Integration, Differentiation, and Root Finding
Project #2

Due on Saturday, March 7, 2015 (*extended)*

Inst. Pavel Snopok, Spring 2015

Michael Lee

March 9, 2015

Abstract

Intrinsically, calculus is a computationally intensive process. The algorithms to finding slopes, area under curves, and zeros of a function are straightforward but doing so by hand is a tedious and time-consuming activity. If we also assume that we are trying to solve nontrivial problems, our mathematical models demand that the underlying equations be complex and analytically unsolvable due to the very nature of the physical world. Since this is the case, we use numerical approximations not to attain perfect answers but to derive meaning and trends from the answer. Numerical methods also allows us to automate and discover those kinds of trends more efficiently by speeding the data creation process.

1 Introduction

Calculus is a powerful tool we have at our disposal. We use this tool to calculate the change of quantities in relation to one another. The development of modern calculus is credited to Sir Isaac Newton and Gottfried Wilhelm Leibniz in the late 17th century although work in the spirit of calculus existed long before in Ancient Greece, China, India, and Medieval Europe.

Calculus comes in two flavors. They are formally referred to as the First Fundamental Theorem of Calculus (1) and the Second Fundamental Theorem of Calculus (2). Colloquially, (1) and (2) states:

First Fundamental Theorem of Calculus
The quotient of the net change of a quantity over an infinitesimal change of the chosen variable gives the instantaneous change of a quantity.

Second Fundamental Theorem of Calculus
The sum of infinitesimal changes over a chosen variable adds up to the net change in the quantity.

In this lab, we write programs to calculate the derivative and integral of test functions. For integration, we use three methods: the trapezoid rule, Simpson's Rule, and the Gaussian Quadrature. For the derivative, we also use three methods: the forward difference, the central difference, and the extrapolated difference. We will also explore root finding by using the Newton-Raphson method.

2 Theory and Salient Details

2.1 Integration

2.1.1 Trapezoid Rule

The trapezoid rule is similar to the box counting method that is traditionally used for integration. Instead of adding the areas of boxes, however, we add the areas of trapezoids. This method improves the accuracy of the integration because there is less overshooting and clipping but it still results in a slight error because the curve is continuous whilst the trapezoids are discrete.

The area of each individual trapezoid is

$$A = \frac{1}{2}hf$$

where h is the base and f is the height of the trapezoid. To define these two values, we determine a region of integration $[a, b]$ and the number of points $N$where each point, excluding the first and last, splits the interval into even sections.

The base then simply becomes

$$h = \frac{b-a}{N-1}.$$

We use $N-1$ because the number of trapezoids is one less than the number of points.

The height of the trapezoid is the value of the function at x_i and x_{i+1} where i refers to the index split by N:

$$f = \frac{f(x_i) + f(x_{i+1})}{2}.$$

So the integral calculated through the trapezoid rule is:

$$\int f(x)\ dx = \frac{h}{2}f(x_1) + hf(x_2) + ... + \frac{h}{2}f(x_N)$$

The weights of the indices $1 < i < N$ are just h because they are counted twice. Explicitly, for two trapezoids

$$A_{total} = \frac{1}{2}hf_1 + \frac{1}{2}hf_2 =$$

$$\frac{h}{2}(f(x_1) + f(x_2) + f(x_2) + f(x_3)) =$$

$$\frac{h}{2}f(x_1) + hf(x_2) + \frac{h}{2}f(x_3).$$

We can write the weight function as

$$w_i = \{\frac{h}{2}, h, h, ... , \frac{h}{2}\}.$$

It is instructive to write

$$\int f(x)\ dx = \sum_{i=1}^{N} w_i f(x_i).$$

2.1.2 Simpson's Rule

Simpson's Rule is a way to numerically approximate a complex integral by transforming the function between an equally split interval of $[a, b]$ into a parabola so that

$$f(x) = \alpha x^2 + \beta x + \gamma.$$

This method is highly accurate and has an error of about 10^9 times less than the trapezoid rule. For the case where the interval is $[-1, 1]$, the area under the parabola is

$$\int_{-1}^{1} \alpha x^2 + \beta x + \gamma\ dx = \frac{2\alpha}{3} + 2\beta$$

Notice that

$$f(-1) = \alpha - \beta + \gamma,\ f(0) = \gamma,\ f(1) = \alpha + \beta + \gamma$$

so then

$$\alpha = \frac{f(-1) + f(1)}{2} - f(0),\ \gamma = f(0),$$

$$\beta = \frac{f(1) - f(-1)}{2} - f(0).$$

Eliminating the dummy variables gives us for the integral

$$\int_{-1}^{1} \alpha x^2 + \beta x + \gamma\ dx = \frac{f(-1)}{3} + \frac{4f(0)}{3} + \frac{f(1)}{3}$$

In general, the integral becomes

$$\int_{x_i-h}^{x_i+h} f(x)\ dx = \int_{x_i-h}^{x} f(x)\ dx + \int_{x}^{x_i+h} f(x)\ dx =$$

$$\frac{h}{3}f_{i-1} + \frac{4h}{3}f_i + \frac{h}{3}f_{i+1}$$

The Simpson's Rule requires that you have pairs of intervals, meaning that there needs to be at least three points for every elementary integration. Because of this, N has to be odd and greater than 3.

