Sơn Hoài Ân - 20130056

Initial-Value Problems for ODEs

Euler's method

1. Theory This method gives us the way to calculate the integral of a function on an interval A = [a, b] by sum of all values of the function at each points on the interval. The points must be satisfied the distance between each points and insider the interval.
2. Algorithms

```
FUNCTION EulerMethod(f, lower, upper, y0, step ):
    # set value to function processing
    x = lower

    # value of a function at an initial point
    y = y0

    # calculate length of step
    step_len = abs(upper - lower)/step

    # Loop over the number of steps
    FOR i FROM 0 TO n-1 DO:
        # Update y using Euler's formula
        y = y + step_len * f(x, y)

        # Update x
        x = x + step_len

    #return the result of y
RETURN y
```

If you want to return the series of results, you can modify the function above to get them

3. Practice

```
def EulerMethod(func, lower, upper, seed, step):
    x = lower
    y = seed

    # length of step
    step_len = abs(lower - upper)/step

    # Loop and take sum of function values
    for i in range(step):
        # update the result
        y = y + step_len*func(x, y)

        # re-assign x
        x += step_len

    return y
```

4. Trials run

   We will try to evaluate the function $f(x) = x^2 + 5$ on the interval $[\mathbf{2, 4}]$ with 50 steps

```
def EulerMethod(func, seed, lower, upper, step):
    x = lower
    y = seed

    # Loop and take sum of function values
    while x <= upper:
        # update the result
        y = y + step*func(x)

        # re-assign x
        x = x + step

    return y

f = lambda x: x**2 + 5
a = 2
b = 4
```

```
stp = 0.001
int_val = 0.0

res = EulerMethod(f, int_val, a, b, stp)
```

For this code above for the function $f(x) = x^2 + 5$, the program will be calculate the sum of function $f(x)$ from 2 to 4 with the initial value of zero (0) ## Higher - Order Taylor Methods

1. Theory

   The Higher - order Taylor method is a nummerical method which is used to solve ODE by using approximation solutions based on Taylor's series. This method can provide better results and more accuracy when comparing to the lower - order methods as the Euler's method and its variations, etc.

2. Algorithms

   When using this method, we need to use the corresponding higher order of derivatives. In the psuedo below, we will try to see the second order Taylor method.

```
FUNCTION f(x)

FUNCTION f_second_diff(x)
    RETURN DIFF(f(x))

FUNCTION taylor_2nd(func, initial_val, lower, upper, step_len):

    ASSIGN:
        x = lover
        h = step_len
        y = initial_val


    DO:
        y = y + h*f(x) + (h^2/2)*f_second_diff(f(x))
        x += h
    WHILE(x <= upper)

DISPLAY:
    PRINT(RES)
```

$\hspace{10pt}$ In the algorithms above, the function will be calculate the second derivate o

3. Practice

```
import sympy as sp

# declare function
x = sp.symbols('x')
f = x**2 + 5

f_prime = sp.diff(f, x)

# declare variable for calculation
a = 2
b = 4
step_len = 0.001
y = 0 # initial value

# Higher Order methods Function (2nd order)
def Taylor2ndOrder(func, initial_val, lower, upper, distance):
    x0 = lower
    f_prime = sp.diff(func, x)
    y = initial_val

    while (x0 <= upper):
        y = y + distance*func.subs(x, x0) + (distance**2/2)*f_prime.subs(x, x0)
        x0 += distance

    return y

res = Taylor2ndOrder(f, y, a, b, step_len)
print(res)
```

```
28.6876699999985
```

Runge - Kutta Methods

In this section, we will cover Runge-Kutta Methods. If we have a quick look on previous methods, this may be include some of them. The first kind of Runge-Kutta methods is exactly the Euler's method. Now, we try to implement and get to know the three other kinds of the Runge-Kutta methods.

