Project report (4)

Hau Shian Chin (300493343)

Question 1

Introduction

The purpose of this project is to develop an understanding of numerical integration and differentiation through computation mathematics

Numerical differentiation

In [2]:

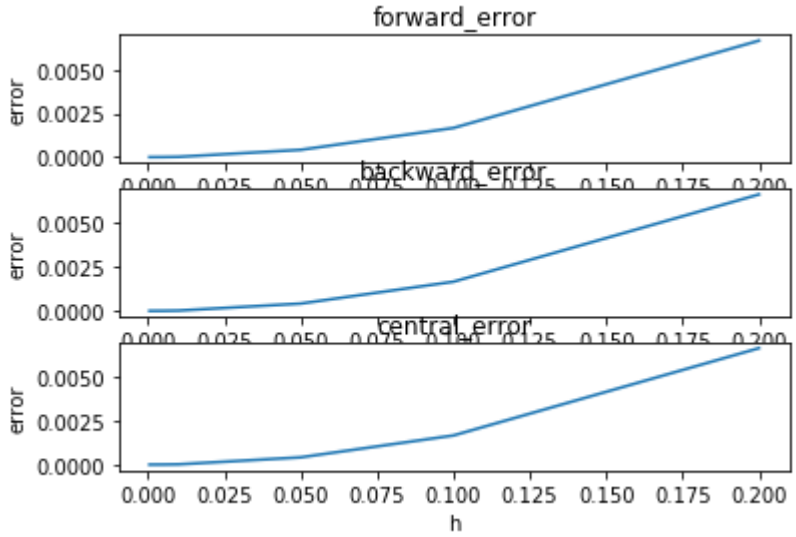
```
import numpy as np
import matplotlib.pyplot as plt

def f(x):
    y = np.sin(x)
    return y

def df(x):
    y = np.cos(x)
    return y

def forward_difference(f, x, h):
    x0 = x
    derivative = (f(x0 + h) - f(x0))/h
    return derivative

def backward_difference(f, x, h):
    x0 = x
    derivative = (f(x0+h) - f(x0))/h
    return derivative

def central_difference(f, x, h):
    x0 = x
    derivative = (f(x0+h) - f(x0-h))/(2.0*h)
    return derivative

def print_differences(difference, f, x, values_h, direction):

    if(direction == "forward"):
        for i in range(len(values_h)):
            print("at h =", values_h[i], "forward difference is",difference(f,x,values_h[i
]), "error = ", abs(df(x) - difference(f,x,values_h[i])))
            forward_error[i] = abs(df(x) - difference(f,x,values_h[i]))
    elif(direction == "backward"):
        for i in range(len(values_h)):
            print("at h =", values_h[i], "backward difference is",difference(f,x,values_h[
i]), "error = ", abs(df(x) - difference(f,x,values_h[i])))
            backward_error[i] = abs(df(x) - difference(f,x,values_h[i]))
    else:
        for i in range(len(values_h)):
            print("at h =", values_h[i], "central difference is",difference(f,x,values_h[i
]), "error = ", abs(df(x) - difference(f,x,values_h[i])))
            central_error[i] = abs(df(x) - difference(f,x,values_h[i]))

x = np.pi
h = [0.2, 0.1, 0.05, 0.01, 0.005, 0.001, 0.0005]
forward_error = h.copy()
backward_error = h.copy()
central_error = h.copy()
print("\nforward differences")
print_differences(forward_difference, f, x, h, "forward")
print("\nbackward differences")
print_differences(backward_difference, f, x, h, "backward")
print("\ncentral differences")
print_differences(central_difference, f, x, h, "central")
```

```
plt.subplot(3,1,1)
plt.title("forward_error")
plt.plot(h, forward_error)
plt.xlabel("h")
plt.ylabel("error")

plt.subplot(3,1,2)
plt.title("backward_error")
plt.plot(h, backward_error)
plt.xlabel("h")
plt.ylabel("error")

plt.subplot(3,1,3)
plt.title("central_error")
plt.plot(h, central_error)
plt.xlabel("h")
plt.ylabel("error")
```

```
forward differences
at h = 0.2 forward difference is -0.9933466539753069 error =  0.0066533460246
93141
at h = 0.1 forward difference is -0.9983341664682823 error =  0.0016658335317
176753
at h = 0.05 forward difference is -0.9995833854135631 error =  0.000416614586
4369364
at h = 0.01 forward difference is -0.9999833334166452 error =  1.666658335475
1907e-05
at h = 0.005 forward difference is -0.9999958333385203 error =  4.16666147973
1992e-06
at h = 0.001 forward difference is -0.9999998333332315 error =  1.66666768497
41435e-07
at h = 0.0005 forward difference is -0.999999958333668 error =  4.16663320512
71127e-08

backward differences
at h = 0.2 backward difference is -0.9933466539753069 error =  0.006653346024
693141
at h = 0.1 backward difference is -0.9983341664682823 error =  0.001665833531
7176753
at h = 0.05 backward difference is -0.9995833854135631 error =  0.00041661458
64369364
at h = 0.01 backward difference is -0.9999833334166452 error =  1.66665833547
51907e-05
at h = 0.005 backward difference is -0.9999958333385203 error =  4.1666614797
31992e-06
at h = 0.001 backward difference is -0.9999998333332315 error =  1.6666676849
741435e-07
at h = 0.0005 backward difference is -0.999999958333668 error =  4.1666332051
271127e-08

central differences
at h = 0.2 central difference is -0.9933466539753069 error =  0.0066533460246
93141
at h = 0.1 central difference is -0.9983341664682823 error =  0.0016658335317
176753
at h = 0.05 central difference is -0.999583385413563 error =  0.0004166145864
370474
at h = 0.01 central difference is -0.9999833334166451 error =  1.666658335486
293e-05
at h = 0.005 central difference is -0.9999958333385205 error =  4.16666147950
9948e-06
at h = 0.001 central difference is -0.9999998333332315 error =  1.66666768497
41435e-07
at h = 0.0005 central difference is -0.9999999583336677 error =  4.1666332273
31573e-08
```

