2018

Semester 2

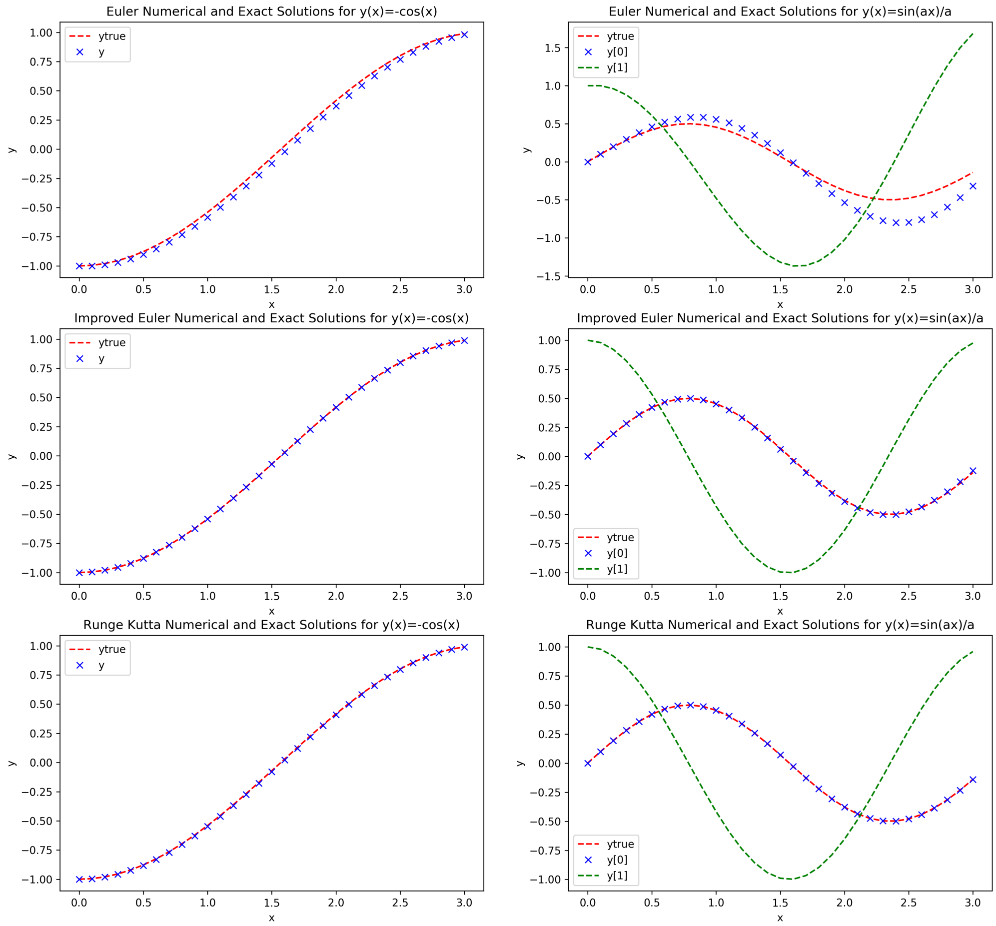
ENGCSI 331 Lab 2 ODEs

Connor McDowall
cmcd398 530913386

August 14, 2018

1 Improved Euler and Runge Kutta Solves

1.1 Hand in 1



```python
def improved_euler_solve(f, x0, y0, x1, h, *args):
    ''' Compute solution to ODE using improved Euler method

        inputs
        ------
        f : callable
            derivative function which, for any input x and ya, yb, yc, ... values, returns a tuple of
            derivative values
        x0 : float
            initial value of independent variable
        y0 : a float, or a numpy array of floats
            array of initial values of solution variables (ya, yb, yc, ...)
        x1 : float
            final value of independent variable
        h : float
            step size
        *args : '*args'optional parameters
            optional parameters to pass to derivative function f()

        returns
        -------
        a list, xs, that gives each of the x values where the solution has been estimated
        a list of numpy arrays, where each array is an estimate of the solution (ya, yb, yc, ...)
        at the corresponding x value
    '''
    n = int(np.ceil((x1-x0)/h))         # number of Improved Euler steps to take
    xs = [x0+h*i for i in range(n+1)]   # x's we will evaluate function at
    ys = [y0]                           # list to store solution; we will append to this

    # iteration
    for k in range(n):
        ys.append( improved_euler_step(f, xs[k], ys[k], h, *args) )

    return xs, ys


def improved_euler_step(f, xk, yk, h, *args):
    ''' Compute a single improved Euler step.

        inputs
        ------
        f : callable
            derivative function
        xk : float
            independent variable at beginning of step
        yk : a float, or a numpy array of floats
            solution at beginning of step
        h : float
            step size
        *args : '*args' optional parameters
            optional parameters to pass to derivative function

        returns
        -------
        a float, or a numpy array of floats, giving solution at end of the step
    '''
    # Compute the improved euler solve to get the new co-ordinate point
    yeuler = yk + h*f(xk,yk,*args)
    # Compute the improved euler step
