mathhw3

October 20, 2021

1 CS 21850 HW 3 Zack Wang

[ ]:
```
import math
```

1.1 Overview of Gaussian Rule

We basically want to make a function that estimates the value of the integral on the chunk using w(x) * f(p) for some point(s).

To do so, we need to identify 1) the Gauss point(s) on the interval and 2) the corresponding coefficients w_i.

By Theorem 13.3 in the textbook, this Gaussian quadrature will exactly equal any function of 2n-1. I.E if we use one point, we can accurately calculate any $f(x)w(x) \in P_1$ and if we use two points we can accurately calculate any $f(x)w(x) \in P_3$. We will use these assumptions when writing our G1 and G2 functions.

1.1.1 G1 Function: 1-Point Gauss Rule

For G1, we need to generate a single Gauss point p_1 and w_1. To generate the weight function, we take $f(x) = 1$ and solve the equation $\int_a^b w(x)\,dx = w_1$. We then solve for the Gauss point by assuming f(x) is of the form $f(x) = x - gp$. We solve the equation $\int_a^b w(x)(x - gp)\,dx = w_1 * gp$

1.1.2 G2 Function: 2-Point Gauss Rule

For G2, we need to find two Gauss points and two weights w_1 and w_2. To find these values, we need four functions. To get the first two, we use the same method as in G1, where we set f = 1, f = x. Then, we exploit the fact that we are dealing with orthogonal polynomials, so (1, p_2) = 0 = (x, p_2) and thus we can solve for p_0 and p_1.

With the two inner products, we are able to solve for the sum of points $p_0 + p_1$. Because we assume that the Gauss points are evenly spaced, of form $p_i = mu + -delta$, we solve for mu. This then lets us solve for the delta value, giving us all the information needed to calculate the Gauss points.

[ ]:
```
#G1 1-point rule
def computeExInt(a, b, p):
```

```
    #given bounds and a power of x in the integral, will return val
    #Example: to find int(x^5) from 0 to 1 do computeExInt(0, 1, 5)
    return ((b ** (p + 1)) / (p + 1)) - (a ** (p + 1)) / (p + 1)

def getWeight(a, b, p):
    #returns weight_0, checked and seems to work
    newE = p + 1
    return ((b ** newE) - (a ** newE)) / newE

def getPoint(a, b, p, w):
    return  computeExInt(a, b, p + 1) / w

def g1(a, b, foo, p):
    #takes in bounds, spits out value
    weight = getWeight(a, b, p)
    point = getPoint(a, b, p, weight)
    return weight * foo(point)
```

[ ]:
```
g1(0, 1, lambda x : x, 0)
```

[ ]:
```
0.5
```

[ ]:
```
#G2 2-Point Rule
def getPoints(a, b, p):
    c = computeExInt(a, b, p + 1) + computeExInt(a, b, p + 2)
    d = computeExInt(a, b, p) + computeExInt(a, b, p + 1)
    e = computeExInt(a, b, p + 3) + computeExInt(a, b, p + 2)
    mu = ((computeExInt(a, b, p + 2) - (e * (computeExInt(a, b, p) / d))) /␣
 ↪(computeExInt(a, b, p + 1) - (c * computeExInt(a, b, p) / d)) ) / 2
    b = ((2 * mu * c) - e) / d
    mu2 = mu ** 2
    delta = min(((2 * mu) - (((4 * mu2) - (4 * (mu2 - b)))** .5)) / 2, ((2 *␣
 ↪mu) + (((4 * mu2) - (4 * (mu2 - b)))** .5)) / 2)
    #s = (() + ()) * (p + 3) / (())
    return mu - delta, mu + delta

def getWeights(a, b, p):
    #returns weight_0, checked and seems to work
    p0, p1 = getPoints(a, b, p)
    w1 = (computeExInt(a, b, p+1) - (p0 * computeExInt(a, b, p))) / (p1 - p0)
    w0 = computeExInt(a, b, p) - w1
    return w0, w1

def g2(a, b, foo, p):
    w0, w1 = getWeights(a, b, p)
    p0, p1 = getPoints(a, b, p)
    return (w0 * foo(p0)) + (w1 * foo(p1))
```

