Basic Info

Submission Info

Ni Jiasheng(Alex)

jn2294

2022-2-11

The complete code is attached at the end.

Speed Test Info

The speed tests are performed under i7-9750H CPU at 2.60Ghz without GPU acceleration.

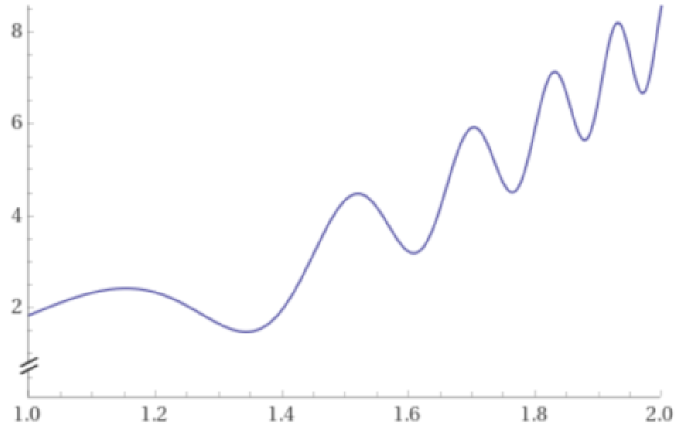
The speed are displayed with second unit.

Problem Solution

Problem Description

Problems 2.1-2.5: Given the following function (that I'm happy to have made it up):

$$f(x) = \sin(x^5) + x^3$$



Problem 2.1-2.2

Problems 2.1-2.2 Write a program to compute $f'(1.5)$ and $f'(1.7)$ with step size $h = 10^{-2}$ and $h = 10^{-3}$ for each. You should expect 4 numbers.

Problem 2.1

Problem 2.1 uses the central difference method.

Algorithm

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + O(h^2)$$

- Directly apply the algorithm

Code

```
    # 2.1 Central Difference Method
    def cdm(self,h,x0,speed_test=False,dis_error =
False):
```

```
        """
        :param h: the step size/ solution
        :param x0: center point
        :param speed_test: whether to take speed
test, switch off to exclude the effect of print
statement
        :param dis_error: switch off for speed test
to exclude the effect of print estimation error
        :return:
        """

        res = (self.f(x0+h) - self.f(x0-h))/(2*h)

        if not speed_test:
            print("f'({0}) under h={1} is
{2}".format(x0,h,res))

            if dis_error:
                correct = self.f_diff(x0)
                print("The correct result is {3},
the error is {4}"
                      .format(x0, h,
res,correct,np.abs(correct - res)))
```

Result

```
Testing Central Difference Method !
f'(1.5) under h=0.01 is 13.112629767913164
The correct result is 13.263017466669073, the error
is 0.15038769875590852
f'(1.5) under h=0.001 is 13.261503320618484
The correct result is 13.263017466669073, the error
is 0.0015141460505887494
f'(1.7) under h=0.01 is 5.982295901277723
The correct result is 6.1073885872023475, the error
is 0.1250926859246242
f'(1.7) under h=0.001 is 6.106085359878222
The correct result is 6.1073885872023475, the error
is 0.0013032273241257997
```

Problem 2.2

Problem 2.2 uses the forward difference method.

Algorithm

$$f'(x) = \frac{f(x+h) - f(x)}{h} + O(h)$$

- Directly apply the algorithm

Code

```
    # 2.2 Forward Difference Method
    def fdm(self,h,x0,speed_test=False,dis_error =
False):
        """
        :param h: the step size/ resolution
```

```
        :param x0: center point
        :param speed_test: whether to take speed
test, switch off to exclude the effect of print
statement
        :param dis_error: switch off for speed test
to exclude the effect of print estimation error
        :return:
        """

        res = (self.f(x0 + h) - self.f(x0)) /  h

        if not speed_test:
            print("f'({0}) under h={1} is
{2}".format(x0, h, res))

            if dis_error:
                correct = self.f_diff(x0)
                print("The correct result is {3},
the error is {4}"
                      .format(x0, h,
res,correct,np.abs(correct - res)))
```

Result

```
Testing Forward Difference Method !
f'(1.5) under h=0.01 is 10.16137178294576
The correct result is 13.26301746669073, the error
is 3.101645683723312
f'(1.5) under h=0.001 is 12.96512510091663
The correct result is 13.26301746669073, the error
is 0.2978923657524426
f'(1.7) under h=0.01 is -2.5740464745511282
The correct result is 6.1073885872023475, the error
is 8.681435061753476
f'(1.7) under h=0.001 is 5.237970831388772
The correct result is 6.1073885872023475, the error
is 0.8694177558135756
```

Speed Test 1

We find that the **efficiency** of cdm and fdm are **almost the same** for estimating differentiation, but the **accuracy** of central difference method is higher than that of forward difference method.

```
 # Speed test for computing differentiation
    def speed_test1(self):
        print("Speed testing central difference
method!")
        times = []

        for i in range(200):
            start = time.time()
            self.cdm(10e-3, 1.5,speed_test=True)
            self.cdm(10e-4, 1.5,speed_test=True)
            self.cdm(10e-3, 1.7,speed_test=True)
            self.cdm(10e-4, 1.7,speed_test=True)
            times.append(time.time()-start)
```

```
            print("Central Difference Method takes
{0} on average".format(np.mean(times)))

            print("Speed testing forward difference
method!")
            times = []
            for i in range(200):
                start = time.time()
                self.fdm(10e-3, 1.5,speed_test=True)
                self.fdm(10e-4, 1.5,speed_test=True)
                self.fdm(10e-3, 1.7,speed_test=True)
                self.fdm(10e-4, 1.7,speed_test=True)
                times.append(time.time() - start)

                print("Forward Difference Method
takes {0} on average".format(np.mean(times)))
```

```
Speed testing central difference method!
Central Difference Method takes 0.0005885851383209228 on average
Speed testing forward difference method!
Forward Difference Method takes 0.0005882632732391357 on average
```

Problem 2.3-2.5

Problems 2.3-2.5 Write a program to compute

$$\int_{1.5}^{2.0} f'(x)\,dx$$

Problem 2.3

Problem 2.3 use the **Midpoint Rule** with $n = 10^1, 10^2, 10^3$ intervals.

Algorithm

$$\int_a^b f(x)\, dx \approx \Delta x \left[f\left(x_1^*\right) + f\left(x_2^*\right) + \cdots + f\left(x_n^*\right)\right]$$

- Compute Δx
- Record the separation points in **sep_points[n+1]**
- Construct the midpoint array **midpoint** by adding up two n-item slices of sep_point (with 1 shift)
- Compute the result

Code

```
# 2.3 Midpoint Rule
def midpoint(self,n, speed_test=False,dis_error =
False):
    """
        :param n: Number of intervals
        :param speed_test: whether to take speed
test, switch off to exclude the effect of print
statement
        :param dis_error: switch off for speed test
to exclude the effect of print estimation error
        :return:
        """

    delta_x = (self.end-self.start)/n
    sep_points =
np.linspace(self.start,self.end,n+1)
    v1 = np.array(sep_points[:-1])
    v2 = np.array(sep_points[1:])
    mid_points = (v1+v2)/2
```

```
    result = delta_x*sum(map(lambda
x:self.f_diff(x),mid_points))

    if not speed_test:
        print("The integration of the f'(x) under n=
{0} on [{1},{2}] is {3}".format(n,1.5,2.0,result))

        if dis_error:
            correct = 4.21009627762213
            print("The correct result is {0}, the
error is {1}"
                  .format( correct, np.abs(correct -
result)))
```

Result

```
Testing Midpoint Rule !
The integration of the f'(x) under n=10 on [1.5,2.0]
is 4.6235541856854
The correct result is 4.21009627762213, the error is
0.4134579080632701
The integration of the f'(x) under n=100 on
[1.5,2.0] is 4.21301643365242
The correct result is 4.21009627762213, the error is
0.0029201560302904994
The integration of the f'(x) under n=1000 on
[1.5,2.0] is 4.210125350515661
The correct result is 4.21009627762213, the error is
2.9072893530823762e-05
```

Problem 2.4

Problem 2.4 use the **Simpson's 1/3 Rule** with $n = 10^1, 10^2, 10^3$ intervals.

Algorithm

$$\int_a^b f(x)\,dx \approx \frac{\Delta x}{3}\left[f(x_0) + 4f(x_1) + 2f(x_2) + \cdots + 2f(x_{n-2}) + 4f(x_{n-1}) + f(x_n)\right]$$

- Compute Δx
- Constuct separation points in **points[n+1]**
- Construct coefficient as a vector
- Construct $f(x)$s mapped from each separation points into a vector
- Multiply the two vector to obtain the result

Code

```
# 2.4 Simpson's 1/3 Rule
def simpson(self,n,speed_test=False,dis_error = False):
    """
        :param n: Number of intervals
        :param speed_test: whether to take speed test, switch off to exclude the effect of print statement
        :param dis_error: switch off for speed test to exclude the effect of print estimation error
        :return:
        """
    delta_x = (self.end-self.start)/n
    points = np.linspace(self.start,self.end,n+1)
    if n > 2:
```

```
        coeff_vec = np.array([1]+[4,2]*((n-2)//2)+
[4]+[1],dtype=float)
    else:
        coeff_vec = np.array([1,4,1],dtype=float)
        function_value =
np.array([self.f_diff(point) for point in
points],dtype=float)
        result =
delta_x*coeff_vec.dot(function_value)/3

    if not speed_test:
          print("The integration of f'(x) under n=
{0} on [{1},{2}] is
{3}".format(n,self.start,self.end,result))

          if dis_error:
               correct = 4.21009627762213
               print("The correct result is {0}, the
error is {1}"
                          .format( correct,
np.abs(correct - result)))
```

Result

```
Testing Simpson's 1/3 Rule !
The integration of f'(x) under n=10 on [1.5,2.0] is
-1.2675480683480485
The correct result is 4.21009627762213, the error is
5.4776443459701785
The integration of f'(x) under n=100 on [1.5,2.0] is
4.210156146034342
The correct result is 4.21009627762213, the error is
5.986841221261585e-05
The integration of f'(x) under n=1000 on [1.5,2.0]
is 4.210096283550819
The correct result is 4.21009627762213, the error is
5.928689539302923e-09
```

Problem 2.5

Problem 2.5 use the **Gaussian quadrature** with $n = 3, 4, 5$ points.
Note: Please use x_i and c_i provided in the Lecture Notes for this problem.

Algorithm

$$\int_a^b f(x)dx \simeq \frac{b-a}{2}\sum_{i=1}^{n} w_i f(\frac{b-a}{2}x_i + \frac{b+a}{2})$$

- Do a linear transformation with change of bases in the integral
- Apply the algorithm directly above with w_i and x_i in the table lookup.

Code

```
# 2.5 Gaussian Quadrature
def gaussian(self,n,speed_test=False,dis_error =
False):
    """
        :param n: Number of points
        :param speed_test: whether to take speed
test, switch off to exclude the effect of print
statement
        :param dis_error: switch off for speed test
to exclude the effect of print estimation error
        :return:
        """
    # x = 1.75+0.25xd, where xd ~ [-1,1]
    a = (self.end+self.start)/2  # 1.75
    b = (self.end-self.start)/2  # 0.25
    if n == 3:
        I = float((5 / 9 * self.f_diff(a + b *
sqrt(3 / 5)) +
                   5 / 9 * self.f_diff(a - b *
sqrt(3 / 5)) +
                   8 / 9 * self.f_diff(a + b * 0)) *
b)
        elif n == 4:
            I = float(((18 + sqrt(30)) / 36 *
self.f_diff(a + b * sqrt(3 / 7 - 2 / 7 * sqrt(6 /
5))) +
                       (18 + sqrt(30)) / 36 *
self.f_diff(a - b * sqrt(3 / 7 - 2 / 7 * sqrt(6 /
5))) +
                       (18 - sqrt(30)) / 36 *
self.f_diff(a + b * sqrt(3 / 7 + 2 / 7 * sqrt(6 /
5))) +
                       (18 - sqrt(30)) / 36 *
self.f_diff(a - b * sqrt(3 / 7 + 2 / 7 * sqrt(6 /
5)))) * b)
```

```
            elif n == 5:
                I = float(((322 + 13 * sqrt(70)) / 
900 * self.f_diff(a + 1 / 3 * b * sqrt(5 - 2 * 
sqrt(10 / 7))) +
                           (322 + 13 * sqrt(70)) / 
900 * self.f_diff(a - 1 / 3 * b * sqrt(5 - 2 * 
sqrt(10 / 7))) +
                           (322 - 13 * sqrt(70)) / 
900 * self.f_diff(a + 1 / 3 * b * sqrt(5 + 2 * 
sqrt(10 / 7))) +
                           (322 - 13 * sqrt(70)) / 
900 * self.f_diff(a - 1 / 3 * b * sqrt(5 + 2 * 
sqrt(10 / 7))) +
                           128 / 225 * self.f_diff(a 
+ b * 0)) * b)
            else:
                I = 0

        if not speed_test:
            print("The integration of f'(x) under 
n={0} on [{1},{2}] is 
{3}".format(n,self.start,self.end,I))

            if dis_error:
                correct = 4.21009627762213
                print("The correct result is {0}, 
the error is {1}"
                                   .format( correct, 
np.abs(correct - I)))
```

Result

```
Testing Gaussian Quadrature !
The integration of f'(x) under n=3 on [1.5,2.0] is
-15.735090897799367
The correct result is 4.21009627762213, the error is
19.94518717542l495
The integration of f'(x) under n=4 on [1.5,2.0] is
2.50913009345486
The correct result is 4.21009627762213, the error is
1.7009661841672696
The integration of f'(x) under n=5 on [1.5,2.0] is
2.0362350023993208
The correct result is 4.21009627762213, the error is
2.173861275222809
```

Speed Test 2

We find that the **efficiency** of the three algorithms in descending order is: **Guassain Quadrature > Midpoint Rule > Simpson 1/3** . Even if we use vector multiplication, the speed of Simpson 1/3 is still slow.

The accuracy in descending order is: **Simpson 1/3 > Midpoint Rule > Guassian Quadrature(n = 3,4,5)**, however, when $n > 20$, Guassain output perform both Simpson 1/3 and Midpoint Rule Method. Below is the table of the results:

n	Estimation Result
1	-13.2620943
2	13.89111531
3	-15.7350909
4	2.509130094
5	2.036235001
6	16.22896135
7	-3.282233929
8	5.265206035
9	4.670227722
10	4.029624315
11	4.221936163
12	4.21455594
13	4.209246185
14	4.210087542
15	4.210111532
16	4.210095321
17	4.210096152
18	4.210096297
19	4.210096273
20	4.210096278
21	4.210096275
22	4.210096271

n	Estimation Result
23	4.210096284
24	4.210096282
25	4.210096291

Code

```
#Speed test for computing integration, 200 tests for averaging
    def speed_test2(self):
        print("Testing Midpoint Rule")
        times = []
        for i in range(200):
            start = time.time()
            self.midpoint(10,speed_test=True)
            self.midpoint(10 ** 2,speed_test=True)
            self.midpoint(10 ** 3,speed_test=True)
            times.append(time.time()-start)
        print("Time taken on average: {}".format(np.mean(times)))


        print("Testing Simpson Rule")
        times = []
        for i in range(200):
            start = time.time()
            self.simpson(10 ,speed_test=True)
            self.simpson(10 ** 2 ,speed_test=True)
            self.simpson(10 ** 3 ,speed_test=True)
            times.append(time.time() - start)
        print("Time taken on average: {}".format(np.mean(times)))
```

```
        print("Testing Guassian Quadrature")
        times = []
        for i in range(200):
            start = time.time()
            self.gaussian(3,speed_test=True)
            self.gaussian(4,speed_test=True)
            self.gaussian(5,speed_test=True)
            times.append(time.time() - start)
        print("Time taken on average:
{}".format(np.mean(times)))
```

```
Testing Midpoint Rule
Time taken on average: 0.30989138007164
Testing Simpson Rule
Time taken on average: 0.31891303896903994
Testing Guassian Quadrature
Time taken on average: 0.002977032661437988З
```

Code Overview

```
import numpy as np
import matplotlib.pyplot as plt
from sympy import *
from scipy import linalg
import time

class HW2():

    x = symbols('x')
    func1 = sin(x**5) + x**3
    func2 = 5*x**4*cos(x**5) + 3*x**2
```

```
    start = 1.5
    end= 2.0


    def __init__(self):
        self.first_diff = diff(self.func1, self.x)
        # self.first_diff = self.func2

    # 2.1 Central Difference Method
    def cdm(self,h,x0,speed_test=False,dis_error =
False):
        """
        :param h: the step size/ solution
        :param x0: center point
        :param speed_test: whether to take speed
test, switch off to exclude the effect of print
statement
        :param dis_error: switch off for speed test
to exclude the effect of print estimation error
        :return:
        """


        res = (self.f(x0+h) - self.f(x0-h))/(2*h)

        if not speed_test:
            print("f'({0}) under h={1} is
{2}".format(x0,h,res))

            if dis_error:
                correct = self.f_diff(x0)
                print("The correct result is {3},
the error is {4}"
                      .format(x0, h,
res,correct,np.abs(correct - res)))



    # 2.2 Forward Difference Method
```

```
    def fdm(self,h,x0,speed_test=False,dis_error =
False):
        """
        :param h: the step size/ resolution
        :param x0: center point
        :param speed_test: whether to take speed
test, switch off to exclude the effect of print
statement
        :param dis_error: switch off for speed test
to exclude the effect of print estimation error
        :return:
        """


        res = (self.f(x0 + h) - self.f(x0)) /  h

        if not speed_test:
            print("f'({0}) under h={1} is
{2}".format(x0, h, res))

            if dis_error:
                correct = self.f_diff(x0)
                print("The correct result is {3},
the error is {4}"
                      .format(x0, h,
res,correct,np.abs(correct - res)))

    # 2.3 Midpoint Rule
    def midpoint(self,n, speed_test=False,dis_error
= False):
        """
        :param n: Number of points
        :param speed_test: whether to take speed
test, switch off to exclude the effect of print
statement
        :param dis_error: switch off for speed test
to exclude the effect of print estimation error
        :return:
```

```
        """

        delta_x = (self.end-self.start)/(n-1)
        sep_points =
np.linspace(self.start,self.end,n)
        v1 = np.array(sep_points[:-1])
        v2 = np.array(sep_points[1:])
        mid_points = (v1+v2)/2
        result = delta_x*sum(map(lambda
x:self.f_diff(x),mid_points))

        if not speed_test:
            print("The integration of the f'(x)
under n={0} on [{1},{2}] is
{3}".format(n,1.5,2.0,result))

            if dis_error:
                correct = 4.21009627762213
                print("The correct result is {0},
the error is {1}"
                      .format( correct,
np.abs(correct - result)))

    # 2.4 Simpson's 1/3 Rule
    def simpson(self,n,speed_test=False,dis_error =
False):
        """
        :param n: Number of points
        :param speed_test: whether to take speed
test, switch off to exclude the effect of print
statement
        :param dis_error: switch off for speed test
to exclude the effect of print estimation error
        :return:
        """

        n = n+1
```

```
        delta_x = (self.end-self.start)/(n-1)
        points = np.linspace(self.start,self.end,n)
        if n > 3:
            coeff_vec = np.array([1]+[4,2]*((n-
3)//2)+[4]+[1],dtype=float)
        else:
            coeff_vec =
np.array([1,4,1],dtype=float)
        function_value =
np.array([self.f_diff(point) for point in
points],dtype=float)
        result =
delta_x*coeff_vec.dot(function_value)/3

        if not speed_test:
            print("The integration of f'(x) under n=
{0} on [{1},{2}] is
{3}".format(n,self.start,self.end,result))

            if dis_error:
                correct = 4.21009627762213
                print("The correct result is {0},
the error is {1}"
                      .format( correct,
np.abs(correct - result)))

    # 2.5 Gaussian Quadrature
    def gaussian(self,n,speed_test=False,dis_error =
False):
        """
        :param n: Number of points
        :param speed_test: whether to take speed
test, switch off to exclude the effect of print
statement
        :param dis_error: switch off for speed test
to exclude the effect of print estimation error
        :return:
```

```
        """
        # x = 1.75+0.25xd, where xd ~ [-1,1]
        a = (self.end+self.start)/2  # 1.75
        b = (self.end-self.start)/2  # 0.25
        if n == 3:
            I = float((5 / 9 * self.f_diff(a + b *
sqrt(3 / 5)) +
                       5 / 9 * self.f_diff(a - b *
sqrt(3 / 5)) +
                       8 / 9 * self.f_diff(a + b *
0)) * b)
        elif n == 4:
            I = float(((18 + sqrt(30)) / 36 *
self.f_diff(a + b * sqrt(3 / 7 - 2 / 7 * sqrt(6 /
5))) +
                    (18 + sqrt(30)) / 36 *
self.f_diff(a - b * sqrt(3 / 7 - 2 / 7 * sqrt(6 /
5))) +
                    (18 - sqrt(30)) / 36 *
self.f_diff(a + b * sqrt(3 / 7 + 2 / 7 * sqrt(6 /
5))) +
                    (18 - sqrt(30)) / 36 *
self.f_diff(a - b * sqrt(3 / 7 + 2 / 7 * sqrt(6 /
5)))) * b)

        elif n == 5:
            I = float(((322 + 13 * sqrt(70)) / 900 *
self.f_diff(a + 1 / 3 * b * sqrt(5 - 2 * sqrt(10 /
7))) +
                       (322 + 13 * sqrt(70)) / 900
* self.f_diff(a - 1 / 3 * b * sqrt(5 - 2 * sqrt(10 /
7))) +
                       (322 - 13 * sqrt(70)) / 900
* self.f_diff(a + 1 / 3 * b * sqrt(5 + 2 * sqrt(10 /
7))) +
```

```
                        (322 - 13 * sqrt(70)) / 900
* self.f_diff(a - 1 / 3 * b * sqrt(5 + 2 * sqrt(10 /
7))) +
                        128 / 225 * self.f_diff(a +
b * 0)) * b)
        else:
            I = 0

        if not speed_test:
            print("The integration of f'(x) under n=
{0} on [{1},{2}] is
{3}".format(n,self.start,self.end,I))

            if dis_error:
                correct = 4.21009627762213
                print("The correct result is {0},
the error is {1}"
                      .format( correct,
np.abs(correct - I)))

    def f(self,value):
        """
        Calculate the value of a given function at
specified x
        :param func: sympy function object
        :param value: value of independent variable
        :return: None
        """
        return float(self.func1.evalf(subs={self.x:
value}))

    def f_diff(self,value):
        return float(self.first_diff.evalf(subs=
{self.x: value}))


    def test(self):
```

```
       print("Testing Central Difference Method !")
       self.cdm(10e-3, 1.5,dis_error=True)
       self.cdm(10e-4, 1.5,dis_error=True)
       self.cdm(10e-3, 1.7,dis_error=True)
       self.cdm(10e-4, 1.7,dis_error=True)

print("####################################")

       print("Testing Forward Difference Method !")
       self.fdm(10e-3, 1.5,dis_error=True)
       self.fdm(10e-4, 1.5,dis_error=True)
       self.fdm(10e-3, 1.7,dis_error=True)
       self.fdm(10e-4, 1.7,dis_error=True)

print("####################################")

       print("Testing Midpoint Rule !")
       self.midpoint(10,dis_error=True)
       self.midpoint(10**2,dis_error=True)
       self.midpoint(10**3,dis_error=True)

print("####################################")

       print("Testing Simpson's 1/3 Rule !")
       self.simpson(10,dis_error=True)
       self.simpson(10**2,dis_error=True)
       self.simpson(10**3,dis_error=True)

print("####################################")

       print("Testing Gaussian Quadrature !")
       self.gaussian(3,dis_error=True)
       self.gaussian(4,dis_error=True)
       self.gaussian(5,dis_error=True)

print("####################################")
```

```
    # Speed test for computing differentiation
    def speed_test1(self):
        print("Speed testing central difference
method!")
        times = []

        for i in range(200):
            start = time.time()
            self.cdm(10e-3, 1.5,True)
            self.cdm(10e-4, 1.5,True)
            self.cdm(10e-3, 1.7,True)
            self.cdm(10e-4, 1.7,True)
            times.append(time.time()-start)

        print("Central Difference Method takes {0}
on average".format(np.mean(times)))

        print("Speed testing forward difference
method!")
        times = []
        for i in range(200):
            start = time.time()
            self.fdm(10e-3, 1.5,True)
            self.fdm(10e-4, 1.5,True)
            self.fdm(10e-3, 1.7,True)
            self.fdm(10e-4, 1.7,True)
            times.append(time.time() - start)

        print("Forward Difference Method takes {0}
on average".format(np.mean(times)))


    #Speed test for computing integration
    def speed_test2(self):
        print("Testing Midpoint Rule")
        times = []
```

```
        for i in range(200):
            start = time.time()
            self.midpoint(10,speed_test=True)
            self.midpoint(10 ** 2,speed_test=True)
            self.midpoint(10 ** 3,speed_test=True)
            times.append(time.time()-start)
        print("Time taken on average:
{}".format(np.mean(times)))


        print("Testing Simpson Rule")
        times = []
        for i in range(200):
            start = time.time()
            self.simpson(10 ,speed_test=True)
            self.simpson(10 ** 2 ,speed_test=True)
            self.simpson(10 ** 3 ,speed_test=True)
            times.append(time.time() - start)
        print("Time taken on average:
{}".format(np.mean(times)))



        print("Testing Guassian Quadrature")
        times = []
        for i in range(200):
            start = time.time()
            self.gaussian(3,speed_test=True)
            self.gaussian(4,speed_test=True)
            self.gaussian(5,speed_test=True)
            times.append(time.time() - start)
        print("Time taken on average:
{}".format(np.mean(times)))


if __name__ =="__main__":
```

```
    hw2 = HW2()

    hw2.test()

    # hw2.speed_test1()

    hw2.speed_test2()
```