App.E: Programming of differential equations

Joakim Sundnes[1,2]

[1]Simula Research Laboratory
[2]University of Oslo, Dept. of Informatics

Nov 6, 2019

Ordinary differential equations (ODEs) are widely used in science and engineering, in particular for modeling dynamic processes. While simple ODEs can be solved with analytical methods, non-linear ODEs are generally not possible to solve in this way, and we need to apply numerical methods. In this chapter we will see how we can program general numerical solvers that can be applied to any ODE. We will first consider scalar ODEs, i.e. ODEs with a single equation and a single unknown, and then later extend the ideas to systems of coupled ODEs. Understanding the concepts of this chapter is useful not only for programming your own ODE solvers, but also for using a wide variety of general-purpose ODE solvers available both in Python and other programming languages.

1 Creating a general-purpose ODE solver

When solving ODEs analytically one will typically consider a specific ODE or a class of ODEs, and try to derive a formula for the solution. In this chapter we want to implement numerical solvers that can be applied to any ODE, not restricted to a single example or a particular class of equations. For this purpose, we need a general abstract notation for an arbitrary ODE. We will write the ODEs on the following form:

$$u'(t) = f(u(t), t), \tag{1}$$

which means that the ODE is fully specified by the definition of the right hand side function $f(u, t)$. Examples of this function may be:

$$\begin{aligned}
f(u,t) &= \alpha u, \quad \text{exponential growth}\\
f(u,t) &= \alpha u\left(1 - \frac{u}{R}\right), \quad \text{logistic growth}\\
f(u,t) &= -b|u|u + g, \quad \text{falling body in a fluid}
\end{aligned}$$

Notice that for generality we write all these right hand sides as functions of both u and t, although the mathematical formulations only involve u. It will become clear later why such a general formulation is useful. Our aim is now to write functions and classes that take f as input, and solve the corresponding ODE to produce u as output.

Finite difference approximation of the derivative turns an ODE into a difference equation. All the numerical methods we will considered here rely on approximating the derivatives in the equation $u' = f(u,t)$ by finite differences. Assume that we have computed u at discrete time points $t_0, t_1, \ldots, t_k$. At t_k we have the ODE

$$u'(t_k) = f(u(t_k), t_k),$$

and we can now approximate $u'(t_k)$ by a forward finite difference;

$$u'(t_k) \approx \frac{u(t_{k+1}) - u(t_k)}{\Delta t}.$$

Inserting this approximation into the ODE at $t = t_k$ yields the following equation

$$\frac{u(t_{k+1}) - u(t_k)}{\Delta t} = f(u(t_k), t_k),$$

which we recognize as a difference equation for computing $u(t_{k+1})$ from the known value $u(t_k)$ are known. We can rearrange the terms to obtain the explicit formula

$$u(t_{k+1}) = u(t_k) + \Delta t f(u(t_k), t_k).$$

This is known as the *forward Euler method*, and is the simplest numerical method for solving and ODE. As with the difference equations considered in Appendix A, we start from the known initial condition $u(t_0)$, and apply the formula repeatedly to compute $u(t_1)$, $u(t_2)$, $u(t_3)$ and so forth. We can simplify the formula by using the notation for difference equations introduced in Appendix A. If we let u_k denote the numerical approximation to the exact solution $u(t)$ at $t = t_k$, the difference equation can be written as

$$u_{k+1} = u_k + \Delta t f(u_k, t_k). \tag{2}$$

This is a regular difference equation which can be solved using arrays and a for-loop, just as we did for the other difference equations in Appendix A.