Out[2]:

```
Text(0, 0.5, 'error')
```


forward_error
backward_error
central_error
error
0.0050
0.0025
0.0000
0.000
0.025
0.050
0.075
0.100
0.125
0.150
0.175
0.200
h

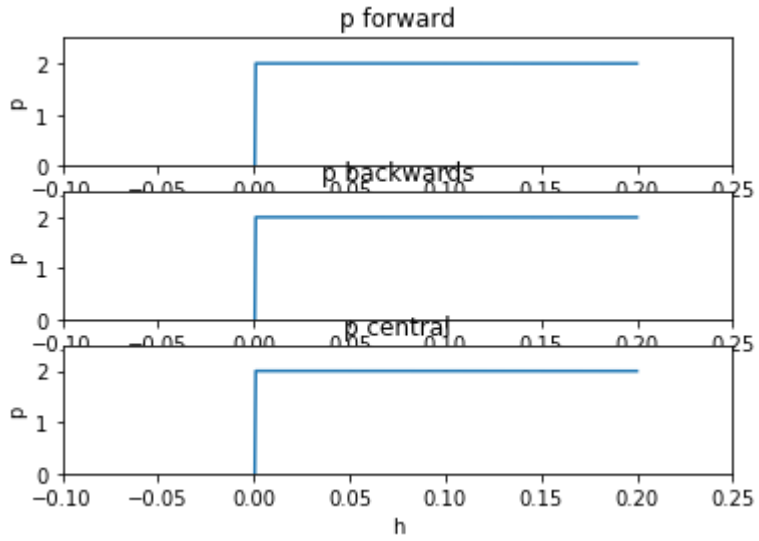
In [3]:

```
def approx(errors, h, p):

    for i in range(len(h) - 1):
        p[i] = np.log(errors[i]/errors[i+1])/np.log(h[i]/h[i+1])
    #print(p)

print("p forward")
p_forward = h.copy()
approx(forward_error, h, p_forward)

print("p backward")
p_backward = h.copy()
approx(backward_error, h, p_backward)

print("p central")
p_central = h.copy()
approx(central_error, h, p_central)

plt.subplot(3,1,1)
plt.title("p forward")
plt.plot(h, p_forward)
plt.xlabel("h")
plt.ylabel("p")
plt.xlim([-0.1, 0.25])
plt.ylim([0, 2.5])

plt.subplot(3,1,2)
plt.title("p backwards")
plt.plot(h, p_backward)
plt.xlabel("h")
plt.ylabel("p")
plt.xlim([-0.1, 0.25])
plt.ylim([0, 2.5])

plt.subplot(3,1,3)
plt.title("p central")
plt.plot(h, p_central)
plt.xlabel("h")
plt.ylabel("p")
plt.xlim([-0.1, 0.25])
plt.ylim([0, 2.5])
```

```
p forward
p backward
p central
```

Out[3]:

```
(0, 2.5)
```



Observations

I noticed the error between forward, backwards and central are marginal. There little difference in error between all the methods besides central error being slightly better than forwards and bacwards difference. However, the varying difference in h values will determine how accurate the approximation is. smaller the h value, smaller the error, vice versa.

