update in terms of the state at a future time. These methods can take large steps, but may be limited in accuracy. Implicit methods also require the solution of linear or nonlinear equations for each update.
- To solve systems of equations, we can apply generalizations of the techniques developed for single ODEs.

DOI: 10.1016/B978-0-12-812253-2.00019-4

In this chapter we will use numerical integration techniques to solve ordinary differential equations (ODEs) where the value of the solution is specified at a single value of the independent variable. Because these problems often correspond to time-dependent problems with an initial condition, these are called initial value problems.

Consider the generic, initial value problem with a first-order derivative given by

$$y'(t) = f(y,t), \qquad y(0) = y_0,$$

where $f(y,t)$ is a function that in general depends on y and t. Typically, we call t the time variable and y our solution. For a problem of this sort we can simply integrate both sides of the equation from $t = 0$ to $t = \Delta t$, where Δt is called the time step. Doing this we get

$$y(\Delta t) - y(0) = \int_0^{\Delta t} f(y,t)\,dt. \tag{17.1}$$

17.1 FORWARD EULER

To proceed we will treat the integral in the previous equation using a numerical integration rule. One rule that is so basic we did not talk about in the chapters on numerical integration is the left-hand rectangle rule. Here we estimate the integral as

$$\int_0^{\Delta t} f(y,t)\,dt \approx \Delta t f(y(0),0). \tag{17.2}$$

Using this relation in Eq. (17.1) gives us

$$y(\Delta t) = y(0) + \Delta t f(y(0),0). \tag{17.3}$$

This will give an approximate value of the solution Δt after the initial condition. If we wanted to continue out to later times, we could apply the rule again repeatedly. To do this we define the value of y after n timesteps, each of width Δt as

$$y^n = y(t^n) = y(n\Delta t), \qquad \text{for } n = 0, \dots, N. \tag{17.4}$$

Using this we can write the solution using the left-hand rectangle rule for integration as

$$y^{n+1} = y^n + \Delta t f(y^n, t^n).$$

This method is called the explicit Euler method or the forward Euler method after the Swiss mathematician whose name is commonly pronounced *oi-ler*, much like a hockey team from Edmonton. The method is said to be explicit, not because sometimes it will make you want to shout profanity, rather that the update is explicitly defined by the value of the solution at time t^n. Below we define a Python function that for a given right-hand side, initial condition, and time step and number of time steps, N, performs the forward Euler method. This function will take the name of the function on the right-hand side as an input.

BOX 17.1 NUMERICAL PRINCIPLE

An explicit numerical method formulates the update to time step $n+1$ in terms of quantities only at time step n or before. The classical example of an explicit method is the forward Euler method which writes the solution to $y'(t) = f(y,t)$ as

$$y^{n+1} = y^n + \Delta t f(y^n, t^n) \qquad \text{Forward Euler.}$$

```
In [1]: def forward_euler(f,y0,Delta_t,numsteps):
            """Perform numsteps of the forward Euler method starting at y0
            of the ODE y'(t) = f(y,t)
            Args:
                f: function to integrate takes arguments y,t
                y0: initial condition
                Delta_t: time step size
                numsteps: number of time steps

            Returns:
                a numpy array of the times and a numpy
                array of the solution at those times
            """
            numsteps = int(numsteps)
            y = np.zeros(numsteps+1)
            t = np.arange(numsteps+1)*Delta_t
            y[0] = y0
            for n in range(1,numsteps+1):
                y[n] = y[n-1] + Delta_t * f(y[n-1], t[n-1])
            return t, y
```

We will test this on a simple problem:

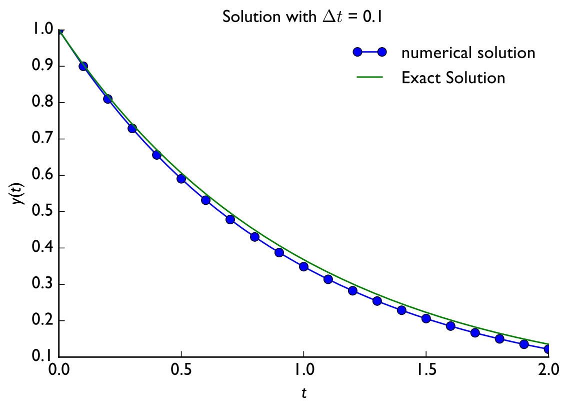
$$y'(t) = -y(t), \qquad y_0 = 1.$$

The solution to this problem is

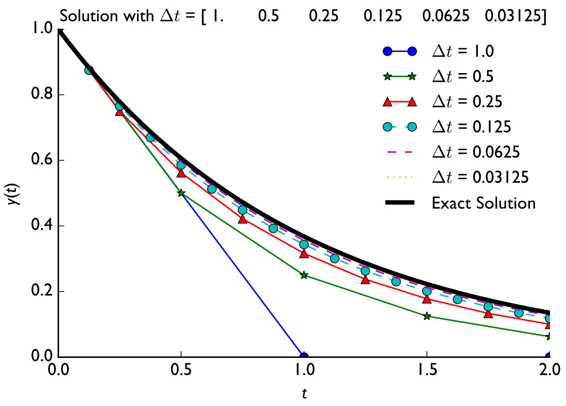
$$y(t) = e^{-t}.$$

The following code compares the exact solution to the forward Euler solution with a time step of 0.1.

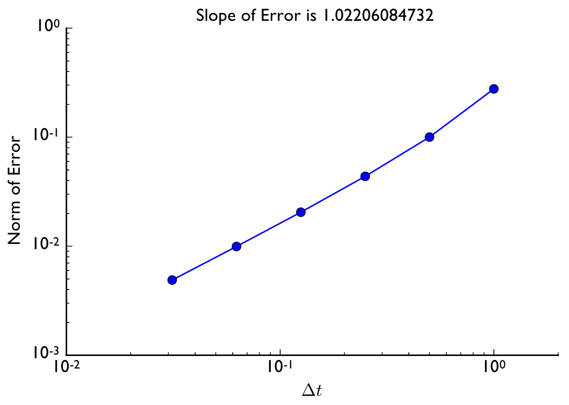
```
In [2]: RHS = lambda y,t: -y
        Delta_t = 0.1
        t_final = 2
        t,y = forward_euler(RHS,1,Delta_t,t_final/Delta_t)
        plt.plot(t,y,'o-',label="numerical solution")
        t_fine = np.linspace(0,t_final,100)
        plt.plot(t_fine,np.exp(-t_fine),label="Exact Solution")
```



That looks pretty close. The numerical solution appears to be slightly below the exact solution, but the difference appears to be small. We could re-do this with different size time steps and compare the solutions as a function of Δt.



Also, we can compute the error as a function of Δt. On a log-log scale this will tell us about the convergence of this method in Δt.

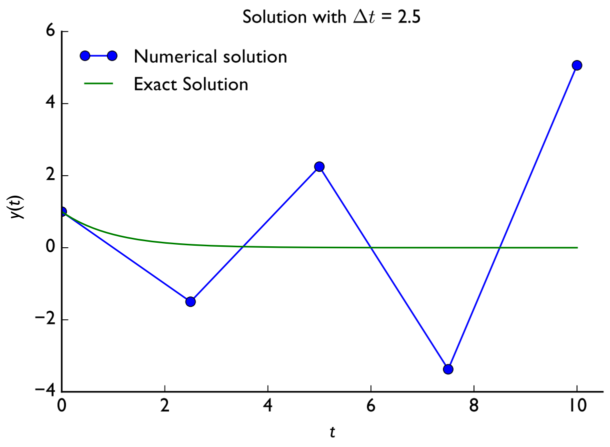


Notice that the error indicates that this is a first-order method in Δt: when we decrease Δt by a factor of 2, the error decreases by a factor of 2. In this case we measured the error with a slightly different error norm:

$$\text{Error} = \frac{1}{\sqrt{N}}\sqrt{\sum_{n=1}^{N}\left(y^n_{\text{approx}} - y^n_{\text{exact}}\right)^2},$$

where N is the number of steps the ODE is solved over.

One thing that can happen with the forward Euler method is that if the time step is too large, it can become unstable. What this means is the solution diverges to be plus or minus infinity (sometimes it goes to both). In our case, forward Euler is unstable if $\Delta t > 2$, as we will prove later and demonstrate numerically here.



The solution grows over time, even though the true solution decays to 0. This happens because the magnitude of the solution grows, which makes the right-hand side for the next update larger. This makes the solution grow in magnitude each step.

Stability is an important consideration, and we will talk more about it later.

17.2 BACKWARD EULER

We could use a different method to integrate our original ODE rather than the left-hand rectangle rule. An obvious alternative is the right-hand rectangle rule:

$$y^{n+1} = y^n + \Delta t f(y^{n+1}, t^{n+1}).$$

This method is called he backward Euler method or the implicit Euler method.