    return yk + 0.5*(h*f(xk,yk,*args) + h*f(xk+h,yeuler,*args))
```

```python
# Runge Kutta Solve
def runge_kutta_solve(f, x0, y0, x1, h, *args):
    ''' Compute solution to ODE using the clasical 4th order Runge Kutta method

        inputs
        ------
        f : callable
            derivative function which, for any input x and ya, yb, yc, ... values, returns a tuple of
            derivative values
        x0 : float
            initial value of independent variable
        y0 : a float, or a numpy array of floats
            array of initial values of solution variables (ya, yb, yc, ...)
        x1 : float
            final value of independent variable
        h : float
            step size
        *args : '*args' optional parameters
            optional parameters to pass to derivative function f()

        returns
        -------
        a list, xs, that gives each of the x values where the solution has been estimated
        a list of numpy arrays, where each array is an estimate of the solution (ya, yb, yc, ...)
        at the corresponding x value
    '''
    n = int(np.ceil((x1-x0)/h))         # number of Runge Kutta steps to take
    xs = [x0+h*i for i in range(n+1)]   # x's we will evaluate function at
    ys = [y0]                           # list to store solution; we will append to this

    # iteration
    for k in range(n):
        ys.append( runge_kutta_step(f, xs[k], ys[k], h, *args) )

    return xs, ys
```

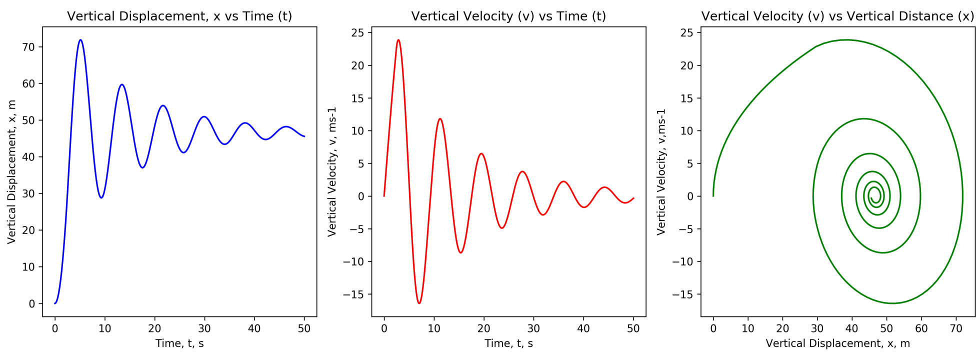
```python
# Runge Kutta
def runge_kutta_step(f, xk, yk, h, *args):
    ''' Compute a single Runge Kutter step.

        inputs
        ------
        f : callable
            derivative function
        xk : float
            independent variable at beginning of step
        yk : a float, or a numpy array of floats
            solution at beginning of step
        h : float
            step size
        *args : '*args' optional parameters
            optional parameters to pass to derivative function

        returns
        -------
        a float, or a numpy array of floats, giving solution at end of the step
    '''
    f0 = f(xk,yk,*args)
    f1 = f(xk + 0.5*h,yk + 0.5*h*f0,*args)
    f2 = f(xk + 0.5*h,yk + 0.5*h*f1,*args)
    f3 = f(xk + 0.5*h,yk + h*f2,*args)
    return yk + ((h/6)*(f0 + 2*f1 + 2*f2 + f3))
```