[ ]:
```
g2(0, 1, lambda x: x, .5)
```

[ ]:
```
0.4
```

[ ]:
```
def approximate(foo, a, b, chunks, p):
    dist = b - a
    stepsize = dist / chunks
    points = list(range(chunks + 1))
    points = list(map(lambda y: y * stepsize, points))
    points = list(zip(points, points[1:]))
    val = 0
    for left, right in points:
        val += g1(left, right, foo, p)
    return val

def approximate2(foo, a, b, chunks, p):
    dist = b - a
    stepsize = dist / chunks
    points = list(range(chunks + 1))
    points = list(map(lambda y: y * stepsize, points))
    points = list(zip(points, points[1:]))
    val = 0
    for left, right in points:
        val += g2(left, right, foo, p)
    return val
```

[ ]:
```
approximate(lambda x: x, 0, 5, 1, 0)
```

[ ]:
```
12.5
```

[ ]:
```
print("CASE 0")
print("One Point Approximations")
c0n1 = approximate(lambda x: x, 0, 1, 1, 2)
c0n2 = approximate(lambda x: x, 0, 1, 2, 2)
c0n8 = approximate(lambda x: x, 0, 1, 8, 2)
c0ex = approximate(lambda x: x, 0, 1, 20, 2)
print("1 Chunk: {}, Err = {}".format(c0n1, c0n1 - c0ex))
print("2 Chunk: {}, Err = {}".format(c0n2, c0n2 - c0ex))
print("8 Chunk: {}, Err = {}".format(c0n8, c0n8 - c0ex))
print("Exact: {}".format(c0ex))

print("Two Point Approximations")
c0n12 = approximate2(lambda x: x, 0, 1, 1, 2)
c0n22 = approximate2(lambda x: x, 0, 1, 2, 2)
c0n82 = approximate2(lambda x: x, 0, 1, 8, 2)
c0ex2 = approximate2(lambda x: x, 0, 1, 20, 2)
print("1 Chunk: {}, Err = {}".format(c0n12, c0n12 - c0ex2))
```

```
print("2 Chunk: {}, Err = {}".format(c0n22, c0n22 - c0ex2))
print("8 Chunk: {}, Err = {}".format(c0n82, c0n82 - c0ex2))
print("Exact: {}".format(c0ex2))
```

```
CASE 0
One Point Approximations
1 Chunk: 0.25, Err = 0.0
2 Chunk: 0.25, Err = 0.0
8 Chunk: 0.25, Err = 0.0
Exact: 0.25
Two Point Approximations
1 Chunk: 0.25, Err = 0.0
2 Chunk: 0.25, Err = 0.0
8 Chunk: 0.25, Err = 0.0
Exact: 0.25
```

[ ]:
```
print("CASE 1")
print("One Point Approximations")
c1n1 = approximate(lambda x: x ** 3, 0, 1, 1, -.5)
c1n2 = approximate(lambda x: x ** 3, 0, 1, 2, -.5)
c1n8 = approximate(lambda x: x ** 3, 0, 1, 8, -.5)
c1ex = approximate(lambda x: x ** 3, 0, 1, 20, -.5)
print("1 Chunk: {}, Err = {}".format(c1n1, c1n1 - c1ex))
print("2 Chunk: {}, Err = {}".format(c1n2, c1n2 - c1ex))
print("8 Chunk: {}, Err = {}".format(c1n8, c1n8 - c1ex))
print("Exact: {}".format(c1ex))

# 2-POINT APPROX
print("Two Point Approximations")
c1n12 = approximate2(lambda x: x ** 3, 0, 1, 1, -.5)
c1n22 = approximate2(lambda x: x ** 3, 0, 1, 2, -.5)
c1n82 = approximate2(lambda x: x ** 3, 0, 1, 8, -.5)
c1ex2 = approximate2(lambda x: x ** 3, 0, 1, 20, -.5)
print("1 Chunk: {}, Err = {}".format(c1n12, c1n12 - c1ex2))
print("2 Chunk: {}, Err = {}".format(c1n22, c1n22 - c1ex2))
print("8 Chunk: {}, Err = {}".format(c1n82, c1n82 - c1ex2))
print("Exact: {}".format(c1ex2))
```

```
CASE 1
One Point Approximations
1 Chunk: 0.07407407407407406, Err = -0.2112220647549084
2 Chunk: 0.2398101383019598, Err = -0.045486000527022674
8 Chunk: 0.28307448411503366, Err = -0.0022216547139488063
Exact: 0.28529613882898247
Two Point Approximations
1 Chunk: 0.09638771043967392, Err = -0.18890919471818485
2 Chunk: 0.24194983031834522, Err = -0.043347074839513555
8 Chunk: 0.28309281856594515, Err = -0.002204086591913623
```

```
Exact: 0.28529690515785877
```