An ODE needs an initial condition. In mathematics, an initial condition for u must be specified to have a unique solution of equation (1). When solving the equation numerically, we need to set u_0 in order to start our method and compute a solution at all. As an example, consider the very simple ODE

$$u' = u.$$

This equation has the general solution $u = Ce^t$ for any constant C, so it has an infinite number of solutions. Specifying an initial condition $u(0) = u_0$ gives

$C = u_0$, and we get the unique solution $u = u_0 e^t$. When solving the equation numerically, we start from our known u_0, and apply formula (2) repeatedly:

$$u_1 = u_0 + \Delta t u_0$$
$$u_2 = u_1 + \Delta t u_1$$
$$u_3 = u_2 + \dots$$

To start with a concrete and simple example, let us first implement this solution algorithm in a program. The algorithm may be sketclike this; for a given time step Δt (`dt`) and number of time steps n, preform the following steps

1. Create arrays `t` and `u` of length $n + 1$
2. Set initial condition: `u[0]` $= U_0$, `t[0]=0`
3. For $k = 0, 1, 2, \dots, n - 1$:
   - `t[k+1] = t[k] + dt`
   - `u[k+1] = (1 + dt)*u[k]`

The Python implementation of this algorithm may look like

```
import numpy as np
import matplotlib.pyplot as plt

dt = 0.2
u0 = 1
T = 4
n = int(T/dt)

t = np.zeros(n+1)
u = np.zeros(n+1)

t[0] = 0
u[0] = 0
for k in range(n):
    t[k+1] = t[k] + dt
    u[k+1] = (1 + dt)*u[k]

plt.plot(t,u)
plt.show()
```

The solution is shown in Figure 1, for two different choices of the time step Δt. We see that the approximate solution improves as Δt is reduced, although both the solutions are quite inaccurate. However, reducing the time step further will easily create a solution that cannot be distinguished from the exact solution.

Extending the solver to a the general ODE. As stated above, the purpose of this chapter is to create general-purpose ODE solvers, that can solve any ODE written on the form $u' = f(u, t)$. This requires a very small modification of the algorithm above;

1. Create arrays `t` and `u` of length $n + 1$

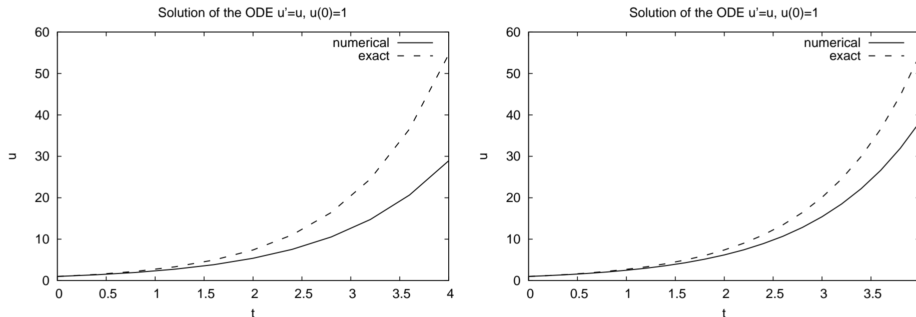

Figure 1: Solution of $u' = u, u(0) = 1$ with $\Delta t = 0.4$ and $\Delta t = 0.2$.

2. Set initial condition: `u[0]` $= u_0$, `t[0]=0`
3. For $k = 0, 1, 2, \ldots, n-1$:
    - `u[k+1] = u[k] + dt*f(u[k], t[k])` (the only change!)
    - `t[k+1] = t[k] + dt`

We see that the only change of the algorithm is in the formula for computing `u[k+1]` from `u[k]`. For the case considerd above we had $f(u,t) = u$, and to create a general-purpose ODE solver we simply replace `u[k]` with the more general `f(u[k],t[k])`. The implementation of the algorithm in a Python function may look like:

```
def ForwardEuler(f, U0, T, n):
    """Solve u'=f(u,t), u(0)=U0, with n steps until t=T."""
    import numpy as np
    t = np.zeros(n+1)
    u = np.zeros(n+1)  # u[k] is the solution at time t[k]

    u[0] = U0
    t[0] = 0
    dt = T/float(n)

    for k in range(n):
        t[k+1] = t[k] + dt
        u[k+1] = u[k] + dt*f(u[k], t[k])

    return u, t
```

This simple function can solve any ODE written on the form (1)! The right hand side function $f(u,t)$ needs to be implemented as a Python function, and then passed as an argument to `ForwardEuler` together with the initial condition, the stop time `T` and the number of time steps.

To illustrate how the `ForwardEuler` function is used, let us apply it to the same problem above; $u' = u$, $u(0) = 1$, for $t \in [0,4]$. The following code uses the `ForwardEuler` function to solve this problem:

```python
def f(u, t):
    return u

U0 = 1
T = 3
n = 30
u, t = ForwardEuler(f, U0, T, n)
```

We see that the `ForwardEuler` function returns the two arrays `u` and `t`, which can then be plotted or processed further as we want. One thing worth noticing in this implementation is the definition of the right hand side function `f`. As we commented above, we always write this function with two arguments `u` and `t`, although in this case only `u` is used inside the function. The two arguments are needed because of how the function is used inside `ForwardEuler`, it is called as `f(u[k], t[k])`. If the function was defined as `def f(u):` we would get an error message from this line, but simply writing `def f(u,t):` even if `t` is never used solves this problem.

Now you can solve any ODE! For being only 15 lines of Python code, the capabilities and generality of the `ForwardEuler` function above are quite remarkable. Using this function, we can solve any kind of linear or non-linear ODE, most of which would be impossible to solve using analytical techniques. The general recipe goes as follows:

1. Identify $f(u,t)$ in your ODE
2. Make sure you have an initial condition u_0
3. Implement the $f(u,t)$ formula in a Python function `f(u, t)`
4. Choose the time step Δt or the number of steps n
5. Call `u, t = ForwardEuler(f, U0, T, n)`
6. Plot the solution

It is worth mentioning that the Forward Euler method is the simplest of all ODE solvers, and many will argue that it is not very good. This is partly true, since there are many other methods that are more accurate and more stable when applied to challenging ODEs. We shall look at a few examples of such methods later in this chapter. However, the Forward Euler method is quite suitable for solving most ODEs. If we are not happy with the accuracy we can simply reduce the time step, and in most cases this will give the accuracy we need with a negligible increase in computing time.

The general ODE solver can also be implemented as a class. We can increase the flexibility of the `ForwardEuler` solver function by implementing it as a class. The usage of the class may be as follows:

```python
method = ForwardEuler(f, dt)
u, t = method.solve(T)
plot(t, u)
```

The benefits of using a class instead of a function may not be obvious at this point, but it will become cleare later. For now, let us just look at how the class can be implemented:

- We need to store f, Δt, and the sequences u_k, t_k as attributes
- The steps in the `ForwardEuler` function can be split into two separate methods:
  - the constructor (`__init__`)
  - `solve`, which runs the for loop to solve the ODE and returns the solution

In addition to these methods, it may be convenient to implement the formula for advancing the solution as a separate method `advance`. In this way it becomes very easy to implement new numerical methods, since we typically only need to change the `advance` method. A first version of the solver class may look as follows:

```
import numpy as np

class ForwardEuler_v1:
    def __init__(f, U0, T, n):
        self.f, self.U0, self.T, self.n = f, U0, T, n
        self.dt = T/float(n)
        self.u = np.zeros(self.n+1)
        self.t = np.zeros(self.n+1)

    def solve(self, T):
        """Compute solution for 0 <= t <= T."""
        self.u[0] = float(self.U0)
        self.t[0] = float(0)

        for k in range(self.n):
            self.k = k
            self.t[k+1] = self.t[k] + self.dt
            self.u[k+1] = self.advance()
        return self.u, self.t

    def advance(self):
        """Advance the solution one time step."""
        # Create local variables to get rid of "self." in
        # the numerical formula
        u, dt, f, k, t = self.u, self.dt, self.f, self.k, self.t

        unew = u[k] + dt*f(u[k], t[k])
        return unew
```

This class does essentially the same tasks as the `ForwardEuler` function above. The main advantage of the class implementation is the increased flexibility that comes with the `advance` method. As we shall see later, implementing a different numerical method typically only requires implementing a new version of this method, while all other code can be reused.