2 Bungee Jumper Derivative

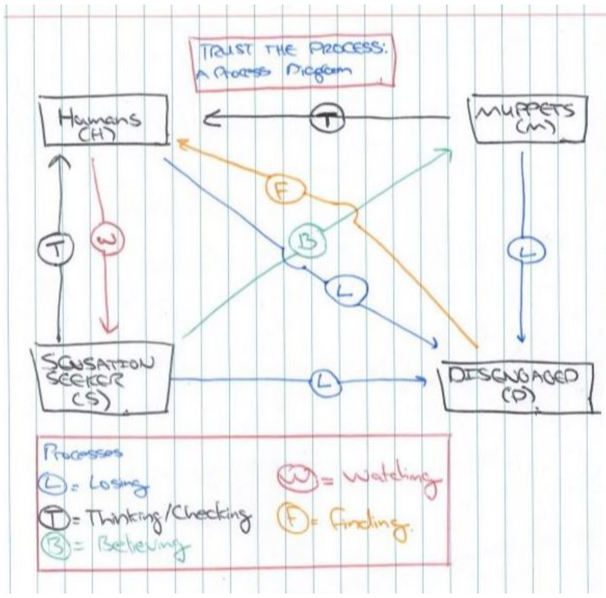
2.1 Hand in 2

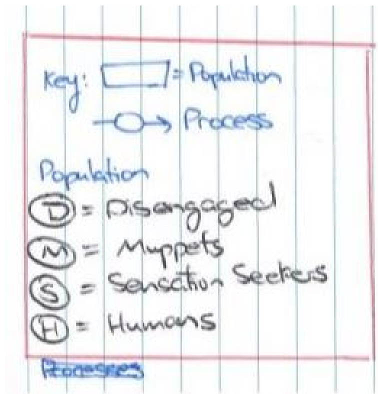



```python
# Task 2
# Higher Order System: Bungy Jumper Model
def bungee_jumper_derivative(t,x,g,cd,m,k,L,gamma):
    if (x[0]<= L):
        result =  np.array([x[1],g-np.sign(x[1])*(cd/m)*x[1]**2] )
        return result
    else:
        result = np.array([x[1],(g-np.sign(x[1])*(cd/m)*x[1]**2 - (k/m)*(x[0]-L) - gamma*x[1]/m)])
        return result
```

3 Fakes News

3.1 Hand in 3: Process Diagram





3.2 Hand in 4: Derivative Relationship

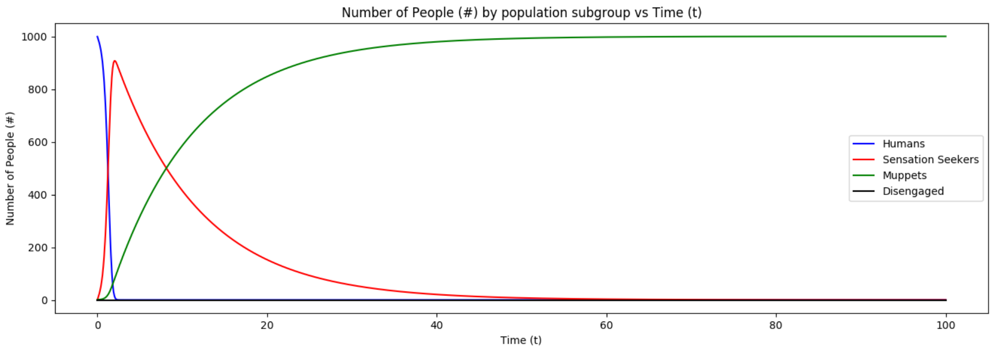
$$\frac{dH}{dt} = (S \times pt) - (H \times M \times pw) + (D \times pf) + (M \times pt) - (H \times pl) \quad (1)$$