Rungu - Kuta 2

This section we will try to explore two variations of the Runge-Kutta. The first one is the Midpoind method and the Euler enhanced method. Both of them are replied on the traditional Runge-Kutta method.

Midpoint method

In this kinds of Runge-Kutta, we will try to evaluate the integral of a function $y' = f(x_1, x_2, ..., x_n)$.

1. Theory

Assume that the funtion y' = F(x, y) is continous on the interval $[a, b]$, and n is a small distance between the 2-nearest point in the interval - n will be defined by $n = k\frac{|b-a|}{\text{number of sub-intervals}}$; Beside that, we have m_i is the midpoint of i^{th} interval. Then, the fomular to calculate y' on *[a, b]* is given by:

For 1D:

$$\int_a^b y' = \sum_{i=1}^{k} nf(m_i)$$

For 2D:

$$\int_b^a y' = \sum_{i=0}^{k} y_i$$

with

$$y_i = y_{i-1} + nF(x_{i-1} + \frac{n}{2}, y_{i-1} + \frac{n}{2}F(x_{i-1}, y_{i-1}))$$

2. Algorithm

From theory, we have psuedo code for this method. However, $F(x, y)$ must be defined correctly according with the give equation of $f(x_1, x_2, ..., x_n)$

```
FUNCTION F(x, y)

# declare variable
DEFINE:
    lower
    upper
    initial_val
    step_len

# Main processing
# Using while/for loops to iliterate x_i from "lower" to "upper"

FUNCTION MidPoint(func, initial_value, a, b, deltaX):

    ASSIGN:
        x = a
        y = initial_value

    DO:
        y = y + deltaX*F(x + deltaX/2, y + (deltaX/2)*F(x, y))
        x = x + deltaX
    WHILE (x <= b)

    RETURN y

RES = MidPoint(F(x, y), initial_value, lower, upper, step_len)

DISPLAY:
    PRINT(RES)
```

3. Practice (2D)

The implementation in 2D as below

```
def MidPoint(func, initial_val, lower, upper, deltaX):

    x = lower
    y = initial_val

    while(x <= upper):
        y = y + deltaX*func(x + deltaX/2, y + (deltaX/2)*func(x, y))
        x += deltaX
```

```
    return y
```

4. Trial run

Now, here comes an example

```
f = lambda x, y = 1: y + x**2 + 5
a = 2
b = 4
y = 9
step_len = 0.0001

def MidPoint(func, initial_val, lower, upper, deltaX):

    x = lower
    y = initial_val
    z = 0

    while(x <= upper):
        y = y + deltaX*func(x + deltaX/2, y + (deltaX/2)*func(x, y))
        x += deltaX
        z += y*deltaX
    return y

res = MidPoint(f, y, a, b, step_len)
print(res)
```

```
146.33734579928574
```

Euler Enhanced Method

1. Theory

Based on the Euler's method for calculating on an interval, Heun suggests improvement to this by adding first-order ordinary differential equations to the original to get better approximation solutions of function of differantial equations. The enhancement is describe as:

$$y_{i+1} = y_i + \frac{h}{2}[f(x_i, y_i) + f(x_{i+1}, y_i + hf(x_i, y_i)]$$

2. Algorithm

The below psuedo is simulating the Euler enhanced method by Heun.

```
FUNCTION F(x, y)

ASSIGN:
    y = y0 #initial value
    a
    b
    step_len

FUNCTION ModifiedEuler(F, initial_val, lower, upper, deltaX):
    INITIALIZE:
        x = lower
        y = initial_val

    WHILE (x <= b):
        k1 = F(x, y)
        y_pred = y + k1 * deltaX
        k2 = F(x + deltaX, y_pred)

        y = y + (k1 + k2) / 2 * deltaX
        x = x + deltaX

    RETURN y
```

The code below will be return the final integeral value of the differential equation. On the next step, we will implement this method using Python.