For any arbitrary interval [a,b], we write then

$$\int_a^b f(x)\ dx = \frac{h}{3}f_1 + \frac{4h}{3}f_2 + \frac{2h}{3}f_3 +$$

$$... + \frac{2h}{3}f_{N-2} + \frac{4h}{3}f_{N-1} + \frac{h}{3}f_N =$$

$$\sum_{i=1}^{N} w_i f_i$$

where

$$w_i = \{\frac{h}{3} + \frac{4h}{3} + \frac{2h}{3} + ... + \frac{4h}{3} + \frac{h}{3}\}.$$

2.1.3 Gaussian Quadrature

The Gaussian Quadrature approaches integration by approximating $f(x) = W(x)g(x)$ so that the integral becomes

$$\int f(x)\ dx = \int W(x)g(x)\ dx = \sum_{i=1}^{N} w_i g_i.$$

This allows you to choose the weighting function and a $2N-1$, where N is the number of integration points, degree polynomial for g(x) in order to make the error vanish. This result is extraordinary because the number of points necessary to approximate a solution would not need to be very large since the polynomial order grows like $2N$. With only a few integration points, you can obtain an error that would take the other methods hundreds of points to attain.

The book omits an analysis of the Gaussian integration points and weights in Chapter 6 so we shall assume that it is beyond the scope of this course.

2.2 Differentiation

2.2.1 Forward Difference

The forward difference is what we typically see when we learn the First Fundamental Theorem of Calculus, disregarding limits. The description for the forward difference is

$$f'_{fd} = \frac{f(x+h) - f(x)}{h}.$$

This is straightforward and gives us the derivative of f at x given a small change h. This method is called the forward difference because it looks for the point $x+h$ from x in order to calculate the slope. To obtain a f' that is close to the actual tangent line at that point, we reduce h to a very small number.

2.2.2 Central Difference

The central difference, instead of looking ahead, tries to find the slope by taking the point $h/2$ step ahead and subtracting it from a point $h/2$ step behind.

$$f'_{cd} = \frac{f(x+h/2) - f(x-h/2)}{h}$$

The error for the central difference will be smaller than the forward difference by a factor of about 1000. Like the forward difference, this approximation is only good for when h is very small.

2.2.3 Extrapolated Difference

The extrapolated difference uses the central difference a half step back and a half step forward. Essentially, it sets $\underline{h}$ to $\frac{h}{2}$ and it then reduces to

$$f'_{ed} = \frac{4f'_{cd}(x+\frac{h}{2}) - f'_{cd}(x+h)}{3h}$$

2.3 Newton-Raphson Root Finding

The Newton-Raphson algorithm is used to solve the equation

$$f(x) = 0$$

or in other words, to find the roots of $f(x)$. The algorithm works like this:

1. Start with a guess x_0.
2. Draw a line tangent to the curve at the point x_0 to the x-axis.
3. Set x_1 to where the tangent line crosses zero.
4. Calculate the value of $f(x_1)$ and if it is not zero, use it as the new guess.
5. Repeat steps 1 - 4 until you obtain a x_i such that $f(x_i) = 0$.

The algorithm is pretty simple and can be expressed as

$$f(x_0) + \frac{dx}{dt}|_{x_0}\Delta x = 0.$$

$$\Delta x = \frac{-f(x_0)}{\frac{dx}{dt}|_{x_0}}$$

Δx gives you the correction factor to find your new guess through

$$x_i = x_{i-1} + \Delta x$$

You continue doing this until $f(x_i) = 0$.

3 Test Cases

3.1 Trapezoid Rule

For our test function, we use $f(x) = x^2$ because we know the analytic solution. The output after entering the function with limits of integration from $x = 0$ to $x = 1$ and with $N = 500$ is 0.333334. This is a good approximation with $\epsilon =$ 6.666e-7 but we can do better if we increase N.

3.2 Simpson's Rule

Using the test function x^2, we get for the value of the integral with limits of integration from $x = 0$ to $x = 1$ and with $N = 500$ to be 0.333333. It does not look like it made a difference but if we write

```
1    a = simpson(f, 0, 1.0, 500)
2    b = a - 1/3.0
3    print b
```

we find that $\epsilon =$ 1.11022302463e-16, nine orders of magnitude better than the trapezoid rule.

3.3 Gaussian Quadrature

Our test function is $f(x) = x^4$ with 3 integration points from $x = 0$ to $x = 2$ gives a value of 6.4 with an error in 1e-7. The Gaussian quadrature is highly accurate.

3.4 Differentiation

Since the coding involved in finding the derivative is minuscule, we wrote the forward, central, and extrapolated differences in one file. We can then use

just one test function for all three methods. We use $f(x) = x^3$ at $x = 3$ with $h =$ 1e-6.

```
    The value of the forward difference
    is 27.00000901, and its error
    is 9.005462e-06.
    The value of the central difference
    is 27.00000000643854, and its error
    is 6.438540e-09.
    The value of the extrapolated
    difference is 26.9999999732798805,
    and its error is 2.672012e-08.
```

3.5 Newton-Raphson Root Finding

For root finding, we use the test function $f(x) = 2cos(x) - x$. The output that we are interested in is

```
    Iteration =  12 , x = 1.02987111183,
    f(x) = -1.24390371776e-05
    Root found, tolerance eps = 1e-06
```

showing us that after the 12th cycle through the algorithm, the program found a desirable x value that met the error threshold we set, which was 1e-06. The zero is $x = 1.02987111183$ and this value agrees with a stronger integration machine like Wolfram Alpha.

4 One Dimensional Harmonic Oscillator

We will test our numerical methods on a simplified real world problem. Consider a one dimensional harmonic oscillator. Lets assume that this is a conservative system such that there is no energy lost due to friction or heat. Then the total energy of the system is:

$$E = K + U(x).$$

The problem is to determine the amount of time it would take for a block of unit mass to travel from $x = 0$ to $x = 1$. We simplify the problem by setting $E = m = k = 1$ so that the only units of interest are time and distance. We set $U(x) = \frac{1}{4}x^4$ to make this problem difficult to solve analytically but easy to solve numerically. This showcases the power and practicality of writing programs to solve calculus problems. So we have for our equation for the total energy of the system:

$$1 = \frac{1}{2}(\frac{dx}{dt})^2 + \frac{1}{4}x^4$$

By moving terms to the left and right hand sides and separating variables, we obtain for the time function:

$$t = \int_0^1 \sqrt{\frac{2}{4 - x^4}}\, dx$$

Using the Simpson's Rule program we wrote, we obtain $t = 0.72694593547$ unit time. Wolfram Alpha agrees to this value.

5 Error Assessment

We will assess the error of the forward, central, and extrapolated differences since they are straightforward and the easiest to analyze. Our test function will be $f(x) = x^3$, our x point at $x = 4$ and the values we will use for h are $h =$ {1e-2, 1e-4, 1e-6, 1e-8, 1e-10}

h	fd ϵ	cd ϵ	ed ϵ
1e-2	1.201e-01	2.500e-05	5.847e-12
1e-4	1.200e-03	2.375e-09	6.558e-12
1e-6	1.200e-05	1.460e-08	1.381e-08
1e-8	2.917e-07	1.839e-06	4.682e-06
1e-10	3.971e-06	3.971e-06	5.644e-04

We see that as h gets smaller, the errors of the forward and central differences begin to converge. This is because $f(x+h) - f(x)$ reaches the roundoff error limit where the difference is less than 10^{-15} and so the approximate error simply becomes the machine precision over h

$$\epsilon_{approx} = \frac{f(x+h) - f(x)}{h} \approx \frac{\epsilon_m}{h}.$$

We see that this is the case for $h =$ 1e-10 since $\frac{10^{-15}}{10^{-10}} = 10^{-5}$.

6 Conclusion

We were able to write programs to calculate integrals, compute derivatives, and find the roots of functions. Using the integration methods listed in this paper, we were able to compare each method and saw how the

Gaussian quadrature bested the trapezoid and Simpson's rule in terms of power and accuracy. For differentiation, we saw how the central difference produced an error 1000 times less than that of the forward difference. As the quantity h decreased, however, the error of both methods converged. We learned that this is because of the limitation of computers and their machine precision, e_m.

By solving a real world problem, albeit simplified, we showed the usefulness of writing code to solve for derivatives and integrals. Numerical techniques allow us to obtain solutions for difficult-to-solve equations that probably have no analytic solutions, making it possible for us to gain insight to the problem without the mess of developing new theories.