Discussion

I believe the reason why there's no difference in error of the forward and backward difference (given that h is the same) is because they're measuring the increment at the same step. As a consequence, the gradient will be the same. So when the differences is removed from an actual derivative it will result in the same. However, since central considers both forward and backward difference it is safe to say that the error difference between the gradient of central difference and the actual derivative is even smaller than forward/backward difference.

The h value determines the actual increment of the steps each difference take, decreasing it will give a more accurate estimate of the derivative at the given point.

Conclusion

in conclusion, h value of 0.0005 and central difference will yield the best estimate in terms of obtaining a gradient at a given point without using a derivative as it has the smallest given error. But forward/backward difference are still pretty accurate on its own.

Numerical integration

In [4]:

```
def midpointquad(func, a, b, N):
    #a = lower limit
    #b = upper limit of integration
    #N = number of points

    x = np.linspace(a,b,N)
    #y = x.copy()
    output = 0.0
    for i in range(N-1):
        output += (x[i+1]-x[i]) * func((x[i] + x[i+1])/2.0)
    return output

def f(x):
    y = 2.0*x
    return y

def runge_func(x):
    y = 1.0/(1.0 + x**2.0)
    return y

print("mid point function of ∫ 2x|0 to 1 dx == ", midpointquad(f,0,1,2))
print("mid point function of ∫ 1/(1+x^2)|-5 to 5 dx", midpointquad(runge_func,-5,5,50))
```

```
mid point function of ∫ 2x|0 to 1 dx ==  1.0
mid point function of ∫ 1/(1+x^2)|-5 to 5 dx 2.7468528501889633
```

the order of accuracy of the mid point method is O(h^3)

In [5]:

```
Ns =  [11, 101,1001, 10001]
hs = [1.0, 0.1, 0.01, 0.001]
arctan52 = 2.0*np.arctan(5)
results = Ns.copy()
error = Ns.copy()

for i in range(len(Ns)):
    #print("∫(1/(1+x^2))|-5 to 5 =", midpointquad(runge_function, -5, 5, Ns[i]))
    results[i] = midpointquad(runge_func, -5, 5, Ns[i])
    error[i] = abs(arctan52 - midpointquad(runge_func, -5, 5, Ns[i]))
    #print("error for N = ", Ns[i], "is equal to ", error[i])

print("N\t h\t mid point results \t error")
print("-----------------------------------------------------------------")
for i in range(len(Ns)):
    print(Ns[i],"\t", hs[i], "\t",results[i], "\t",error[i])

# since error difference is calculated by 2arctan
```

```
N        h       mid point results       error
-----------------------------------------------------------------
11       1.0     2.736307727635371       0.010493806254660676
101      0.1     2.7468138597748015      1.2325884769737172e-05
1001     0.01    2.7468016571640375      1.2327400567002655e-07
10001    0.001   2.74680153512277        1.2327383558385918e-09
```

i believe they scale up by 3 orders of magnitude, judging from the difference made by the incrementation of N values. thus the order of accuracy of the mid point method is O(h^3)

In [6]:

```
#2.

def func(x):
    y = 1.0
    return y

def func1(x):
    y = 2.0*x
    return y

def func2(x):
    y = 3.0*x**2.0
    return y

def func3(x):
    y = 4.0*x**3.0
    return y

interval = 2
functions = [func, func1, func2, func3]
solutions = [1.0, 1.0, 1.0, 1.0]
results2 = solutions.copy()

for i in range(len(functions)):
    results2[i] = midpointquad(functions[i], 0, 1, Ns[i])
    error[i] = abs(solutions[i] - midpointquad(functions[i], 0, 1, Ns[i]))

print("function\t\t results\t\t errors")
print("-----------------------------------------------------------------")
print("∫(1)|0 to 1 = \t    \t", results2[0], "\t         \t",error[0])
print("∫(2x)|0 to 1 = \t   \t", results2[1], "\t         \t",error[1])
print("∫(3x^2)|0 to 1 = \t", results2[2], "\t",error[2])
print("∫(4x^3)|0 to 1 = \t", results2[3], "\t",error[3])
```

```
function                results                 errors
-----------------------------------------------------------------
∫(1)|0 to 1 =           1.0                     0.0
∫(2x)|0 to 1 =          1.0                     0.0
∫(3x^2)|0 to 1 =        0.9999997500000014      2.4999999859165456e-07
∫(4x^3)|0 to 1 =        0.9999999949999971      5.000002856192509e-09
```