[ ]:
```
print("CASE 2")
print("One Point Approximations")
c2n1 = approximate(lambda x: x ** .5, 0, 1, 1, -.5)
c2n2 = approximate(lambda x: x ** .5, 0, 1, 2, -.5)
c2n8 = approximate(lambda x: x ** .5, 0, 1, 8, -.5)
c2ex = approximate(lambda x: x ** .5, 0, 1, 20, -.5)
print("1 Chunk: {}, Err = {}".format(c2n1, c2n1 - c2ex))
print("2 Chunk: {}, Err = {}".format(c2n2, c2n2 - c2ex))
print("8 Chunk: {}, Err = {}".format(c2n8, c2n8 - c2ex))
print("Exact: {}".format(c2ex))

# 2-POINT APPROX
print("Two Point Approximations")
c2n12 = approximate2(lambda x: x ** .5, 0, 1, 1, -.5)
c2n22 = approximate2(lambda x: x ** .5, 0, 1, 2, -.5)
c2n82 = approximate2(lambda x: x ** .5, 0, 1, 8, -.5)
c2ex2 = approximate2(lambda x: x ** .5, 0, 1, 20, -.5)
print("1 Chunk: {}, Err = {}".format(c2n12, c2n12 - c2ex2))
print("2 Chunk: {}, Err = {}".format(c2n22, c2n22 - c2ex2))
print("8 Chunk: {}, Err = {}".format(c2n82, c2n82 - c2ex2))
print("Exact: {}".format(c2ex2))
```

```
CASE 2
One Point Approximations
1 Chunk: 1.1547005383792515, Err = 0.14648906863538413
2 Chunk: 1.0797973851068847, Err = 0.0715859153630174
8 Chunk: 1.0204311209245787, Err = 0.012219651180711333
Exact: 1.0082114697438673
Two Point Approximations
1 Chunk: 1.144940018196995, Err = 0.13721831273427365
2 Chunk: 1.0749023843012995, Err = 0.06718067883857803
8 Chunk: 1.0192067207309603, Err = 0.011485015268238863
Exact: 1.0077217054627214
```