3. Implementation

The definition of this method in Python shows as below:

```
def EulerEnhaced(func, initial_val, lower, upper, deltaX):

    # Assign local variable
    y = initial_val
    x = lower

    while (x <= upper):
        y = y + (deltaX /2)* (func(x, y) + func(x + deltaX, y + deltaX*func(x, y)))
        x += deltaX

    return y
```

4. Trial run

We will evalute this method to the previous one.

```
f = lambda x, y = 1: y + x**2 + 5

init_val = 9
a = 2
b = 4
step_len = 0.0001

def EulerEnhaced(func, initial_val, lower, upper, deltaX):

    # Assign local variable
    y = initial_val
    x = lower

    while (x <= upper):
        y = y + (deltaX /2)* (func(x, y) + func(x + deltaX, y + deltaX*func(x, y)))
        x += deltaX

    return y

res = EulerEnhaced(f, init_val, a, b, step_len)
print(res)
```

```
146.33734581525678
```

The Modified Euler Method provides a more accurate approximation of the solution to ODEs compared to the standard Euler method by refining the estimate of the slope. This method can be applied to various problems in science and engineering where such equations arise.

Rungu - Kuta 3 / 4

Based on the 2^{nd}-order of Runge-Kutta methods, we can derive the third and fouth order of this method as:

1. Runge-Kutta 3:

- Method

$$y_0 = \alpha$$

$$y_i = y_{i-1} + \frac{h}{4}\left[F(x_{i-1}, y_{i-1}) + 3F\left(x_{i-1} + \frac{2h}{3}, y_{i-1} + \frac{2h}{3}F\left(x_{i-1} + \frac{h}{3}, y_{i-1} + \frac{h}{3}F(x_{i-1}, y_{i-1})\right)\right)\right]$$

to simplidy this method for easing control if any issues happens, we will define some temporary variables:

$$k_1 = hF(x_{i-1}, y_{i-1})$$

$$k_2 = hF(x_{i-1} + \frac{h}{3}, y_{i-1} + \frac{1}{3}k_1)$$

$$k_3 = hF(x_{i-1} + \frac{2h}{3}, y_{i-1} + \frac{2}{3}k_2)$$

$$y_i = y_{i-1} + \frac{1}{4}(k_1 + 3k_3)$$

with h is a distance between 2 nearest points on the interval **[a, b]** - Algorithm

```
FUNCTION F(x, y)

ASSIGN:
    initial_value
    a, b # the interval
    step_len

DEFINE RungeKutta3(func, initial_val, lower, upper, deltaX):

    # Define local variable
    ASSIGN:
        x = lower
        y = initial_val

    WHILE (x <= upper):

        # Calculate k1, k2, k3
        k1 = deltaX*func(x, y)

        k2 = deltaX*func(x + deltaX / 3, y + k1 / 3)

        k3 = deltaX*func(x + (deltaX * 2) / 3, y + (k2 * 2) / 3)
```

```
        y = y + (k1 + 3 * k3) / 4
        x += deltaX

    RETURN y

RES = RungeKutta3(F(x,y), initial_value, a, b, step_len)

DISPLAY:
    PRINT(RES)
```

- Implement

```
def RungeKutta3(func, initial_val, lower, upper, deltaX):

    x = lower
    y = initial_val

    while (x <= upper):

        k1 = deltaX*func(x, y)

        k2 = deltaX*func(x + deltaX / 3, y + k1 / 3)

        k3 = deltaX*func(x + (deltaX * 2) / 3, y + (k2 * 2) / 3)

        y = y + (k1 + 3 * k3) / 4

        x += deltaX

    return y
```

- Trial run

```
f = lambda x, y = 1: y + x**2 + 5

init_val = 9
a = 2
b = 4
step_len = 0.0001

def RungeKutta3(func, initial_val, lower, upper, deltaX):
```

```
    x = lower
    y = initial_val

    while (x <= upper):

        k1 = deltaX*func(x, y)

        k2 = deltaX*func(x + deltaX / 3, y + k1 / 3)

        k3 = deltaX*func(x + (deltaX * 2) / 3, y + (k2 * 2) / 3)

        y = y + (k1 + 3 * k3) / 4

        x += deltaX

    return y

res = RungeKutta3(f, init_val, a, b, step_len)

print(res)
```

```
146.3373463743783
```