order of exactness analysis

The order of exactness is 1 as there wasn't any error when we were given 2x. However when we are given 3x^2 we get given a visble error thus the order of exactness is just 1.

In [7]:

```
##trap part
def trapezoid(f, a, b, N):
    x = np.linspace(a, b, N+1)
    y = f(x)
    h = (b - a)/N

    sum = 0.0
    for i in range(1,N):
        sum += 2.0*y[i]
    sum = 0.5*h*(f(a) + sum + f(b))
    return sum

def f(x):
    y = 2.0*x
    return y

def runge_function(x):
    y = 1.0/(1.0 + x**2.0)
    return y


print("∫(2x)|0 to 1 = ", trapezoid(f, 0, 1, 2))
print("∫(1/(1+x^2))|-5 to 5 = ", trapezoid(runge_function, -5, 5, 500))
print("\n")
##############################################################################################
##
Ns =  [11, 101,1001, 10001]
hs = [1.0, 0.1, 0.01, 0.001]
arctan52 = 2.0*np.arctan(5)
results = Ns.copy()
error = Ns.copy()

for i in range(len(Ns)):
    #print("∫(1/(1+x^2))|-5 to 5 =", midpointquad(runge_function, -5, 5, Ns[i]))
    results[i] = trapezoid(runge_function, -5, 5, Ns[i])
    error[i] = abs(arctan52 - trapezoid(runge_function, -5, 5, Ns[i]))
    #print("error for N = ", Ns[i], "is equal to ", error[i])

print("N\t h\t trapezoidal results \t error")
print("-------------------------------------------------------------")
for i in range(len(Ns)):
    print(Ns[i],"\t", hs[i], "\t",results[i], "\t",error[i])

# since error difference is calculated by 2arctan
##############################################################################################

interval = 2
functions = [func1, func2, func3]
solutions = [1.0, 1.0, 1.0]
results2 = solutions.copy()

for i in range(len(functions)):
    results2[i] = trapezoid(functions[i], 0, 1, Ns[i])
    error[i] = abs(solutions[i] - trapezoid(functions[i], 0, 1, Ns[i]))
```

```
print("\n")
print("function\t\t results\t\t errors")
print("-----------------------------------------------------------------")
#print("∫(1)|0 to 1 = \t    \t", results2[0], "\t         \t",error[0]) doesn't work for c
onstants
print("∫(2x)|0 to 1 = \t   \t", results2[0], "\t\t        ",error[0])
print("∫(3x^2)|0 to 1 = \t", results2[1], "\t",error[1])
print("∫(4x^3)|0 to 1 = \t", results2[2], "\t",error[2])
```

```
∫(2x)|0 to 1 =  1.0
∫(1/(1+x^2))|-5 to 5 =  2.7468005476995394


N        h       trapezoidal results     error
-----------------------------------------------------------------
11       1.0     2.7385222270696588      0.008279306820373034
101      0.1     2.7467773665372195      2.416735281229876e-05
1001     0.01    2.746801287834239       2.460557926298179e-07
10001    0.001   2.7468015314250143      2.4650175234341987e-09


function                results                 errors
-----------------------------------------------------------------
∫(2x)|0 to 1 =          1.0                     0.0
∫(3x^2)|0 to 1 =        1.0000490148024705      4.9014802470548346e-05
∫(4x^3)|0 to 1 =        1.0000009980029956      9.980029955780623e-07
```

Comparison between trapezoidal rule and mid point rule

As we can see, the order of exactness is the same between the midpoint method and the trapezoidal method in a linear polynomial. however, the order of accuracy is more accurate for the trapezoid method since it gives us a smaller error overall. This intuitively makes more sense as trapezoidal method should give us a more accurate output. In addition, greater powers of polynomial are more accurate for trapezoidal method as it mean to cover more curvature of the polynomial.