It is implicit because the update is implicitly defined by evaluating f with the value of y we are trying to solve for. That means the update needs to solve the equation

$$y^{n+1} - y^n - \Delta t f(y^{n+1}, t^{n+1}) = 0, \tag{17.5}$$

using a nonlinear solver (unless f is linear in y). Therefore, this method is a bit harder to implement in code.

BOX 17.2 NUMERICAL PRINCIPLE

An implicit numerical method formulates the update to time step $n+1$ in terms of quantities at time step $n+1$ and possibly previous time steps. The classical example of an implicit method is the backward Euler method which write the solution to $y'(t) = f(y,t)$ as

$$y^{n+1} = y^n + \Delta t f(y^{n+1}, t^{n+1})$$

Backward Euler.

A well-known second-order implicit method is the Crank-Nicolson method. It uses values at time step $n+1$ and n:

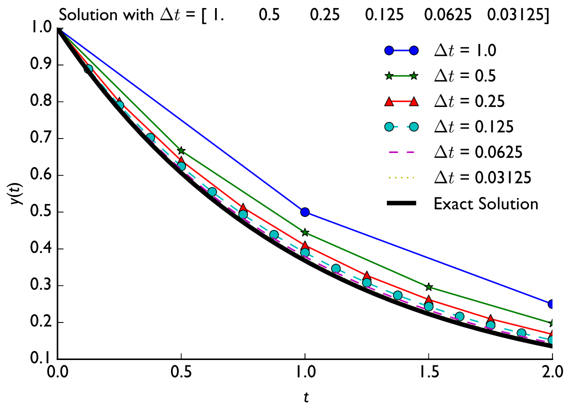
$$y^{n+1} = y^n + \frac{1}{2}\Delta t \left[f(y^n, t^n) + f(y^{n+1}, t^{n+1}) \right]$$

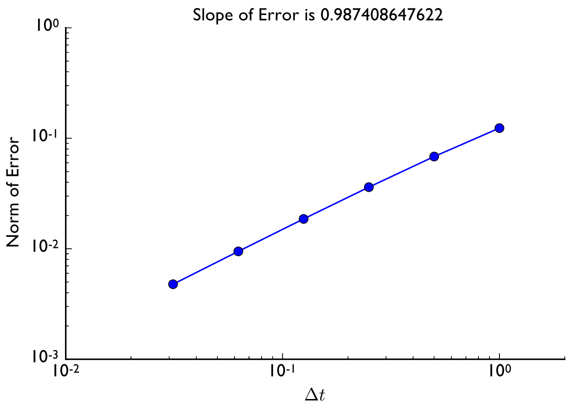
Crank-Nicolson.

The function below uses the inexact Newton method we defined before to solve Eq. (17.5).

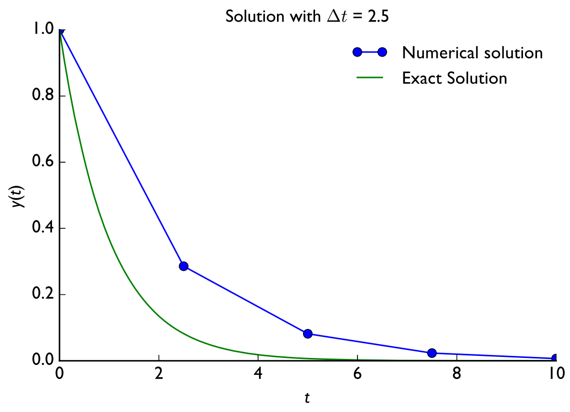
```
In [3]: def backward_euler(f,y0,Delta_t,numsteps):
            """Perform numsteps of the backward Euler method starting at y0
            of the ODE y'(t) = f(y,t)
            Args:
                f: function to integrate takes arguments y,t
                y0: initial condition
                Delta_t: time step size
                numsteps: number of time steps

            Returns:
                a numpy array of the times and a numpy
                array of the solution at those times
            """
            numsteps = int(numsteps)
            y = np.zeros(numsteps+1)
            t = np.arange(numsteps+1)*Delta_t
            y[0] = y0
            for n in range(1,numsteps+1):
                solve_func = lambda u: u-y[n-1] - Delta_t*f(u,t[n])
                y[n] = inexact_newton(solve_func,y[n-1])
            return t, y
```

Performing the test we did for forward Euler gives the following results.





There are several differences between the results from backward Euler and forward Euler. The backward Euler solutions approach the exact solution from above, and the convergence of the error is at the same rate as forward Euler. It seems like we did not get much for the extra effort of solving a nonlinear equation at each step. What we do get is unconditional stability. We can see this by using a large time step.



The solution, though not very accurate, still behaves reasonably well. The exact solution decays to 0 as $t \to \infty$, and the backward Euler solution behaves the same way. This behavior can be very useful on more complicated problems than this simple one.

17.3 CRANK–NICOLSON (TRAPEZOID RULE)

We could use the trapezoid rule to integrate the ODE over the time step. Doing this gives

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

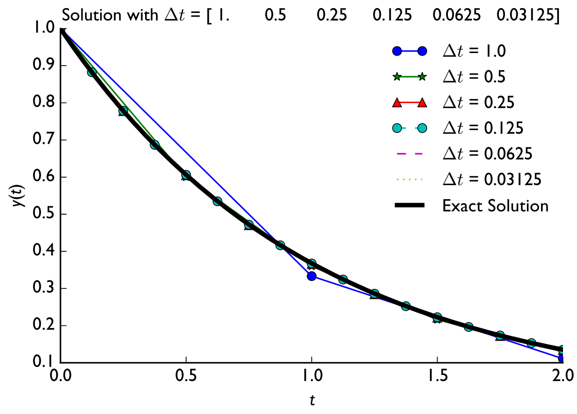
This method, often called Crank–Nicolson, is also an implicit method because y^{n+1} is on the right-hand side of the equation. For this method the equation we have to solve at each time step is

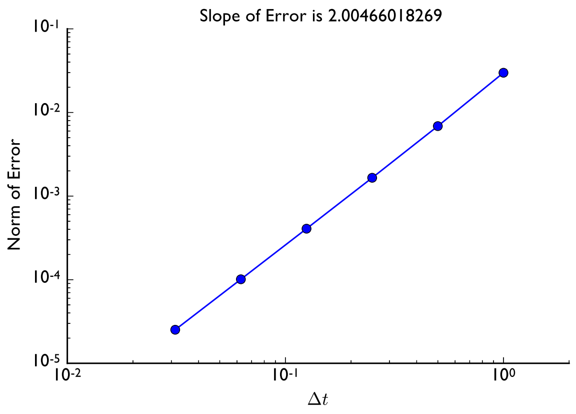
$$y^{n+1} - y^n - \frac{\Delta t}{2}\left(f(y^n,t^n) + f(y^{n+1},t^{n+1})\right) = 0.$$

Implementing this method is no more difficult than backward Euler. The only change is the function given to the inexact Newton method.

```
In [4]: def crank_nicolson(f,y0,Delta_t,numsteps):
            """Perform numsteps of the Crank--Nicolson method starting at y0
            of the ODE y'(t) = f(y,t)
            Args:
                f: function to integrate takes arguments y,t
                y0: initial condition
                Delta_t: time step size
                numsteps: number of time steps

            Returns:
                a numpy array of the times and a numpy
                array of the solution at those times
            """
            numsteps = int(numsteps)
            y = np.zeros(numsteps+1)
            t = np.arange(numsteps+1)*Delta_t
            y[0] = y0
            for n in range(1,numsteps+1):
                solve_func = lambda u:u-y[n-1]-0.5*Delta_t*(f(u,t[n])
                                                        + f(y[n-1],t[n-1]))
                y[n] = inexact_newton(solve_func,y[n-1])
            return t, y
```

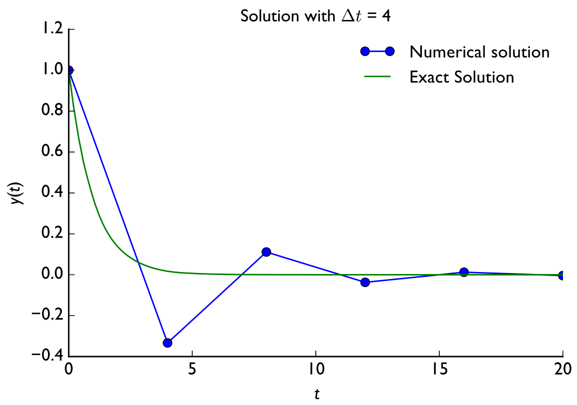
On our test where the solution is e^{-t}, we can see the benefit of having a second-order method:





We do get the expected second-order convergence of the error as evidenced by the error plot. In the comparison of the solutions, except for the large value of Δt, it is hard to distinguish the approximate and the exact solutions.