2. Runge-Kutta 4

With the same theory of Runge-Kutta 3, we have a definition and algorithm as: - Definition:

$$y_i = y_{i-1} + h\left(\frac{1}{4}F(x_{i-1}, y_{i-1}) + \frac{3}{4}F\left(x_{i-1} + \frac{2h}{3}, y_{i-1} + \frac{2h}{3}F\left(x_{i-1} + \frac{h}{3}, y_{i-1} + \frac{h}{3}F(x_{i-1}, y_{i-1})\right)\right)\right)$$

For simplify we calculate:

$$k_1 = hF(x_{i-1}, y_{i-1})$$

$$k_2 = hF\left(x_{i-1} + \tfrac{h}{3}, y_{i-1} + \tfrac{h}{3}k_1\right)$$

$$k_3 = hF\left(x_{i-1} + \tfrac{2h}{3}, y_{i-1} + \tfrac{2h}{3}k_2\right)$$

$$k_4 = hF\left(x_{i-1} + h, y_{i-1} + hk_3\right)$$

$$y_i = y_{i-1} + \tfrac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right)$$

- Algorithm:

```
FUNCTION F(x, y)

ASSIGN:
    initial_value
    a, b # the interval
    step_len

FUNCTION RungeKutta4(func, init_val, lower, upper, deltaX):
    ASSIGN:
        x = lower
        y = init_val

    WHILE (x <= b):

        # Calculate number of RK4
        k1 = deltaX * F(x, y)
        k2 = deltaX * F(x + (1/2) * deltaX, y + (1/2) * k1 * deltaX)
        k3 = deltaX * F(x + (1/2) * deltaX, y + (1/2) * k2 * deltaX)
        k4 = deltaX * F(x + deltaX, y + k3 * deltaX)

        y = y + (1/6) * (k1 + 2 * k2 + 2 * k3 + k4)
        x = x + deltaX

    RETURN y

RES = RungeKutta4(F(x,y), initial_value, a, b, step_len)

DISPLAY:
    PRINT(RES)
```

- Implement

```
def RungeKutta4(func, init_val, lower, upper, deltaX):

    x = lower
    y = init_val

    while (x < upper):
        # Calculate number of RK4
        k1 = deltaX * func(x, y)
        k2 = deltaX * func(x + (1/2) * deltaX, y + (1/2) * k1 * deltaX)
        k3 = deltaX * func(x + (1/2) * deltaX, y + (1/2) * k2 * deltaX)
```

```
        k4 = deltaX * func(x + deltaX, y + k3 * deltaX)

        y = y + (1/6) * (k1 + 2 * k2 + 2 * k3 + k4)
        x = x + deltaX

    return y
```

- Trial run

```
f = lambda x, y = 1: y + x**2 + 5

init_val = 9
a = 2
b = 4
step_len = 0.0001

def RungeKutta4(func, init_val, lower, upper, deltaX):

    x = lower
    y = init_val

    while (x < upper):
        # Calculate number of RK4
        k1 = deltaX * func(x, y)
        k2 = deltaX * func(x + (1/2) * deltaX, y + (1/2) * k1 * deltaX)
        k3 = deltaX * func(x + (1/2) * deltaX, y + (1/2) * k2 * deltaX)
        k4 = deltaX * func(x + deltaX, y + k3 * deltaX)

        y = y + (1/6) * (k1 + 2 * k2 + 2 * k3 + k4)
        x = x + deltaX

    return y

res = RungeKutta4(f, init_val, a, b, step_len)
print(res)
```

```
146.32197159416714
```