It also seems like trapezoidal rule doesn't work for the integral of 1, this make sense as there is no curivutre for the trapezoidal to take advantage of hence throwing an error. For some reason, there seems to be a greater computational error for larger orders of polynomials inside trapezoidal rule which shouldn't make sense. This is because trapezoidal rule take advantage of the curves.

In [8]:

```
from numpy import polynomial as py

def gauss(f,a,b,N):
    [x,w] = py.legendre.leggauss(N);

    c1 = (b-a) * 0.5
    c2 = (b+a) * 0.5

    sum = 0.0
    for i in range(N):
        sum += w[i]*f(c1*x[i]+c2)
    sum = c1*sum
    return sum

def f1(x):
    y = 3.0*x**2.0
    return y

def f2(x):
    y = 4.0*x**3.0
    return y

def f3(x):
    y = 5.0*x**4.0
    return y

def f4(x):
    y = 6.0*x**5.0
    return y

def f5(x):
    y = 7.0*x**6.0
    return y

def test_function(x):
    y = 2.0*x
    return y

print("∫(2x)|0 to 1 =", gauss(test_function, 0,1,2))
funcs = [f1,f2,f3,f4,f5]
funcst = ["3x^2" ,"4x^3","5x^4", "6x^5", "7x^6"]
solutions = [1.0,1.0,1.0,1.0,1.0]
N2 = solutions.copy()
N3 = solutions.copy()



for i in range(len(funcs)):
    N2[i] = abs(solutions[i] - gauss(funcs[i],0,1,2))
    N3[i] = abs(solutions[i] - gauss(funcs[i],0,1,3))

#print(N2)
#print(N3)

print("functions    Error for N = 2\t\t Error for N = 3")
```

```
print("-----------------------------------------------------------------")
print(funcst[0], "\t    ", N2[0], "\t               ", N3[0])
print(funcst[1], "\t    ", N2[1], "\t", N3[1])
print(funcst[2], "\t    ", N2[2], "\t", N3[2])
print(funcst[3], "\t    ", N2[3], "\t", N3[3])
print(funcst[4], "\t    ", N2[4], "\t", N3[4])
print("Degrees\t     3\t\t\t\t 5")
print()
##########################################################################################
##
Ns =  [3,4,5,6,7,11, 15]
arctan52 = 2.0*np.arctan(5)
results = Ns.copy()
error = Ns.copy()

for i in range(len(Ns)):
    #print("∫(1/(1+x^2))|-5 to 5 =", midpointquad(runge_function, -5, 5, Ns[i]))
    results[i] = gauss(runge_function, -5, 5, Ns[i])
    error[i] = abs(arctan52 - gauss(runge_function, -5, 5, Ns[i]))
    #print("error for N = ", Ns[i], "is equal to ", error[i])

print("N\t Gauss-Legendre results  Error")
print("-----------------------------------------------------------------")
for i in range(len(Ns)):
    print(Ns[i], "\t",results[i], "\t",error[i])
```

```
∫(2x)|0 to 1 = 1.0
functions    Error for N = 2             Error for N = 3
-----------------------------------------------------------------
3x^2         0.0                         2.220446049250313e-16
4x^3         1.1102230246251565e-16      2.220446049250313e-16
5x^4         0.02777777777777779         2.220446049250313e-16
6x^5         0.08333333333333348         2.220446049250313e-16
7x^6         0.15740740740740744         0.0024999999999997247
Degrees      3                           5

N        Gauss-Legendre results  Error
-----------------------------------------------------------------
3        4.791666666666666       2.0448651327766343
4        1.8546365914786969      0.8921649424113349
5        3.5347396019544752      0.7879380680644434
6        2.3085027921188503      0.43829874177118144
7        3.0806104010709605      0.3338088671809287
11       2.812290560886776       0.0654890269967443
15       2.760067369009          0.013265835118968283
```

Observations and Analysis

It seems that the Gauss-Legendre method is really accurate depending on the number of intervals. ideally Errors should be around the order of magnitude of 10 ^-16 which has been achieved with low levels of N, given that the interval is small (0 to 1). This is because numbers are infinite and computer memory is finite which means that the smallest possible number would be in order magnitude of -16. The order of exactness = 2N - 1. This means that the exactness will increase linearly to the number of interval.

Conclusion

In concolusion, Numerical integration has their limits. However, it is so close to it's ideal mathematical expression that it can be overlooked.

In [ ]: