Lecture 2, Solving ODE's

Github: GeneSiriviboon

email: phum_siriviboon@brown.edu

There are a number of question marks "?" throughout the markdown section of this notebook. You are responsible for filling those sections in.

Overview:

- Solving initial value problems, a simple example
- Survey of methods.
- Errors and numerical stability
  - Reading: Newman chapter 4.

Next Lecture: Solving systems of coupled ODE's and Newtons 2nd law.

Simple Example, an RC circuit.

We have a simple RC circuit, as shown below. After a long charge with the switch in position 1 it is flipped to position 2. What is the time dependence of the charge across C?

Using Kirchoff's laws, determine the 1st order ODE we need to solve for $Q(t)$:

$$\frac{dQ}{dt} = -\frac{1}{RC}Q$$

What is the closed form solution of this equation:

$$Q(t) = Q_{init}e^{-\frac{t}{RC}}$$

In this case the problem was easy to solve. However, there are many cases of such problems, formulated as ordinary differential equations, where the solution is not so easy (or even impossible) to find. So how might we approximate the solution to this problem and solve numerically?

An obvious thing to try is to Taylor expand $Q(t)$ for small Δt around $t = 0$

$$Q(\Delta t) = Q(0) + \frac{dQ}{dt}\Delta t + \frac{1}{2}\frac{d^2Q}{dt^2}(\Delta t)^2 + \ldots$$

If Δt is very small, then it is usually a good approximation to ignore terms that involve second and higher powers of Δt

$$Q(\Delta t) \approx Q(0) + \frac{dQ}{dt}\Delta t$$

and we know the functional form of the derivative. It is given by the original ODE! We now have:

$$Q(t + \Delta t) \approx Q(t) - \frac{Q(t)}{RC}\Delta t$$

This is the general strategy for solving ODE's on a computer. By Taylor expanding we have taken a problem defined on a continuous variable t, and transformed it to an approximate problem defined on a discrete variable Δt. This is always necessary since computers only operate on discrete quantities.

Methods for ODE's

To understand our approximate methods, we work with a single ordinary differential equation.

$$\frac{dx}{dt} = g(x, t)$$

where $x(t)$ is the solution we seek and $g(x, t)$ is a well-behaved but otherwise arbitrary function.

If we know $x(t)$ we could determine $g(t + \Delta t)$ from the Taylor series:

$$x(t + \Delta t) = x(t) + \frac{dx}{dt}\Delta t + \frac{d^2x}{dt^2}\frac{(\Delta t)^2}{2!} + \frac{d^3x}{dt^3}\frac{\Delta t^2}{3!} + \ldots$$

Euler's method

$$x(t_{n+1}) = x(t_n) + g(x_n, t_n)\Delta t + O[(\Delta t)^2].$$

where x_n is the the value of our solution at time point t_n and discrete step n.

Improved accuracy: Runge-Kutta methods

RK2:

$$\begin{aligned} k_1 &= \Delta t g(x_n, t_n) \\ k_2 &= \Delta t g(x_n + 1/2k_1, t_n + \Delta t/2) \\ x(t_{n+1}) &= x(t_n) + k_2 \end{aligned}$$

RK4:

$$\begin{aligned} k_1 &= \Delta t g(x_n, t_n) \\ k_2 &= \Delta t g(x_n + 1/2k_1, t_n + \Delta t/2) \\ k_3 &= \Delta t g(x_n + 1/2k_2, t_n + \Delta t/2) \\ k_4 &= \Delta t g(x_n + k_3, t_n + \Delta t) \\ x(t_{n+1}) &= x(t_n) + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \end{aligned}$$

Pseudocode (for all methods):

Initialization

- Assign initial values and constants: $x(0)$, Δt, any constants in $g(x, t)$.
- Set the number of time steps and the initial value of time.

Calculation

- At time step x_i, calculate x_{i+1} and t_{i+1} at step $i + 1$.
- Increase to t_i to the next time step $t_{i+1} = t_i + \Delta t$.
- repeat for $n - 1$ time steps.

Analysis

- store the values for x_i and t_i in a file.
- plot results or use in subsequent calculation.

Exercise

1. Define a function for the exact solution of our simple RC circuit, $Q(t)$. This will be used to test the accuracy of our numerical methods.
2. Define another function to find the solution of our differential equation numerically using an RK2 method.
3. Using the functions in 1 and 2, plot the exact and approximate solutions over-top of each other for the range of time from t = 0 to 10, and using the parameters: $R = 1$, $C = 1$, $Q_0 = 1$, and Δt = 0.1 .
4. Finally, create a function to calculate and then plot the total cumulative error of our approximation as a function of the step size Δt from Δt = 0.001 to 10 over a time from t = 0 to 10. How does this plot change with increasing final time?

You will find some starter code in the cells below

In [27]:

```
# it is good practice to include all import statements in the first code cell at the top of your w
orkbook or script.
import numpy as np
import matplotlib.pyplot as plt

# this is a so called "magic" function that allows for interactive plotting,
# you will see below when we plot the results
%matplotlib notebook
```

In [28]:

```
#  Modify this function to output the exact solution for Q(t) from above
# Q0 and t0 are the initial charge and time

# Note the default argument for t0 = 0. Using this means we only have to pass a value for t0 to th
e function
# if we want something different from 0

def exact(t, params, Q0, t0 =0.0):
    # !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!! exact solution here  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    R,C = params
    Q = Q0*np.exp(-t/R/C)
    return Q
```

In [29]:

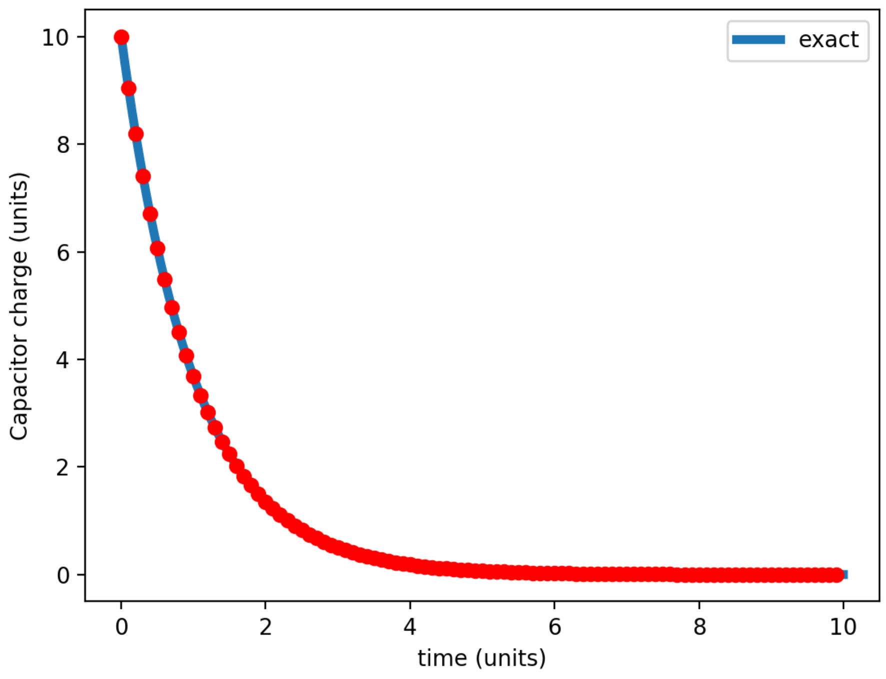
```
# Modify this function to implement an RK2 method for our RC circuit ODE

# note the template here is for one way to do this, and not necessarily the best way.
# I encourage you to try and think of a better and more general way to write this Euler method fun
ction.

#define the first derivative of Q as a function

def DQ(Q, t, params):
    R,C = params
    return -Q/R/C

def approx_rk2(dt, t_final, params, Q0, t0=0.0):

    #initialize some empty lists to store our data and time steps
    Q_list = []
    t_list = []

    # number of time steps, nust be an integer
    n   = int(t_final/dt)

    # the main loop over all time steps
    for ii in range(n):
        if ii == 0.0:
            # set initial conditions at time zero
            Q = Q0
            t = t0
        else :
            # !!!!!!!!!!!!!!!!!!!!!!   RK2 method here  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

            k1 = dt * DQ(Q_list[-1], t, params)
            k2 = dt * DQ(Q_list[-1] + k1/2, t + dt/2, params)

            # update value of Q at each time step and step time by dt
            Q += k2
            t += dt
        # use append to add on to the end of a list
        Q_list.append(Q)
        t_list.append(t)

    # we turn the lists into numpy arrays before returning

    Q_list = np.array(Q_list)
    t_list = np.array(t_list)

    return t_list, Q_list
```