[ ]:
```
print("CASE 3")
print("One Point Approximations")
c3n1 = approximate(lambda x: (1 - math.sin(x)) ** 2, 0, 1, 1, -.5)
c3n2 = approximate(lambda x: (1 - math.sin(x)) ** 2, 0, 1, 2, -.5)
c3n8 = approximate(lambda x: (1 - math.sin(x)) ** 2, 0, 1, 8, -.5)
c3ex = approximate(lambda x: (1 - math.sin(x)) ** 2, 0, 1, 20, -.5)
print("1 Chunk: {}, Err = {}".format(c3n1, c3n1 - c3ex))
print("2 Chunk: {}, Err = {}".format(c3n2, c3n2 - c3ex))
print("8 Chunk: {}, Err = {}".format(c3n8, c3n8 - c3ex))
print("Exact: {}".format(c3ex))
```

```
# 2-POINT APPROX
print("Two Point Approximations")
c3n12 = approximate2(lambda x: (1 - math.sin(x)) ** 2, 0, 1, 1, -.5)
c3n22 = approximate2(lambda x: (1 - math.sin(x)) ** 2, 0, 1, 2, -.5)
c3n82 = approximate2(lambda x: (1 - math.sin(x)) ** 2, 0, 1, 8, -.5)
c3ex2 = approximate2(lambda x: (1 - math.sin(x)) ** 2, 0, 1, 20, -.5)
print("1 Chunk: {}, Err = {}".format(c3n12, c3n12 - c3ex2))
print("2 Chunk: {}, Err = {}".format(c3n22, c3n22 - c3ex2))
print("8 Chunk: {}, Err = {}".format(c3n82, c3n82 - c3ex2))
print("Exact: {}".format(c3ex2))
```

```
CASE 3
One Point Approximations
1 Chunk: 0.9053339520384431, Err = -0.18558171577230698
2 Chunk: 1.047274265397741, Err = -0.04364140241300918
8 Chunk: 1.0887088520757846, Err = -0.002206815734965506
Exact: 1.0909156678107501
Two Point Approximations
1 Chunk: 0.925616870227603, Err = -0.16530987809331033
2 Chunk: 1.0510350200511878, Err = -0.039891728269725535
8 Chunk: 1.0888209010966547, Err = -0.00210584722425855
Exact: 1.0909267483209133
```

[ ]:
```
print("CASE 4")
print("One Point Approximations")
c4n1 = approximate(lambda x: (x ** .5) / ((math.sin(x/2)) ** .5), 0, 1, 1, -.5)
c4n2 = approximate(lambda x: (x ** .5) / ((math.sin(x/2)) ** .5), 0, 1, 2, -.5)
c4n8 = approximate(lambda x: (x ** .5) / ((math.sin(x/2)) ** .5), 0, 1, 8, -.5)
c4ex = approximate(lambda x: (x ** .5) / ((math.sin(x/2)) ** .5), 0, 1, 20, -.5)
print("1 Chunk: {}, Err = {}".format(c4n1, c4n1 - c4ex))
print("2 Chunk: {}, Err = {}".format(c4n2, c4n2 - c4ex))
print("8 Chunk: {}, Err = {}".format(c4n8, c4n8 - c4ex))
print("Exact: {}".format(c4ex))

# 2-POINT APPROX
print("Two Point Approximations")
c4n12 = approximate2(lambda x: (x ** .5) / ((math.sin(x/2)) ** .5), 0, 1, 1, -.
 ↪5)
c4n22 = approximate2(lambda x: (x ** .5) / ((math.sin(x/2)) ** .5), 0, 1, 2, -.
 ↪5)
c4n82 = approximate2(lambda x: (x ** .5) / ((math.sin(x/2)) ** .5), 0, 1, 8, -.
 ↪5)
c4ex2 = approximate2(lambda x: (x ** .5) / ((math.sin(x/2)) ** .5), 0, 1, 20, -.
 ↪5)
print("1 Chunk: {}, Err = {}".format(c4n12, c4n12 - c4ex2))
print("2 Chunk: {}, Err = {}".format(c4n22, c4n22 - c4ex2))
print("8 Chunk: {}, Err = {}".format(c4n82, c4n82 - c4ex2))
```

```
print("Exact: {}".format(c4ex2))
```

```
CASE 4
One Point Approximations
1 Chunk: 2.8349880805797323, Err = -0.005335915147206105
2 Chunk: 2.839022499410468, Err = -0.001301496316470363
8 Chunk: 2.84025640085947, Err = -6.759486746821253e-05
Exact: 2.8403239957269384
Two Point Approximations
1 Chunk: 2.835550409847861, Err = -0.004773905723790328
2 Chunk: 2.839122923549124, Err = -0.0012013920225273367
8 Chunk: 2.840259559185011, Err = -6.475638664049654e-05
Exact: 2.8403243155716513
```

In summary, we have made a robust and adaptive program that, given any range and number of chunks, can estimate the particular integral. We see that the two point approximation universally does as well or better than the one point, with the increase in chunks improving the accuracy of the estimation.