In terms of stability, Crank–Nicolson is a mixed bag. The method will not diverge as $\Delta t \to \infty$, but it can oscillate around the correct solution:



Notice that the oscillation makes the numerical solution negative. This is the case even though the exact solution, e^{-t}, cannot be negative.

17.3.1 Comparison of the Methods

We will compare the methods on a problem that has a driving term as well as decay:

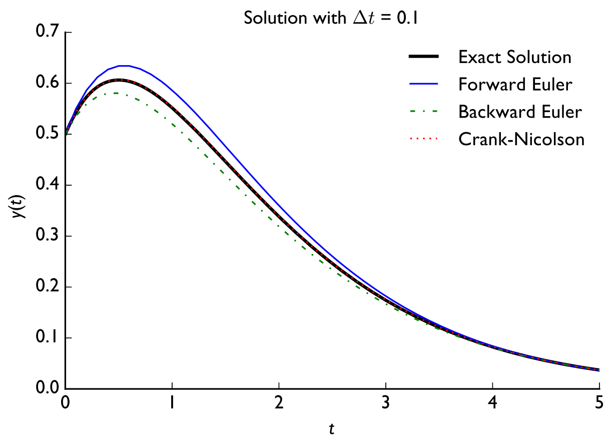
$$y'(t) = \left(\frac{1}{t+1/2} - 1\right) y(t), \qquad y(0) = \frac{1}{2}. \tag{17.6}$$

The purpose of this exercise is to show that the issues of accuracy are even more important when the solution is not a simple exponential.

The solution to this problem is

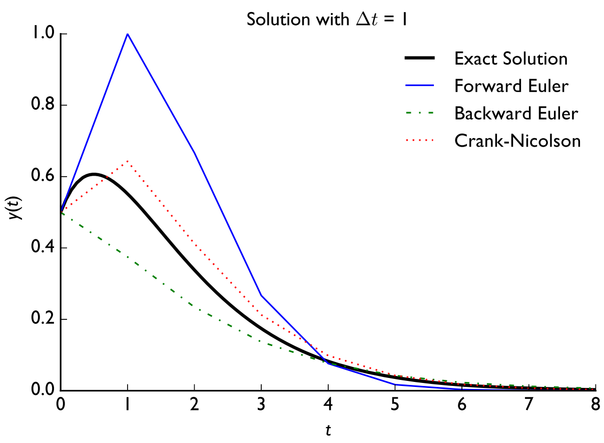
$$y(t) = \left(t + \frac{1}{2}\right) e^{-t}.$$

We will start with a small value of Δt:



The two Euler methods are equally inaccurate, they just differ in how they are wrong (above or below). Crank–Nicolson does a good job of following the solution, and is indiscernible from the exact solution.

With a bigger time step we can see the warts of the methods that we have presented:



In these results we see that for a large time step, forward Euler completely overshoots the initial growth, backward Euler just starts decaying, and Crank–Nicolson starts off too high before beginning to decay. It appears that even Crank-Nicolson is not accurate enough for this problem with this large step.

17.4 STABILITY

There is a formal definition of stability for a numerical method for integrating ODEs. To get to this definition, consider the ODE

$$y'(t) = -\alpha y(t). \tag{17.7}$$

For any single-step method we can write

$$y^{n+1} = g y^n.$$

A solution method is said to be stable if $|g| \leq 1$. Furthermore, a solution is said to be non-oscillatory if $0 \leq g \leq 1$. The quantity g is often called the growth rate.

We first look at forward Euler on this ODE:

$$y^{n+1} = y^n - \alpha \Delta t y^n = (1 - \alpha \Delta t) y^n,$$

this implies that the growth rate for forward Euler is

$$g_{\text{FE}} = 1 - \alpha \Delta t.$$

To make sure that $|g| \leq 1$ we need to have $\alpha \Delta t \leq 2$. To be non-oscillatory we need $\alpha \Delta t \leq 1$. This is why when we solved

$$y'(t) = -y(t)$$

with $\Delta t = 2.5$, the solution grew in an unstable manner. Because there is a restriction on the time step for stability, we call the forward Euler method conditionally stable.

BOX 17.3 NUMERICAL PRINCIPLE

There are several types of stability that are important. These are usually discussed in how the update for the ODE, $y'(t) = -\alpha y(t)$, is given. For any single-step method we can write the update for this problem as

$$y^{n+1} = gy^n.$$

Here g is the growth rate.

- A method is **conditionally stable** if the growth rate has a magnitude greater than 1 for certain positive values of the time step size.
- A method is **unconditionally stable** if the growth rate has a magnitude smaller than or equal to 1 for all positive values of the time step size.
- A method is **non-oscillatory** if the growth rate is between 0 and 1.

The value of the growth rate for backward Euler can be easily derived. We start with

$$y^{n+1} = y^n - \alpha\Delta t y^{n+1},$$

which when rearranged is

$$y^{n+1} = \frac{y^n}{1+\alpha\Delta t}.$$

This makes

$$g_{\mathrm{BE}} = \frac{1}{1+\alpha\Delta t}.$$

For any $\Delta t > 0$, g will be between 0 and 1. Therefore, backward Euler is unconditionally stable and unconditionally non-oscillatory.

The Crank–Nicolson method has

$$g_{\mathrm{CN}} = \frac{2-\alpha\Delta t}{2+\alpha\Delta t}.$$

This method will be unconditionally stable because

$$\lim_{\Delta t\to\infty} g_{\mathrm{CN}} = -1.$$

It is conditionally non-oscillatory because $g_{\mathrm{CN}} < 0$ for $\alpha\Delta t > 2$. In the original example, we had $\alpha\Delta t = 4$, and we saw noticeable oscillations.

In the contrast between the implicit methods, Crank–Nicolson and backward Euler, we see a common theme in numerical methods: to get better accuracy one often has to sacrifice robustness. In this case Crank–Nicolson allows oscillations in the solution, but its error decreases at second-order in the time step size. Backward Euler is non-oscillatory, but the error converges more slowly than Crank–Nicolson.

17.5 FOURTH-ORDER RUNGE–KUTTA METHOD

There is one more method that we need to consider at this point. It is an explicit method called a Runge–Kutta method. The particular one we will learn is fourth-order accurate in Δt. That means that if we decrease Δt by a factor of 2, the error will decrease by a factor of $2^4 = 16$. The method can be written as

$$y^{n+1} = y^n + \frac{1}{6}(\Delta y_1 + 2\Delta y_2 + 2\Delta y_3 + \Delta y_4), \tag{17.8}$$

where

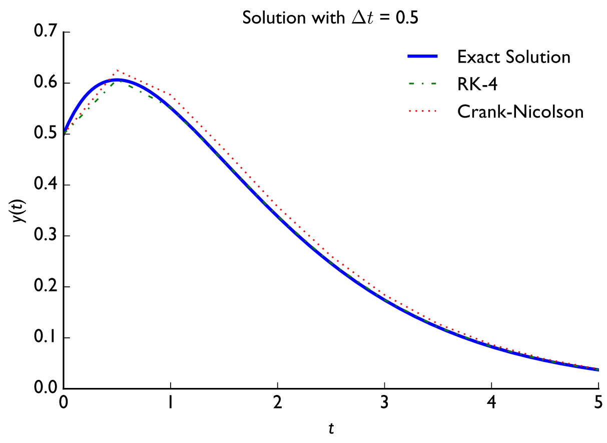
$$\begin{aligned}
\Delta y_1 &= \Delta t f(y^n, t^n), \\
\Delta y_2 &= \Delta t f\left(y^n + \frac{\Delta y_1}{2}, t^n + \frac{\Delta t}{2}\right), \\
\Delta y_3 &= \Delta t f\left(y^n + \frac{\Delta y_2}{2}, t^n + \frac{\Delta t}{2}\right), \\
\Delta y_4 &= \Delta t f\left(y^n + \Delta y_3, t^n + \Delta t\right).
\end{aligned}$$

To get fourth-order accuracy, this method takes a different approach to integrating the right-hand side of the ODE. Basically, it makes several projections forward and combines them in such a way that the errors cancel to make the method fourth-order.