In [30]:

```
# Define our constants
Q0, t0 = 10, 0.0
t_final, dt = 10, 0.1
R, C = 1, 1
params = (R, C)

#create an array with 1000 equally spaced points spanning t0 to _final to pass to our exact expres
sion
t = np.linspace(t0,t_final,1000)

# Calculate the exact solution
Q_exact = Q0 * np.exp(- t/R/C)


#Call our  approximate function, this returns function values and time points
t_approx, Q_approx = approx_rk2(dt, t_final, params, Q0, t0=0.0)
```

In [68]:

```
# Generate plots
fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(t, Q_exact, linestyle = '-', label = "exact", linewidth = '4')
# plot approximate solution (use discrete points with no connecting line)
ax.plot(t_approx, Q_approx, 'ro')

plt.legend()
plt.xlabel("time (units)")
plt.ylabel("Capacitor charge (units)")
```



Out[68]:

```
Text(0, 0.5, 'Capacitor charge (units)')
```

In [62]:

```python
# Write a function to calculate the cumulative error
# hint, try reading the help file on np.sum
def cumulative_error(dt, t_final, params, Q0):
    t_apprx, Q_apprx = approx_rk2(dt, t_final, params, Q0, t0=0.0)
    Q_exact = exact(t_apprx, params, Q0, t0 = 0.0)
    error = np.abs(Q_exact - Q_apprx)
    return np.sum(error)
```

In [63]:

```python
cumulative_error(dt, t_final, params, Q0)
```

Out[63]:

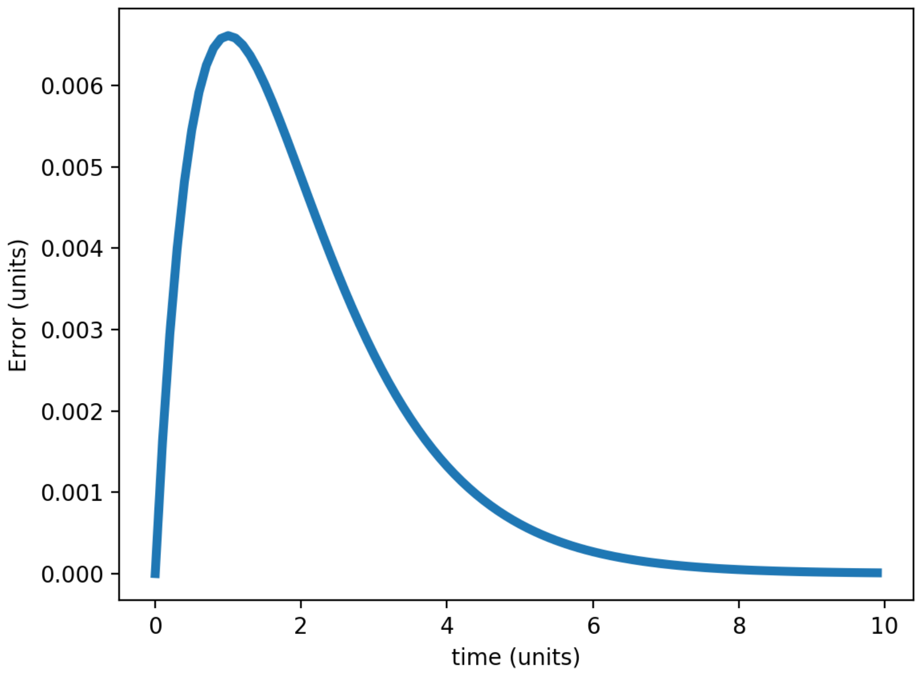
```
0.1797436456964346
```

In [66]:

```python
def partial_error_array(dt, t_final, params, Q0):
    t_apprx, Q_apprx = approx_rk2(dt, t_final, params, Q0, t0=0.0)
    Q_exact = exact(t_apprx, params, Q0, t0 = 0.0)
    error_apprx = np.abs(Q_exact - Q_apprx)
    return t_apprx, error_apprx
```

In [67]:

```python
t_array , error_array = partial_error_array(dt, t_final, params, Q0)
fig2 = plt.figure()
bx = fig2.add_subplot(111)
bx.plot(t_array, error_array, linestyle = '-', label = "error", linewidth = '4')

plt.xlabel("time (units)")
plt.ylabel("Error (units)")
```



Out[67]:

```
Text(0, 0.5, 'Error (units)')
```

In [ ]:

```
```