We can also use a class to hold the right-hand side $f(u,t)$, which is particularly convenient for functions with parameters. Consider for instance the model for logistic growth;

$$u'(t) = \alpha u(t)\left(1 - \frac{u(t)}{R}\right), \quad u(0) = U_0, \quad t \in [0, 40],$$

which is a commonly used model in biology. The right hand side function has two parameters α and R, but if we want to solve it using our `ForwardEuler` function or class, it must be implemented as a function of u and t only. As we discussed in Chapter 7, a class with a call method provides a very flexible implementation of such a function, since we can set the parameters as attributes in the constructor and use them inside the `__call__` method:

```
class Logistic:
    def __init__(self, alpha, R, U0):
        self.alpha, self.R, self.U0 = alpha, float(R), U0

    def __call__(self, u, t):   # f(u,t)
        return self.alpha*u*(1 - u/self.R)
```

The main program for solving the logistic growth problem may now look like:

```
problem = Logistic(0.2, 1, 0.1)
method = ForwardEuler_v1(problem,problem.U0,40,401)
u, t = method.solve(T)
```

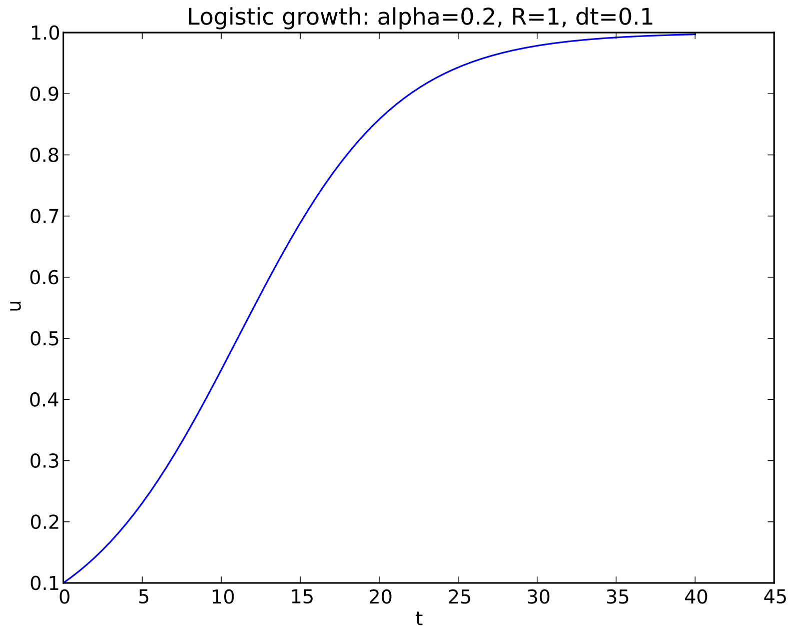

Figure 2: Solution of the logistic growth model.

2 A class hierarchy of ODE solvers

As mentioned above, the Forward Euler method is not the most sophisticated ODE solver, although it is sufficiently accurate for most of the appliactions we will consider. Many alternative methods exist, with better accuracy and stability than Forward Euler. One very popular class of ODE solvers is known as Runge-Kutta methods. The simplest example of a Runge-Kutta method is in fact the Forward Euler method;

$$u_{k+1} = u_k + \Delta t\, f(u_k, t_k),$$

which is an example of an first-order explicit Runge-Kutta method with a single stage. Another popular method is the fourth-order method:

$$u_{k+1} = u_k + \frac{1}{6}\left(K_1 + 2K_2 + 2K_3 + K_4\right)$$

where the $K_1, \ldots, K_4$ are intermediate variables defined by

$$
\begin{aligned}
K_1 &= \Delta t\, f(u_k, t_k), \\
K_2 &= \Delta t\, f(u_k + \frac{1}{2}K_1, t_k + \frac{1}{2}\Delta t), \\
K_3 &= \Delta t\, f(u_k + \frac{1}{2}K_2, t_k + \frac{1}{2}\Delta t), \\
K_4 &= \Delta t\, f(u_k + K3, t_k + \Delta t).
\end{aligned}
$$

This is called a four-stage Runge-Kutta method, and we may observe that the update formula is quite similar to Forward Euler. The difference is that we compute the intermediate variables $K_1, \ldots, K_4$, and advance the solution using a weighted average of these rather then simply using $\Delta t\, f(u_k, t_k)$. This improves the accuracy of the approximation. All Runge-Kutta methods follow the same idea, and differ only in the number of stages and the coefficients and weights used for computing the K-values and advancing the solution.

We now want to implement the fourth-order Runge-Kutta method as a class, similar to the implementation of the Forward Euler class introduced above. When inspecting the `ForwardEuler_v1` class, we quickly observe that most of the code is not specific to the particular ODE method, but are common to all ODE solvers. For instance, we always need to create an array for holding the solution, and the general solution method using a for-loop is the same. The only difference is how the solution is advanced from one step to the next. Recalling the ideas of Object-Oriented Programming from Chapter 9, it becomes obvious that a class hierarchy is very convenient for implementing a collection of ODE solvers. In this way we can collect common code in a superclass, and rely on inheritance to avoid code duplication. The superclass can handle most of the more "administrative" steps of the ODE solver, such as

- Storing the solution u_k and the corresponding time levels t_k, $k = 0, 1, 2, \ldots, n$
- Storing the right-hand side function $f(u, t)$
- Storing and applying initial condition
- Running the loop over all time steps

The implementation of the superclass should follow the principles introduced in Chapter 9:

- Common data and functionality are placed in the superclass `ODESolver`
- The parts that differ between different solvers are isolated in a single method (i.e. `advance`)
- Subclasses, e.g., `ForwardEuler`, just implement the specific numerical formula in `advance`

The implamentation of the superclass may look like

```python
class ODESolver:
    def __init__(self, f):
        self.f = f

    def advance(self):
        """Advance solution one time step."""
        raise NotImplementedError  # implement in subclass

    def set_initial_condition(self, U0):
        self.U0 = float(U0)

    def solve(self, time_points):
        self.t = np.asarray(time_points)
        self.u = np.zeros(len(self.t))
        # Assume that self.t[0] corresponds to self.U0
        self.u[0] = self.U0

        # Time loop
        for k in range(n-1):
            self.k = k
            self.u[k+1] = self.advance()
        return self.u, self.t
```

Comparing with the `ForwardEuler_v1` class above, we see that we have made some minor modifications. First, we have introduced a method `set_initial_codition` instead of passing the initial condition as an argument to the constructor. Second, the `solve`-method takes an array of time points as argument, instead of a final time `T`. These choices offer some extra flexibility compared with `ForwardEuler_v1`, and are more convenient for some applications but probably not for all. As usual, there are multiple ways to solve a given task, and as programmers we need to make some choices.

With the `ODESolver` superclass at hand, the implementation of a `ForwardEuler` class becomes very simple:

```python
class ForwardEuler(ODESolver):
    def advance(self):
        u, f, k, t = self.u, self.f, self.k, self.t

        dt = t[k+1] - t[k]
        unew = u[k] + dt*f(u[k], t)
        return unew
```

Similarly, the fourth-order Runge-Kutta method can also be a subclass with a single method:

```python
class RungeKutta4(ODESolver):
    def advance(self):
        u, f, k, t = self.u, self.f, self.k, self.t

        dt = t[k+1] - t[k]
        dt2 = dt/2.0
        K1 = dt*f(u[k], t)
        K2 = dt*f(u[k] + 0.5*K1, t + dt2)
        K3 = dt*f(u[k] + 0.5*K2, t + dt2)
        K4 = dt*f(u[k] + K3, t + dt)
        unew = u[k] + (1/6.0)*(K1 + 2*K2 + 2*K3 + K4)
        return unew
```

The use of these classes is quite similar to the first version of the Forward Euler class. Considering the same simple ODE used above; $u' = u$, $u(0) = 1$, $t \in [0, 3]$, $\Delta t = 0.1$, the code looks like:

```
import numpy as np
import matplotlib.pyplot as plt

from ODESolver import ForwardEuler, RungeKutta4
def test(u, t):
    return u

time_points = np.linspace(0, 3, 31)

method1 = ForwardEuler(test)
method1.set_initial_condition(U0=1)
u1, t1 = method1.solve(time_points)
plt.plot(t1, u1)

method2 = RungeKutta4(test)
method2.set_initial_condition(U0=1)
u2, t2 = method1.solve(time_points)
plt.plot(t2, u2)

plt.show()
```

This code will solve the same equation using both methods, and plot the solutions in the same window. Experimenting with the time step size should reveal the difference in accuracy between the two methods.

The solver hierarchy makes it convenient to extend and improve the implementation. An obvious advantage of the implementing the solvers as a class hierarchy is that we can easily introduce new methods as subclasses, with very little new code required. Another advantage is that we can easily improve the functionality of the superclass, and the improvements will automatically apply to all our solvers. As an example, we may introduce a form of automatic termination of our solution process, which checks the solution at each steps and stops the loop if a certain condition is fulfilled. For instance, for some equations it may be useful to stop the solution process when the solution becomes zero, i.e. when $u < 10^{-7} \approx 0$. To include such functionality in our solvers, we can modify the `solve` method take two arguments; `solve(time_points, terminate)`. The second argument should be a user-defined function `terminate(u, t, step_no)` that is called at every time step, and returns `True` when the time stepping should be terminated. The extended `solve` method may be implemented as follows:

```
    def solve(self, time_points, terminate=None):
        if terminate is None:
            terminate = lambda u, t, step_no: False

        # Assume that self.t[0] corresponds to self.U0
        self.u[0] = self.U0

        # Time loop
        for k in range(n-1):
            self.k = k
```

```
            self.u[k+1] = self.advance()
            if terminate(self.u, self.t, self.k+1):
                break  # terminate loop over k
        return self.u[:k+2], self.t[:k+2]
```

Notice the default value provided for `terminate`, which makes this argument optional, and the if-test inside the method ensures that it works as expected if no `terminate` argument is provided. With the class hierarchy, an extension like this, implemented in the superclass, is automatically available in all solver subclasses. The following code provides an example of using the improved class hierarchy:

```
import numpy as np
import matplotlib.pyplot as plt

from ODESolver import RungeKutta4
def decay(u, t):
    return -u

def terminate(u, t, step_no):
    eps = 1.0E-6                     # small number
    return abs(u[step_no,0]) < eps   # close enough to zero?

time_points = np.linspace(0, 10, 101)

method = ForwardEuler(decay)
method.set_initial_condition(U0=1)
u1, t1 = method.solve(time_points, terminate)
plt.plot(t1, u1)

plt.show()
```

3 Systems of ordinary differential equations

So far we have only considered ODEs with a single solution component, often called scalar ODEs. Many interesting processes can be described by systems of ODEs, i.e., mutlipe ODEs where the right hand side of one depends on the solution of the others. Such equation systems are also referred to as vector ODEs. One simple example is

$$
\begin{aligned}
u' &= v, u(0) &= 1 \\
v' &= -u, v(0) &= 0.
\end{aligned}
$$

The solution of this system is $u = cos(t), v = sin(t)$ which can easily be verified by insterting the solution into the equations and initial conditions. For more general cases, it is usually even more difficult to find analytical solutions of ODE systems than of scalar ODEs, and numerical methods are usually required. The purpose of this last part of the chapter is to extend the solvers introduced above to be able to solve systems of ODEs. We shall see that such extension requires relatively small modifications of the code.

We want to develop general software that can be applied to any vector ODE or scalar ODE, and for this purpose it is useful to introduce general mathematical

notation. We have n unknowns

$$u^{(0)}(t), u^{(1)}(t), \ldots, u^{(n-1)}(t)$$

in a system of n ODEs:

$$\begin{aligned}
\frac{d}{dt}u^{(0)} &= f^{(0)}(u^{(0)}, u^{(1)}, \ldots, u^{(n-1)}, t), \\
\frac{d}{dt}u^{(1)} &= f^{(1)}(u^{(0)}, u^{(1)}, \ldots, u^{(n-1)}, t), \\
\vdots &= \qquad\qquad\qquad\qquad \vdots \\
\frac{d}{dt}u^{(n-1)} &= f^{(n-1)}(u^{(0)}, u^{(1)}, \ldots, u^{(n-1)}, t).
\end{aligned}$$