$$\frac{dM}{dt} = (-M \times pt) - (M \times pl) + (S \times pb) \quad (2)$$

$$\frac{dS}{dt} = (-S \times pt) + (H \times M \times pw) - (S \times pl) - (S \times pb) \quad (3)$$

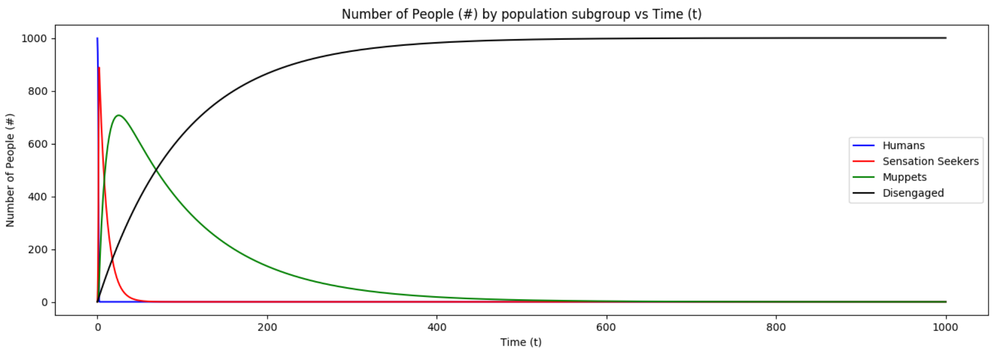
$$\frac{dD}{dt} = (S \times pl) + (H \times pl) + (M \times pl) - (D \times pf) \quad (4)$$

3.3 Hand in 5



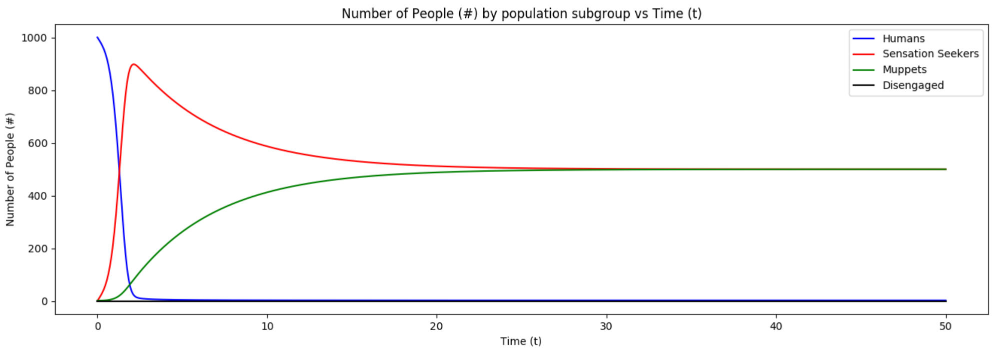
Humans will be rapidly converted to sensation seekers with sensation seekers also converted to muppets at the same time. Once there are no humans left, all sensation seekers will be convert to muppets, enough to occupy street for many years to come.

3.4 Hand in 6



Humans will be converted to sensation seekers, and then muppets. However, losing phones will cause all other parties to be disengaged, with no consuption of fake news. It will take a long time for everyone to lose their phones as the rate is quite small. This seems like a utopia as people will finally talk to eachother.