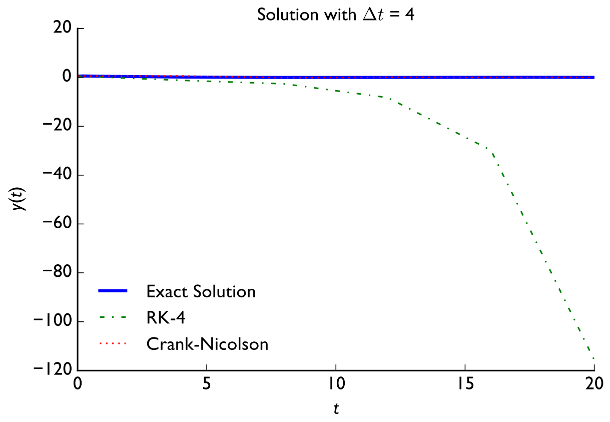
Implementing this method is not difficult either.

```
In [5]: def RK4(f,y0,Delta_t,numsteps):
            """Perform numsteps of the 4th-order Runge-Kutta
            method starting at y0 of the ODE y'(t) = f(y,t)
            Args:
                f: function to integrate takes arguments y,t
                y0: initial condition
                Delta_t: time step size
                numsteps: number of time steps

            Returns:
                a numpy array of the times and a numpy
                array of the solution at those times
            """
            numsteps = int(numsteps)
            y = np.zeros(numsteps+1)
            t = np.arange(numsteps+1)*Delta_t
            y[0] = y0
            for n in range(1,numsteps+1):
                dy1 = Delta_t * f(y[n-1], t[n-1])
                dy2 = Delta_t * f(y[n-1] + 0.5*dy1, t[n-1] + 0.5*Delta_t)
                dy3 = Delta_t * f(y[n-1] + 0.5*dy2, t[n-1] + 0.5*Delta_t)
                dy4 = Delta_t * f(y[n-1] + dy3, t[n-1] + Delta_t)
                y[n] = y[n-1] + 1.0/6.0*(dy1 + 2.0*dy2 + 2.0*dy3 + dy4)
            return t, y
```

We will first try this Runge–Kutta method on the problem with growth and decay given in Eq. (17.6) with a large time step.



It seems to do better that Crank–Nicolson with a large time step, but since it is an explicit method, there are limits: with a large enough time step the solution will oscillate and can be unstable.



Therefore, we need to be careful when we use the RK-4 method that the time step is not too large.

17.5.1 Stability for RK4

For the fourth-order Runge–Kutta method presented above, we can look at the stability by examining how it performs on the problem $y'(t) = -\alpha y$. On this problem the intermediate values are

$$\Delta y_1 = -\alpha \Delta t y^n,$$

$$\begin{aligned}
\Delta y_2 &= -\alpha\Delta t\left(y^n + \frac{\Delta y_1}{2}\right)\\
&= -\alpha\Delta t\left(1-\frac{\alpha\Delta t}{2}\right)y^n,\\
\Delta y_3 &= -\alpha\Delta t\left(y^n + \frac{\Delta y_2}{2}\right)\\
&= -\alpha\Delta t\left(1-\frac{\alpha\Delta t}{2}\left(1-\frac{\alpha\Delta t}{2}\right)\right)y^n,\\
\Delta y_4 &= -\alpha\Delta t\left(y^n + \Delta y_3\right)\\
&= -\alpha\Delta t\left(1-\alpha\Delta t\left(1-\frac{\alpha\Delta t}{2}\left(1-\frac{\alpha\Delta t}{2}\right)\right)\right)y^n.
\end{aligned}$$

Using these results in the update for y^{n+1} in Eq. (17.8) gives

$$y^{n+1} = \left(\frac{\alpha^4\Delta t^4}{24} - \frac{\alpha^3\Delta t^3}{6} + \frac{\alpha^2\Delta t^2}{2} - \alpha\Delta t + 1\right)y^n.$$

This implies that the growth rate for RK4 is

$$g_{\text{RK4}} = \frac{\alpha^4\Delta t^4}{24} - \frac{\alpha^3\Delta t^3}{6} + \frac{\alpha^2\Delta t^2}{2} - \alpha\Delta t + 1. \tag{17.9}$$

This is a fourth-degree polynomial in $\alpha\Delta t$. For $\alpha\Delta t > 0$ the value of g_{RK4} is positive. Additionally, the stability limit is

$$\alpha\Delta t \le \frac{1}{3}\left(4 - 10\sqrt[3]{\frac{2}{43+9\sqrt{29}}} + 2^{2/3}\sqrt[3]{43+9\sqrt{29}}\right) \approx 2.78529356.$$

This implies that RK4 is conditionally stable, and is non-oscillatory where it is stable. The stability criterion is less restrictive for RK4 than for forward Euler, and where Crank–Nicolson is non-oscillatory, RK4 will be as well.

17.6 SYSTEMS OF DIFFERENTIAL EQUATIONS

Often we will be concerned with solving a system of ODEs rather than a single ODE. The explicit methods translate to this scenario directly. However, we will restrict ourselves to systems that can be written in the form

$$\mathbf{y}'(t) = \mathbf{A}(t)\mathbf{y} + \mathbf{c}(t), \qquad \mathbf{y}(0) = \mathbf{y}_0.$$

In this equation $\mathbf{A}(t)$ is a matrix that can change over time, and $\mathbf{c}(t)$ is a function of t only. For systems of this type our methods are written as follows:

- Forward Euler

$$\mathbf{y}^{n+1} = \mathbf{y}^n + \Delta t\mathbf{A}(t^n)\mathbf{y}^n + \Delta t\mathbf{c}(t^n).$$

- Backward Euler

$$\mathbf{y}^{n+1} = \mathbf{y}^n + \Delta t\mathbf{A}(t^{n+1})\mathbf{y}^{n+1} + \Delta t\mathbf{c}(t^{n+1}),$$

  which rearranges to

$$\left(\mathbf{I} - \Delta t\mathbf{A}(t^{n+1})\right)\mathbf{y}^{n+1} = \mathbf{y}^n + \Delta t\mathbf{c}(t^{n+1}).$$

  Therefore, for backward Euler we will have to solve a linear system of equations at each time step.
- Crank-Nicolson

$$\left(\mathbf{I} - \frac{\Delta t}{2}\mathbf{A}(t^{n+1})\right)\mathbf{y}^{n+1} = \left(\mathbf{I} + \frac{\Delta t}{2}\mathbf{A}(t^n)\right)\mathbf{y}^n + \frac{\Delta t}{2}\left(\mathbf{c}(t^{n+1}) + \mathbf{c}(t^n)\right).$$

  This will also involve a linear solve at each step.
- Fourth-order Runge–Kutta

$$\mathbf{y}^{n+1} = \mathbf{y}^n + \frac{1}{6}\left(\Delta\mathbf{y}_1 + 2\Delta\mathbf{y}_2 + 2\Delta\mathbf{y}_3 + \Delta\mathbf{y}_4\right),$$
$$\Delta y_1 = \Delta t\mathbf{A}(t^n)y^n + \mathbf{c}(t^n),$$
$$\Delta y_2 = \Delta t\mathbf{A}\left(t^n + \frac{\Delta t}{2}\right)\left(y^n + \frac{\Delta y_1}{2}\right) + \Delta t\mathbf{c}\left(t^n + \frac{\Delta t}{2}\right),$$
$$\Delta y_3 = \Delta t\mathbf{A}\left(t^n + \frac{\Delta t}{2}\right)\left(y^n + \frac{\Delta y_2}{2}\right) + \Delta t\mathbf{c}\left(t^n + \frac{\Delta t}{2}\right),$$
$$\Delta y_4 = \Delta t\mathbf{A}\left(t^n + \Delta t\right)\left(y^n + \Delta y_3\right) + \Delta t\mathbf{c}(t^n + \Delta t).$$

As noted above, the implicit methods require the solution of a linear system at each time step, while the explicit methods do not.

We first define a function for solving forward Euler on a system.

```
In [6]: def forward_euler_system(Afunc,c,y0,Delta_t,numsteps):
            """Perform numsteps of the backward euler method starting at y0
            of the ODE y'(t) = A(t) y(t) + c(t)
            Args:
                Afunc: function to compute A matrix
                c: nonlinear function of time
                Delta_t: time step size
                numsteps: number of time steps

            Returns:
                a numpy array of the times and a numpy
                array of the solution at those times
            """
            numsteps = int(numsteps)
            unknowns = y0.size
```

```
    y = np.zeros((unknowns,numsteps+1))
    t = np.arange(numsteps+1)*Delta_t
    y[0:unknowns,0] = y0
    for n in range(1,numsteps+1):
        yold = y[0:unknowns,n-1]
        A = Afunc(t[n-1])
        y[0:unknowns,n] = yold + Delta_t * (np.dot(A,yold) + c(t[n-1]))
    return t, y
```

As a test we will solve the ODE:

$$y''(t) = -y(t), \qquad y(0) = 1, \quad y'(0) = 0.$$

At first blush this does not seem compatible with our method for solving a system of equations. Moreover, we have not covered how to solve ODEs with derivatives other than first derivatives. Nevertheless, we can write this as a system using the definition

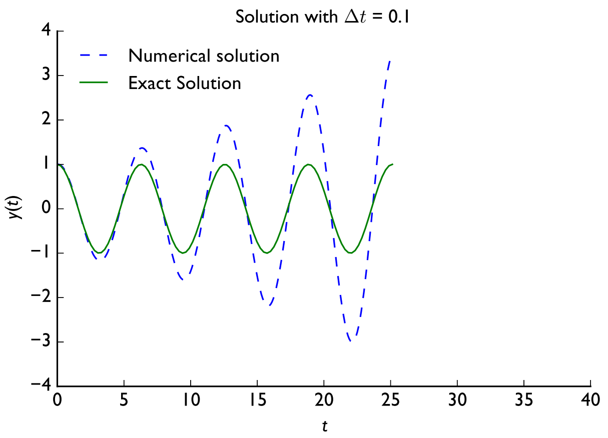
$$u(t) = y'(t),$$

to get

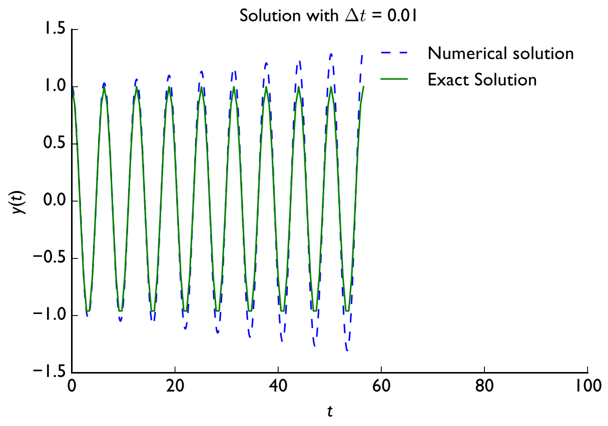
$$\frac{d}{dt}\begin{pmatrix} u \\ y \end{pmatrix} = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}\begin{pmatrix} u \\ y \end{pmatrix}.$$

We will set this up in Python and solve it with forward Euler. The solution is $y(t) = \cos(t)$.

```
In [7]: #Set up A
        Afunc = lambda t: np.array([(0,-1),(1,0)])
        #set up c
        c = lambda t: np.zeros(2)
        #set up y
        y0 = np.array([0,1])
        Delta_t = 0.1
        t_final = 8*np.pi
        t,y = forward_euler_system(Afunc,c,y0,Delta_t,t_final/Delta_t)
```

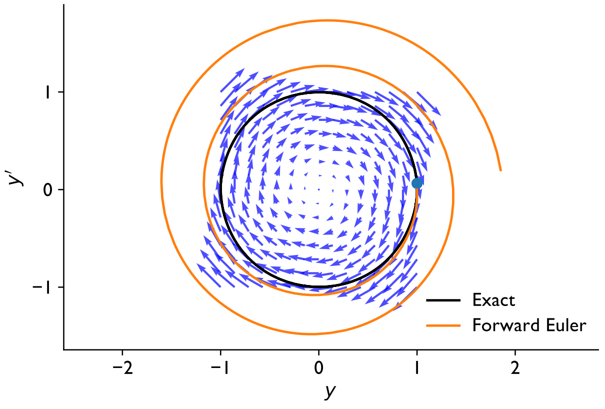


The error grows over time. What's happening here is that the numerical error builds over time and this causes the magnitude to grow over time. Using a smaller value of Δt can help, but not completely remove the problem.



We have really just delayed the inevitable: the growth of the magnitude of the solution is increasing.

To understand what is going on we will look a phase field plot for this ODE. The phase field plot for this system tells us the direction of the derivatives of y and u given a value for each. Using the phase field and a starting point, we can follow the arrows to see how the true solution behaves. In the following phase figure, the solid black line shows the behavior of the solution when $y(0) = 1$ and $u(t) = y'(0) = 0$. The solution starts at the solid circle and goes around twice in the time plotted ($t = 0$ to 4π).

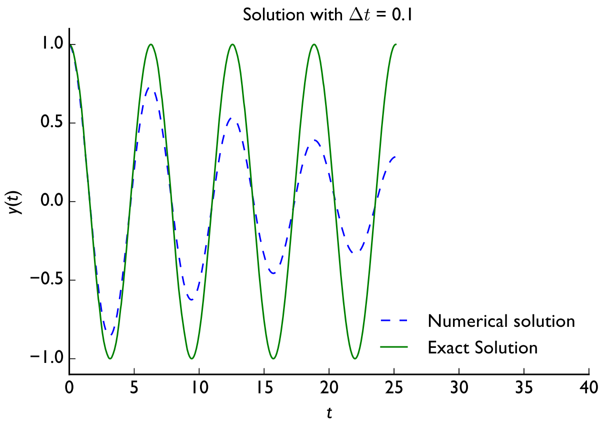


The solution looks like a circle in the phase field because of the repeating periodic nature of the solution. However, the forward Euler solution has errors that grow over time, so that each time around, the circle grows larger.

The backward Euler method is implemented next. To solve the linear system we will use a Gaussian elimination function that we defined previously.

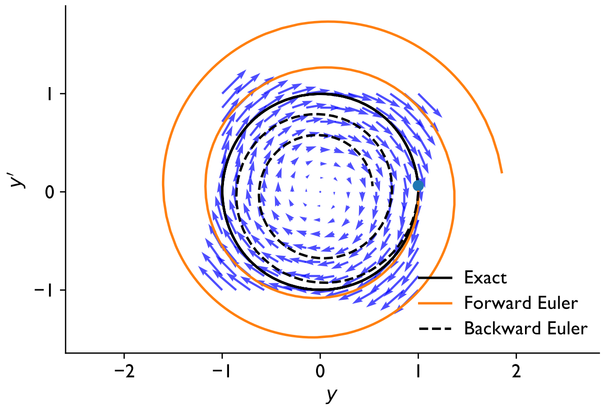
```
In [8]: def backward_euler_system(Afunc,c,y0,Delta_t,numsteps):
            """Perform numsteps of the backward euler method starting at y0
            of the ODE y'(t) = A(t) y(t) + c(t)
            Args:
                Afunc: function to compute A matrix
                c: nonlinear function of time
                y0: initial condition
                Delta_t: time step size
                numsteps: number of time steps

            Returns:
                a numpy array of the times and a numpy
                array of the solution at those times
            """
            numsteps = int(numsteps)
            unknowns = y0.size
            y = np.zeros((unknowns,numsteps+1))
            t = np.arange(numsteps+1)*Delta_t
            y[0:unknowns,0] = y0
            for n in range(1,numsteps+1):
                yold = y[0:unknowns,n-1]
                A = Afunc(t[n])
                LHS = np.identity(unknowns) - Delta_t * A
                RHS = yold + c(t[n])*Delta_t
                y[0:unknowns,n] = GaussElimPivotSolve(LHS,RHS)
            return t, y
```

Results with $\Delta t = 0.1$ show that error builds over time, but in a different way than forward Euler.

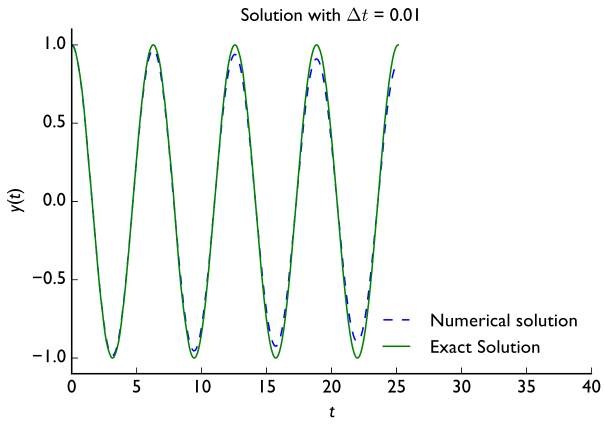


Now the numerical error builds over time, but the error causes the solution to damp out over time. As before, decreasing Δt only delays the onset of the error.

The phase field plot shows that instead of the path growing, backward Euler's path spirals down to zero.



Larger time steps make the solution decay to zero faster. If we take a small time step, the decay is slower but still occurs.



The takeaway here is that first-order accurate methods have errors that build over time. Forward Euler the errors cause the solution to grow, whereas, Backward Euler has the solution damp to zero over time.

17.6.1 Stability for Systems

For a system of initial value problems, the stability condition is derived by looking at the problem

$$\mathbf{y}' = -\mathbf{A}\mathbf{y}.$$

If we multiply both sides of the equation by a left-eigenvector of $\mathbf{A}$, $\mathbf{l}_k$ with associated eigenvector α_k we get

$$\mathbf{u}_k' = -\alpha_k \mathbf{u}_k, \tag{17.10}$$

where $\mathbf{u}_k = \mathbf{l}_k \cdot \mathbf{y}$.

Notice that each row of Eq. (17.10) is identical to the model equation we had for stability for single initial value problems in Eq. (17.7). Therefore, we can replace the α in our equations for the growth rate g by the eigenvalues of the matrix $\mathbf{A}$. Then we can determine under what conditions $|g| \leq 1$ to find a range of stability.

In the example above where

$$\mathbf{A} = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix},$$

the eigenvalues are $\pm i$. Using the growth rate for forward Euler, we get $g_{\mathrm{FE}} = 1 \pm i\Delta t$, which implies

$$|g_{\mathrm{FE}}| = \sqrt{1 + \Delta t^2}.$$

This result means that for any Δt, forward Euler will not be stable because any positive Δt makes $|g_{\mathrm{FE}}| > 1$. This is what we saw in the example. Backward Euler, however, is stable because

$$|g_{\mathrm{BE}}| = \frac{1}{\sqrt{1 + \Delta t^2}},$$

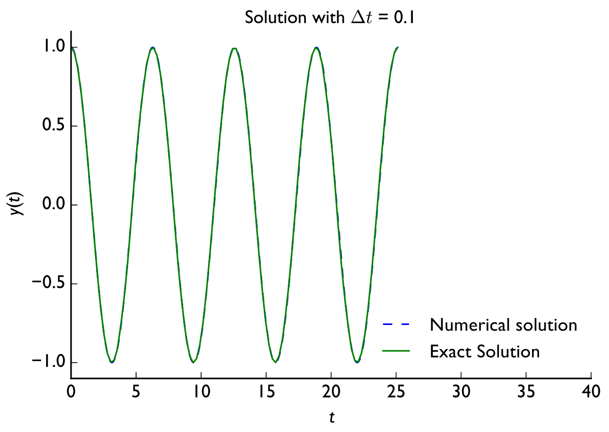
which is less than one for any positive Δt.

17.6.2 Crank–Nicolson for Systems

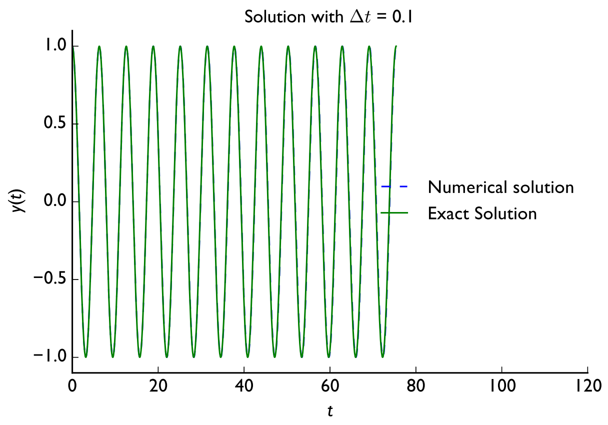
Now we will look at Crank–Nicolson to see how it behaves. A Python implementation of Crank–Nicolson is given next:

```
In [9]: def cn_system(Afunc,c,y0,Delta_t,numsteps):
            """Perform numsteps of the Crank--Nicolson method starting at y0
            of the ODE y'(t) = A(t) y(t) + c(t)
            Args:
                Afunc: function to compute A matrix
                c: nonlinear function of time
                y0: initial condition
                Delta_t: time step size
                numsteps: number of time steps

            Returns:
                a numpy array of the times and a numpy
                array of the solution at those times
            """
            numsteps = int(numsteps)
            unknowns = y0.size
            y = np.zeros((unknowns,numsteps+1))
            t = np.arange(numsteps+1)*Delta_t
            y[0:unknowns,0] = y0
            for n in range(1,numsteps+1):
                yold = y[0:unknowns,n-1]
                A = Afunc(t[n])
                LHS = np.identity(unknowns) - 0.5*Delta_t * A
                A = Afunc(t[n-1])
            RHS = yold +
                  0.5*Delta_t * np.dot(A,yold) + 0.5*(c(t[n-1]) +
                  c(t[n]))*Delta_t
            y[0:unknowns,n] = GaussElimPivotSolve(LHS,RHS)
        return t, y
```

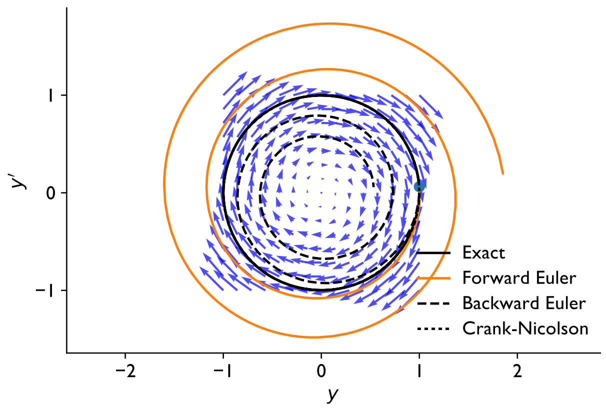
On our test problem Crank–Nicolson solution does not display the significant error that the first-order methods did.



The error build up is not nearly as large of a problem. Even if we look at the solution over a much longer time, the error is not significant:



The phase field plot demonstrates that the accuracy of Crank-Nicolson allows it to not have any strange behavior in the phase field. The numerical solution appears as a circle in the phase field:



One can show that for this problem, Crank–Nicolson is unconditionally stable because $|g_{\text{CN}}| = 1$ for any Δt. This also explains why the solution did not damp or grow over time.

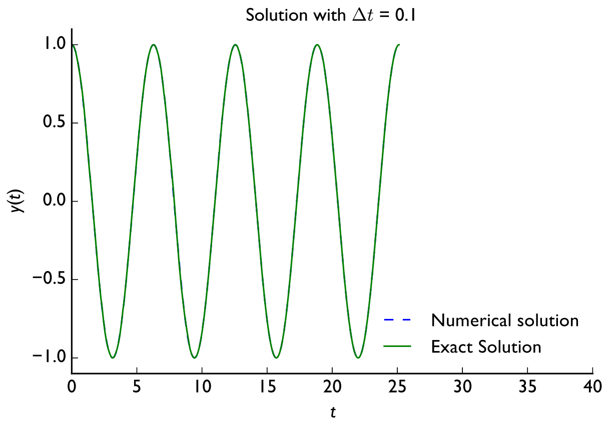
17.6.3 RK4 for Systems

Finally, we generalize our implementation of fourth-order Runge–Kutta to handle systems of equations.

```
In [10]: def RK4_system(Afunc,c,y0,Delta_t,numsteps):
             """Perform numsteps of the 4th-order Runge--Kutta method starting at y0
             of the ODE y'(t) = f(y,t)
             Args:
                 f: function to integrate takes arguments y,t
                 y0: initial condition
                 Delta_t: time step size
                 numsteps: number of time steps

             Returns:
                 a numpy array of the times and a numpy
                 array of the solution at those times
             """
             numsteps = int(numsteps)
             unknowns = y0.size
             y = np.zeros((unknowns,numsteps+1))
             t = np.arange(numsteps+1)*Delta_t
             y[0:unknowns,0] = y0
             for n in range(1,numsteps+1):
                 yold = y[0:unknowns,n-1]
                 A = Afunc(t[n-1])
                 dy1 = Delta_t * (np.dot(A,yold) + c(t[n-1]))
                 A = Afunc(t[n-1] + 0.5*Delta_t)
                 dy2 = Delta_t * (np.dot(A,y[0:unknowns,n-1] + 0.5*dy1)
                                  + c(t[n-1] + 0.5*Delta_t))
                 dy3 = Delta_t * (np.dot(A,y[0:unknowns,n-1] + 0.5*dy2)
                                  + c(t[n-1] + 0.5*Delta_t))
                 A = Afunc(t[n] + Delta_t)
```

```
            dy4 = Delta_t * (np.dot(A,y[0:unknowns,n-1] + dy3) + c(t[n]))
            y[0:unknowns,n] = y[0:unknowns,n-1] +
                              1.0/6.0*(dy1 + 2.0*dy2 + 2.0*dy3 + dy4)
        return t, y
```

Like Crank–Nicolson, RK4 does not display noticeable error on our test problem.



On this simple problem, RK4 appears to be as good as Crank-Nicolson. The stability limit for RK4 can be derived by replacing α with $\pm i$ in Eq. (17.9). The result is that $\Delta t < 2.78529356$ will give a stable, if inaccurate, solution.

17.7 POINT-REACTOR KINETICS EQUATIONS

The simplest model to describe the behavior of the power in a nuclear reactor are the point-reactor kinetics equations (PRKEs) with one delayed neutron precursor group. This is an important model for the time-dependent behavior of nuclear reactors, and other nuclear systems. See the further reading section below for suggestions on background references for this model.

The PRKEs are a system of two ODEs:

$$p'(t) = \frac{\rho - \beta}{\Lambda} p(t) + \lambda C(t),$$
$$C'(t) = \frac{\beta}{\Lambda} p(t) - \lambda C(t).$$

Here p is the number of free neutrons in the reactor, ρ is the reactivity

$$\rho = \frac{k-1}{k},$$

for k the effective multiplication factor, β is the fraction of neutrons produced by fission that are delayed, Λ is the mean time between neutron generations in the reactor, λ is the mean number of delayed neutrons produced per unit time, and C is the number of delayed neutron precursors (nuclides that will emit a neutron).

In our notation

$$\mathbf{y}(t) = (p(t), C(t))^t,$$
$$\mathbf{A}(t) = \begin{pmatrix} \frac{\rho-\beta}{\Lambda} & \lambda \\ \frac{\beta}{\Lambda} & -\lambda \end{pmatrix},$$

and

$$\mathbf{c}(t) = \mathbf{0}.$$

Recall that at steady-state, (i.e. $p'(t) = C'(t) = 0$), to have a non-trivial solution $\rho = 0$, i.e., the solution is critical. In this case, the solution to this system is

$$p = \frac{\lambda \Lambda}{\beta}.$$

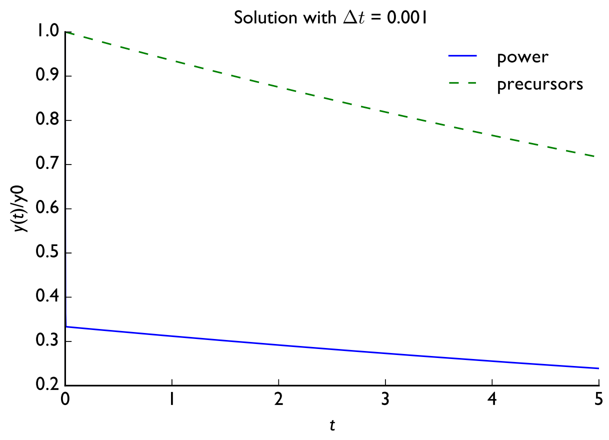
We will need to know this for some of our test cases.

Also, some typical values for the constants are $\beta = 750 \times 10^{-5}$, $\Lambda = 2 \times 10^{-5}$ s, $\lambda = 10^{-1}$ s^{-1}.

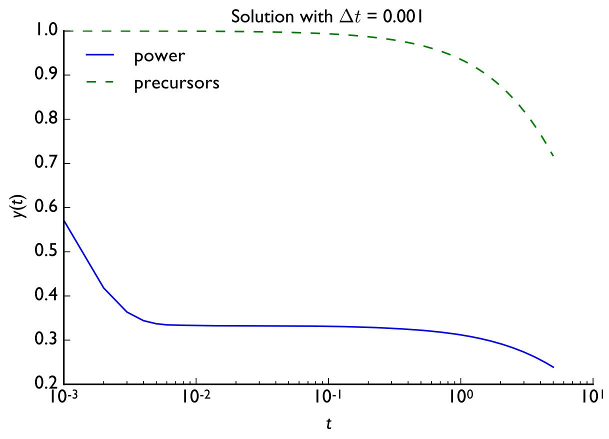
17.7.1 Rod-Drop

Now we simulate the situation where we have a large amount of negative reactivity (a dollar (\$) is a reactivity unit, where one dollar of reactivity is equal to the value of β for the system) inserted into a critical reactor, $\rho = -2\$ = -2\beta$. This is the scenario that occurs when a control is rapidly inserted into the reactor. The tricky part here is setting up the function for $\mathbf{A}(t)$. We will start with RK4.

```
In [32]: #Set up A
         beta = 705.0e-5
         Lambda = 2.0e-5 #s
         l = 1.0e-1 #s**-1
         rho = -2.0*beta
         Afunc = lambda t: np.array([((rho-beta)/Lambda,l),
                                     (beta/Lambda,-l)])
         #set up c
         c = lambda t: np.zeros(2)
         #set up inital vector
         y0 = np.array([1,beta/(l*Lambda)])
         Delta_t = 0.001 #1 millisecond
         t_final = 5
         t,y = RK4_system(Afunc,c,y0,Delta_t,t_final/Delta_t)
```

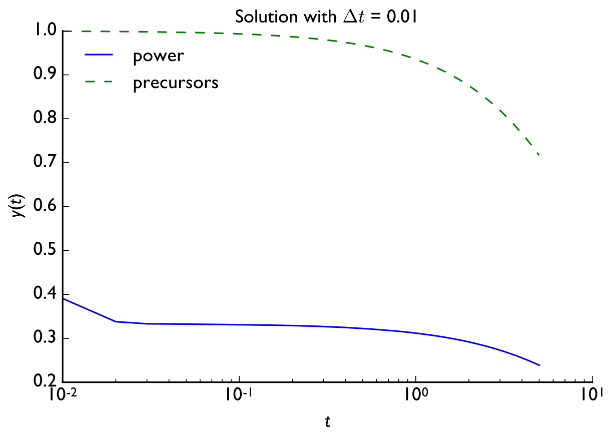


We look at the results on a semilog scale so we can see the prompt jump: the rapid decrease in the neutron population after the control rod is inserted.



Notice that our time step is smaller than the prompt jump (as can be seen on the semilog scale). With a larger time step, RK4 has problems. In fact with a time step of $\Delta t = 0.01$ s, the power is 10^{303} at 12 s. It seems that this time step is beyond the stability limit for RK4. This can be checked by evaluating the eigenvalues of the $\mathbf{A}$ matrix for this problem and using the formula for g_{RK4}. The stability limit is $\Delta t \leq 0.00247574$ s.

Using backward Euler, we can take this larger time step safely:

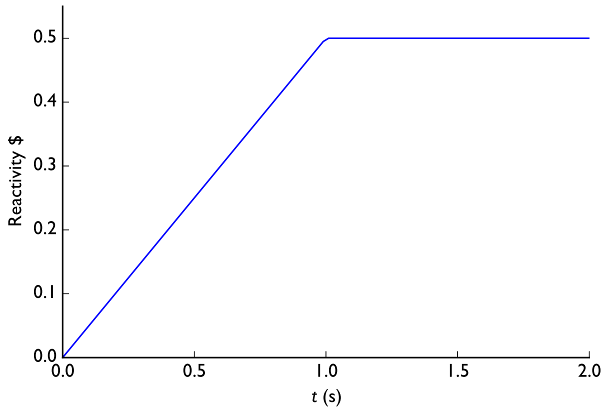


Yes, backward Euler works with this time step size, where RK4 could not. It is here that we see the benefit of implicit methods: with backward Euler we need 10 times fewer time steps to get the solution. The price of this was that we had to solve a linear system at each step.

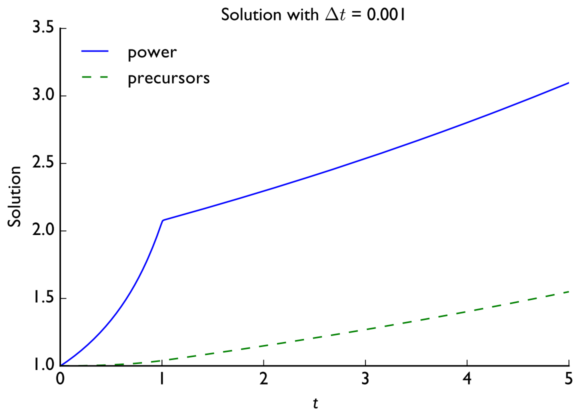
17.7.2 Linear Reactivity Ramp

In this case we will insert positive reactivity linearly, this could be done by slowly removing a control rod, and see what happens. We will have a linear ramp between $\rho = 0$ and $\rho = 0.5\$$ over 1 second.

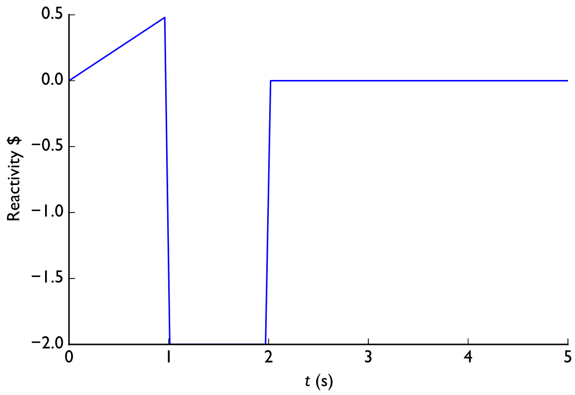
```
In [33]: #reactivity function
         rho = lambda t: (t<1)*beta*0.5*t + beta*0.5*(t>=1)
```



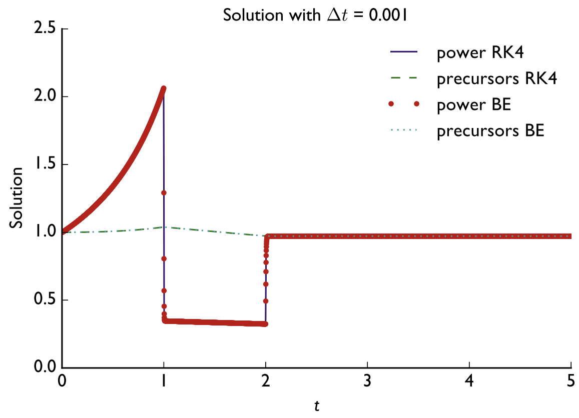
The solution with backward Euler demonstrates two different growth rates: one corresponding to the ramp, and another once the reactivity stops increasing.



To make the scenario more complicated we will now look at a positive reactivity ramp that is then has the control rod completely re-inserted at $t = 1$, and then instantly brought back to critical at $t = 2$. The reactivity for this scenario looks like



We will compare backward Euler and RK4 on this problem with the same time step.



The prompt jump is a wild phenomenon: the solution changes very rapidly during it and the methods still keep up with a small enough time step. In this example, we cannot see on the graph a difference between the solutions. Additionally, the precursors change very slowly over the entire simulation.

CODA

We have discussed 4 classical techniques for integrating initial value problems: forward and backward Euler, Crank–Nicolson, and fourth-order Runge Kutta. Through these methods we discussed the issues of order of accuracy, stability, oscillatory solutions, and the building of error.

Initial value problems are only the first of the type of ODEs that we can solve using the techniques we have covered up to this point. To solve initial value problems we appealed to methods of doing numerical integration. In the next chapter we will solve boundary value problems and use the techniques to evaluate derivatives numerically we discussed previously.

FURTHER READING

Although we have covered the majority of methods used in practice for the solution of initial value problems, there are some modifications that are made to the implementation that makes these methods more powerful. For example, adaptive Runge–Kutta methods take time steps that are smaller when the solution changes rapidly, and larger when the solution is slowly varying. Also there are other methods that we did not cover, including the backward differentiation formulas, known as BDF formulas. The BDF-2 method is a second-order implicit method, but it is less oscillatory than Crank-Nicolson.

For background on the PRKEs, see the reactor physics texts previously mentioned by Lewis [7] and Stacey [10]. There has been much work on developing the best numerical methods for the PRKEs. The recent method developed by Ganapol may be the most effective [21].

PROBLEMS

Short Exercises

Using the five methods we have presented in this chapter to compute the solution to the following problems using the step sizes $\Delta t = 0.5, 0.1, 0.05, 0.01$. Give an explanation for the behavior of each solution.

17.1. $y'(t) = -\left[y(t)\right]^2$ for $t \in [1, 10]$, and $y(1) = 0.5$. The solution to this problem is $y(t) = 1/(t+1)$. *Hint: You can make the substitution* $\hat{t} = t - 1$ *and reformulate the initial condition in terms of* $\hat{t} = 0$.

17.2. $y'(t) = \sin(t)/t$ for $t \in [0, 10]$, and $y(0) = 0$. The solution to this problem is $y(t) = \mathrm{Si}(t)$, where $\mathrm{Si}(t)$ is the Sine integral function.

17.3. $y''(t) = (t-20)y$ for $t \in [0, 24]$ and $y(0) = -0.176406127078$, and $y'(0) = 0.892862856736$. The true solution to this problem is $y(t) = \mathrm{Ai}(t-20)$, where $\mathrm{Ai}(t)$ is the Airy function.

Programming Projects

1. Point Reactor Kinetics

Assume one group of delayed neutrons and an initially critical reactor with no extraneous source. The point reactor kinetics equations under these assumptions are:

$$\frac{dP}{dt} = \frac{\rho - \beta}{\Lambda} P + \lambda C,$$
$$\frac{dC}{dt} = \frac{\beta}{\Lambda} P - \lambda C,$$

with initial conditions: $P(0) = P_0$ and $C(0) = C_0 = \frac{\beta P_0}{\lambda \Lambda}$. Other useful values are $\beta = 750$ pcm, $\Lambda = 2 \times 10^{-5}$ s, $\lambda = 10^{-1}$ s^{-1}.

At time $t = 0$ a control rod is instantaneously withdrawn and the inserted reactivity is $\rho = 50$ pcm $= 50 \times 10^{-5}$. Write a Python code to solve the PRKEs. Do not assume that reactivity is a constant, rather allow it to be a function of t so that you can do a ramp or sinusoidal reactivity.

Your code will utilize the following methods:

- Forward Euler
- Backward Euler
- Fourth-order Runge Kutta
- Crank–Nicolson.

For each method provide plots for $t = 0 \dots 5$ is using $\Delta t = \{1, 3, 5, 6\}$ ms.

- One plot with all the methods per time step size
- One plot with all time step sizes per method.

Make the axes reasonable—if a method diverges do not have the scale go to $\pm\infty$.

Comment your results and explain their behavior using what you know about the accuracy and stability of the methods, including their stability limits. The explanation should include justifications for convergence/divergence or the presence of oscillations. To answer these questions you should look at the analytical solutions of the equations.

REACTIVITY RAMP

A positive reactivity ramp is inserted as follows:

$$\rho(t) = \min(\rho_{\max} t, \rho_{\max}).$$

Pick a method and a time step size of your choice. Solve this problem for $\rho_{\max} = 0.5\beta$ and $\rho_{\max} = 1.1\beta$. Explain your choice and your findings.

SINUSOIDAL REACTIVITY

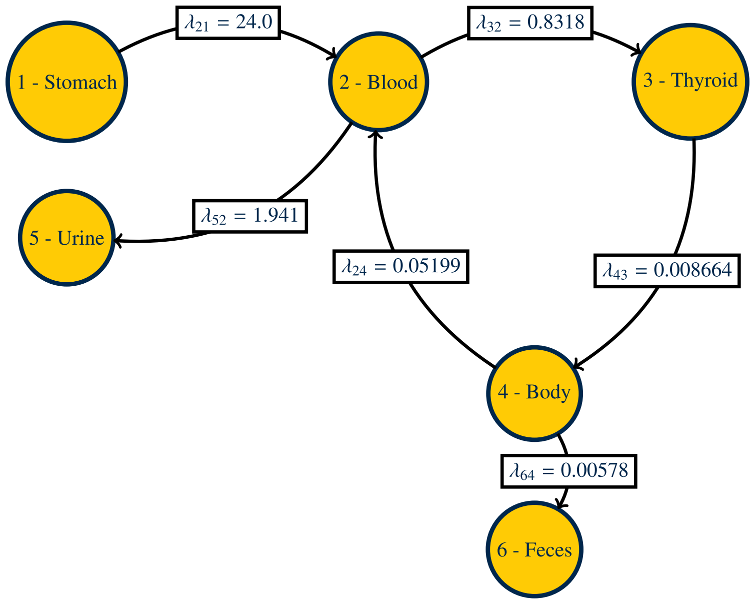
Try

$$\rho(t) = \rho_0 \sin(\omega t).$$

See if the reactor remains stable for any amplitude ρ_0 and/or frequency ω.

2. Iodine Ingestion

The diagram below shows the metabolic compartment model for the uptake of iodine in the thyroid gland of an adult human. The goal is to calculate the fraction of the total ingested activity present in the blood (compartment 2) and the thyroid gland (compartment 3) as a function of time for an ingestion, at time 0, of 1 Bq of ^{131}I (half-life of 8.04 days). In the diagram below λ_{21} represents the biological transfer into compartment 2 from compartment 1. All λ's are expressed in days^{-1}.



- Write down the physical equations governing the conservation of iodine in each of the following compartments: stomach, blood, thyroid, and body. This will form a system of coupled ODEs. Remember that the substance is removed from a compartment by both radioactive decay and transfer between biological compartments.
- Why is it enough to look at the conservation of iodine in stomach, blood, thyroid, and body to get the answer?
- Your system of ODEs should have the following form:

$$Y' = AY + B(t),$$

where $B(t)$ represents the external source terms in your conservation laws. Only the component corresponding to the stomach of B is nonzero. This component is the following function:

$$h(t) = \begin{cases} 1 & \text{if } t = 0 \\ 0 & \text{otherwise} \end{cases}.$$

In practice you will let h be equal to $1/\Delta t$ in the first time step so that $\int_0^{\Delta t} h(t)\,dt = 1$. After that h will always be zero.

- Write a code to solve this system of ODEs. The function B should be a function of t in your code. Your code will utilize the following methods:
  - Forward Euler
  - Backward Euler
  - Fourth-order Runge–Kutta
  - Crank–Nicolson.

  For each method provide plots for $t = 0 \ldots 1$ day using the following step sizes:
  - $0.5/24$ hr
  - $1/24$ hr
  - $1.4/24$ hr
  - $2/24$ hr.

  Note: all λ's are in days^{-1} so be careful about how you express Δt in your code.
  - One plot with all the methods per time step size
  - One plot with all time step sizes per method.

  Make the axes reasonable—if a method diverges don't have the scale go to $\pm\infty$.
  Comment your results and explain their behavior using what you know about the accuracy and stability of the methods, including their stability limits. The explanation should include justifications for convergence/divergence or the presence of oscillations. To answer these questions you should look at the analytical solutions of the equations.