To simplify the notation (and later the implementation), we collect both the solutions $u^{(i)}(t)$ and right-hand side functions $f^{(i)}$ in vectors;

$$u = (u^{(0)}, u^{(1)}, \ldots, u^{(n-1)}),$$

and

$$f = (f^{(0)}, f^{(1)}, \ldots, f^{(n-1)}).$$

Note that f is now a vector-valued function. It takes $n+1$ input arguments (t and the n components of u) and returns a vector of n values. The ODE system can now be written

$$u' = f(u, t), \quad u(0) = u_0$$

where u and f are vectors and u_0 is a vector of initial conditions. We see that we use exactly the same notation as for scalar ODEs, and whether we solve a scalar or system of ODEs is determined by how we define f and the initial condition u_0. This general notation is completely standard in text books on ODEs, and we can easily make the Python implementation just as general.

How can we make the ODESolver class work for systems of ODEs? The `ODESolver` class above was written for a scalar ODE. We now want to make it work for a system $u' = f$, $u(0) = U_0$, where u, f and U_0 are vectors (arrays). To identify how the code needs to be changed, let us start with the simplest method. Applying the forward Euler method to a system of ODEs yields the update formula

$$\underbrace{u_{k+1}}_{\text{vector}} = \underbrace{u_k}_{\text{vector}} + \Delta t \underbrace{f(u_k, t_k)}_{\text{vector}}.$$

In Python code, this may look identical to the version for scalar ODEs;

```
u[k+1] = u[k] + dt*f(u[k], t)
```

with the important difference that both `u[k]` and `u[k+1]` are arrays. Since these are arrays, the solution `u` must be a two-dimensional array, and `u[k]`,`u[k+1]`, etc. are the rows of this array. The function `f` must return a one-dimensional

array, containing all the right-hand sides $f^{(0)}, \ldots, f^{(n-1)}$. To get a better feel for how these arrays look and how they are used, we may compare the array holding the solution of a scalar ODE to that of a system of two ODEs. For the scalar equation, both `t` and `u` are one-dimensional NumPy arrays, and indexing into `u` gives us numbers, representing the solution at each time step:

```
t = [0.  0.4 0.8 1.2 (...) ]

u = [ 1.0 1.4  1.96 2.744  (...)]

u[0] = 1.0
u[1] = 1.4

(...)
```

In the case of a system `t` is the same, but `u` is now a two-dimensional array. Indexing into it yields one-dimensional arrays of length two, which are the two solution components at each time step:

```
u = [[1.0 0.8][1.4 1.1] [1.9 2.7] (...)]

u[0] = [1.0 0.8]
u[1] = [1.4 1.1]

(...)
```

To make a generic `ODESolver` class that works both with scalars and systems, we need to make the following changes:

- Ensure that `f(u,t)` always returns an array.
- Inspect U_0 to see if it is a number or a list/array/tuple and make the corresponding `u` a one-dimensional or two-dimensional array

If these two items are handled and initialized correctly, the rest of the code from above will work with no modifications. The extended superclass implementation may look like:

```python
class ODESolver:
    def __init__(self, f):
        # Wrap user's f in a new function that always
        # converts list/tuple to array (or let array be array)
        self.f = lambda u, t: np.asarray(f(u, t), float)

    def set_initial_condition(self, U0):
        if isinstance(U0, (float,int)):  # scalar ODE
            self.neq = 1                 # no of equations
            U0 = float(U0)
        else:                            # system of ODEs
            U0 = np.asarray(U0)
            self.neq = U0.size           # no of equations
        self.U0 = U0

    def solve(self, time_points, terminate=None):
        if terminate is None:
            terminate = lambda u, t, step_no: False

        self.t = np.asarray(time_points)
```

```
n = self.t.size
if self.neq == 1:  # scalar ODEs
    self.u = np.zeros(n)
else:              # systems of ODEs
    self.u = np.zeros((n,self.neq))

# Assume that self.t[0] corresponds to self.U0
self.u[0] = self.U0

# Time loop
for k in range(n-1):
    self.k = k
    self.u[k+1] = self.advance()
    if terminate(self.u, self.t, self.k+1):
        break  # terminate loop over k
return self.u[:k+2], self.t[:k+2]
```

Some parts of this code are worth some comments. First, the constructor looks almost as in the scalar case, but it will convert any `f` that returns a list or tuple to a function returning a NumPy array. This modification is not strictly needed, since we could just assume that the user implements `f` to return an array, but it makes the class more robust and flexible. Similar tests are included in the `set_initial_condition`, to make sure that `U0` is either a single number (`float`) or a NumPy array, and to set the attribute `self.neq` to hold the number of equations. Finally, the most essential modification is found in the method `solve`. Here, the `self.neq` attribute is inspected, and `u` is initialized to a one- or two-dimensional array of the correct size. The actual for-loop, as well the implementation of the `advance` method in the subclasses, can be left unchanged.

3.1 Example: ODE model for throwing a ball

To demonstrate the use of the extended `ODESolver` hierarchy, let us consider a system of ODEs describing the trajectory of a ball. From Newton's 2nd law we get the equations

$$
\begin{aligned}
a_x = 0 \Rightarrow \quad & \frac{dv_x}{dt} = 0 \ \frac{dx}{dt} = v_x, \\
a_y = -g \Rightarrow \quad & \frac{dv_y}{dt} = -g \ \frac{dy}{dt} = v_y.
\end{aligned}
$$

We have neglected air resistance, but this can easily be added later if needed. The ODE system can be written as

$$
\begin{aligned}
\frac{dx}{dt} &= v_x, \\
\frac{dv_x}{dt} &= 0, \\
\frac{dy}{dt} &= v_y, \\
\frac{dv_y}{dt} &= -g,
\end{aligned}
$$

and the four unknowns are the ball's position $x(t)$, $y(t)$ and the velocity components $v_x(t)$, $v_y(t)$. To solve the system we need to define initial conditions for all these variables, i.e. we need to know the initial position and velocity of the ball. To solve this system using the `ODESolver` class hierarchy, we first define the right hand side as a Python function: Define the right-hand side:

```
def f(u, t):
    x, vx, y, vy = u
    g = 9.81
    return [vx, 0, vy, -g]
```

We see that the function here returns a list, but this is automatically converted to an array by the solver class' constructor, as mentioned above. The main program is not very different from the examples above, but needs to define an initial condition with four components:

```
# Initial condition, start at the origin:
x = 0; y = 0
# velocity magnitude and angle:
v0 = 5; theta = 80*np.pi/180
vx = v0*np.cos(theta); vy = v0*np.sin(theta)

U0 = [x, vx, y, vy]

solver= ForwardEuler(f)
solver.set_initial_condition(U0)
time_points = np.linspace(0, 1.0, 101)
u, t = solver.solve(time_points)
# u is an array of [x,vx,y,vy] arrays, plot y vs x:
x = u[:,0];  y = u[:,2]

plt.plot(x, y)
plt.show()
```

Notice that since `u` is a two-dimensional array, we use array slicing to extract and plot the individual components. A call like `plt.plot(t,u)` will also work, but this will plot all components in the same window which is not very useful in this case.

3.2 Summary

ODE solvers and Obejct Oriented Programming:

- Many different ODE solvers (Euler, Runge-Kutta, ++)
- Most tasks are common to all solvers:
  - Initialization of solution arrays and right hand side
  - Overall `for`-loop for advancing the solution
- Difference; how the solution is advanced from step k to $k + 1$
- OOP implementation:

  - Collect all common code in a base class
  - Implement the different step (`advance`) functions in subclasses

Systems of ODEs

- All solvers and codes are easily extended to systems of ODEs
- Solution at one time step (u_k) is a vector (one-dimensional array), overall solution is a two-dimensional array
- Slightly more book-keeping, but the bulk of the code is identical as for scalar ODEs