3.5 Hand in 7



With critical thinking and education, we will have an equal number of sensation seekers to muppets. A state of of equilibrium. This is important as there will be a school of thought to contest ideas. Half the population won't believe the fake news but will stil consume it as the rate of believing fake news will be the same as thinking about it. Critical thinking and education will continue to be very important.

4 Adaptive step-sizes: Orienteering Model

4.1 Hand in 10

```
PS C:\Users\Connor McDowall> cd 'c:\Users\Connor McDowall\Desktop\Lab 2 3
e\extensions\ms-python.python-2018.7.1\pythonFiles\PythonTools\visualstuc
utput' 'c:\Users\Connor McDowall\Desktop\Lab 2 331\ODETesterMain.py'
Solving Instance 0
 Derivative Call Count=480, Tolerance=0.01, a=5, b=2, c=-0.1, d=20, e=3
 Score = 480 with 480 fn calls, maximum error of 0.000800459 & 0 penaltie
Score is 480.0
```

```
def adaptative_runge_kutta_solve(f, x0, y0, x1, h, tol, *args):
    ''' Compute solution to ODE using the clasical 4th order Runge Kutta method with a variable

        inputs
        ------
        f : callable
            derivative function which, for any input x and ya, yb, yc, ... values,
            returns a tuple of derivative values
        x0 : float
            initial value of independent variable
        y0 : a float, or a numpy array of floats
            array of initial values of solution variables (ya, yb, yc, ...)
        x1 : float
            final value of independent variable
        h : float
            step size
        *args : '*args' optional parameters
            optional parameters to pass to derivative function f()

        returns
        -------
        a list, xs, that gives each of the x values where the solution has been estimated
        a list of numpy arrays, where each array is an estimate of the solution (ya, yb, yc, ...)
        at the corresponding x value
    '''
    # Set up intial lists and values for the function
    xs = [x0]
    ys = [y0]
    xk = x0
    yk = y0
    hnew = h
    # iteration until back at the very end point.
    while xk < x1:
        h = min(x1 - xk, hnew)
        xs.append(xk)
        ys.append(yk)
        xk, yk, hnew = adaptive_runge_kutta_step(f, xk, yk, h, tol, *args)
        xk = xk + h
    return xs, ys
```

```python
# Adaptative runge kutta step
def adaptive_runge_kutta_step(f, xk, yk, h, tol, *args):
    ''' Compute a single Runge Kutter step that adapts the step size.

        inputs
        ------
        f : callable
            derivative function
        xk : float
            independent variable at beginning of step
        yk : a float, or a numpy array of floats
            solution at beginning of step
        h : float
            step size
        *args : '*args' optional parameters
            optional parameters to pass to derivative function

        returns
        -------
        a float, or a numpy array of floats, giving solution at end of the step
    '''
    # Calculate all the function values , 3rd and 4th Order Runge Kutta's
    f0 = f(xk,yk,*args)
    f1 = f(xk + 0.5*h,yk + 0.5*h*f0,*args)
    f2_3rd = f(xk + h,yk - h*f0 + 2*h*f1,*args)
    f2_4th = f(xk + 0.5*h,yk + 0.5*h*f1,*args)
    f3 = f(xk + h,yk + h*f2_4th,*args)
    thirdO = yk + ((h/6)*(f0 + 4*f1 + f2_3rd))
    fourthO = yk + ((h/6)*(f0 + 2*f1 + 2*f2_4th + f3))
    # Calculate the observed error
    obs = abs(thirdO - fourthO)
    # Calculate the new h value
    hnew = h*(np.abs(tol/obs)**0.2) if obs > 1e-12 else h
    # 4th Order Return
    return xk, fourthO, hnew
```

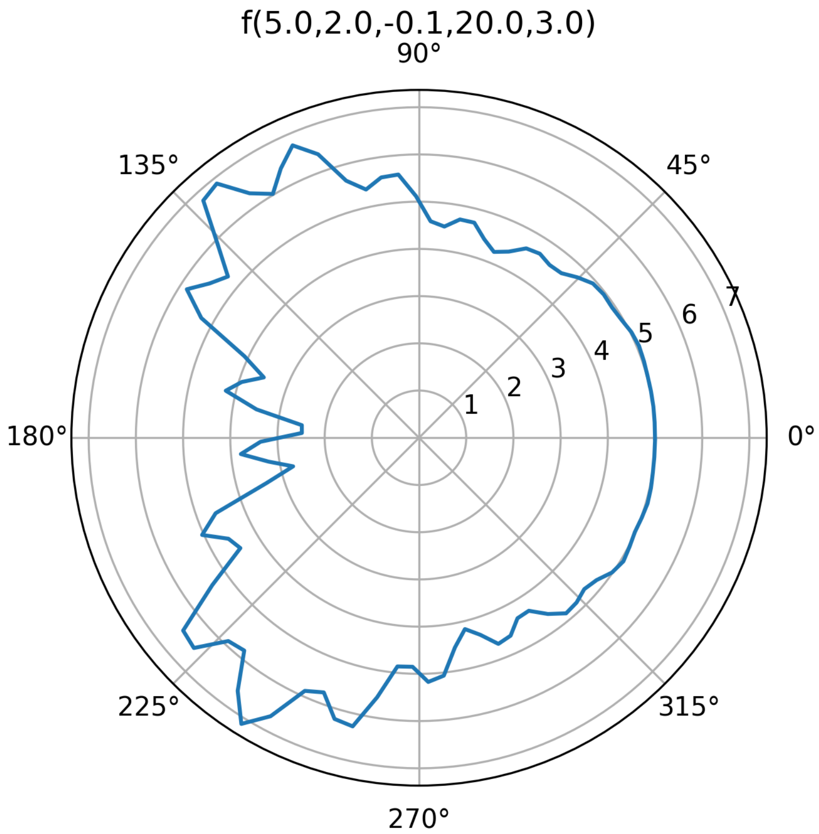
```python
# Use my adaptive runge kutta method
def SolveODE_AdapativeStepping(f, x0, y0, x1, tol, a, b, c, d, e):
    h = 1.05
    x,y = adaptative_runge_kutta_solve(f, x0, y0, x1, h, tol, a, b, c, d, e)

    return (x,y)
```

4.2 Hand in 11

I used error estimatation using embedded runge kutta methods. In the adaptive runge kutta step function, all function evaluations are calculated for third and forth order techniques. An observed difference is calculated between the two function calls. The step sized is scaled by the absolute value of the target difference divided by observed difference, if greater than a machine precision of 1e-12. The stepping function returns the new step size, y and x values.

4.3 Hand in 12




```
Evaluating ODE code for Instance 0
 Derivative Call Count=480, Tolerance=0.01, a=5, b=2, c=-0.1, d=20, e=3
 Score = 480 with 480 fn calls, maximum error of 0.000800459 & 0 penalties.
Evaluating ODE code for Instance 1
 Derivative Call Count=480, Tolerance=0.01, a=4.1, b=2, c=-0.054, d=-20, e=5.2
 Score = 480 with 480 fn calls, maximum error of 0.000921546 & 0 penalties.
Evaluating ODE code for Instance 2
 Derivative Call Count=480, Tolerance=0.001, a=4.1, b=2, c=-0.054, d=-20, e=5.2
 Score = 480 with 480 fn calls, maximum error of 0.000921546 & 0 penalties.
Evaluating ODE code for Instance 3
 Derivative Call Count=480, Tolerance=0.001, a=4.1, b=2, c=-0.2, d=20, e=3
 Score = 480 with 480 fn calls, maximum error of 0.000614508 & 0 penalties.
User = cmcd398: Total Score = 1920
Result cmcd398: 1920 submitted at Tue Aug 14 20:23:42 2018 [ <Response [200]> ]
PS C:\Users\Connor McDowall\Desktop\Lab 2 331>
```

5 Appendix