References

[1] R. Landau, Manual J. Paez, Cristian C. Bordeianu. *A Survey of Computational Physics*, 2010: Princeton University Press.

[2] D. Beazley and Brian K. Jones. *The Python Cookbook*, 2013: O'Reilly.

[3] Newman, M. E. J. *Computational Physics*, 2013: Createspace.

[4] Wikipedia. *Calculus*, 2015: Wikimedia Foundation. http://en.wikipedia.org/wiki/Calculus.

7 Appendix

Trapezoid Rule

```
"""
Written by Michael Lee.

PHYS440: Computational Physics
Project 2, Differentiation, Integration, and Root Finding
"""
import numpy as np

def f(x):
    # Use this as a test function.
    return x**2

def weight(index, step_size, number_of_steps):
    # Returns the weight depending on the index.
    # If it equals 0 or N, returns h.
    # if it does not equal 0 or N, returns h/2.
    if (index == 0) or (index == number_of_steps):
        weight = step_size / 2.0
    else:
        weight = step_size
    return weight

def integral(function, xmin, xmax, number_of_steps):
    # Adds up the trapezoid areas by using
    # the function values and the
    # weight function.
    step = float((xmax - xmin) / number_of_steps)
    sum = 0
    for i in range(0, number_of_steps+1):
        x = xmin + i * step
        w = weight(i, step, number_of_steps)
        sum = sum + w * function(x)
    return sum

if __name__ == "__main__":

    a = integral(f, 0, 1.0, 500)
    error = np.abs(a - (1/3.))
    print "The value of the integral by the trapezoid rule is %.8f " \
                          "and its error value is %e." % (a, error)
```

Simpsons Rule

```
"""
Written by Michael Lee.

PHYS440: Computational Physics
Project 2, Differentiation, Integration, and Root Finding
"""

import numpy as np
```

```
9
10 def f(x):
11     # Use this as your test function.
12     return x**2
13
14 def weight(index, step_size, number_of_steps):
15     if ((index == 0) or (index == number_of_steps)):
16         weight = step_size / 3.0
17     elif (index%2 == 1):
18         weight = 4*step_size / 3.0
19     elif (index != 0) and (index != number_of_steps) and (index%2 == 0):
20         weight = 2*step_size / 3.0
21     return weight
22
23 def simpson(function, xmin, xmax, number_of_steps):
24     step = float((xmax - xmin)/number_of_steps)
25     sum = 0
26     for i in range(0, number_of_steps+1):
27         x = xmin+i*step
28         w = weight(i, step, number_of_steps)
29         sum = sum + w*function(x)
30     return sum
31
32 if __name__ == '__main__':
33
34     a = simpson(f, 0, 1.0, 500)
35     error = np.abs(a-1/3.)
36     print "The value of the integral for Simpson's rule is"\
37     "%.8f and its error is %e." % (a, error)
```

Gaussian Quadrature

```
1  """
2  Adapted from Mark Newman's code from the book "Computational Physics with Python"
3
4  PHYS440: Computational Physics
5  Project 2, Differentiation, Integration, and Root Finding
6
7  """
8
9  from numpy import ones, copy, cos, tan, pi, linspace, abs
10
11 def f(x):
12     return x**4
13
14 def gaussxw(N):
15     # Initial approximation to roots of the Legendre polynomial
16     a = linspace(3,4*N-1,N)/(4*N+2)
17     x = cos(pi*a+1/(8*N*N*tan(a)))
18
19     # Find roots using Newton's method
20     epsilon = 1e-15
21     delta = 1.0
22     while delta > epsilon:
23         p0 = ones(N,float)
24         p1 = copy(x)
25         for k in range(1,N):
26             p0,p1 = p1,((2*k+1)*x*p1-k*p0)/(k+1)
```

```
        dp = (N+1)*(p0-x*p1)/(1-x*x)
        dx = p1/dp
        x -= dx
        delta = max(abs(dx))
    # Calculate the weights
    w = 2*(N+1)*(N+1)/(N*N*(1-x*x)*dp*dp)
    return x,w

if __name__ == "__main__":

    N = 5
    a = 0.0
    b = 2.0

    # Calculate the sample points and weights, then map them
    # to the required integration domain
    x, w = gaussxw(N)
    xp = 0.5*(b-a)*x + 0.5*(b+a)
    wp = 0.5*(b-a)*w

    # Perform the integration
    s = 0.0
    for k in range(N):
        s += wp[k]*f(xp[k])

    error = abs(s - 2**5 / 5.)
    print "The value for the Gaussian Quadrature rule is %f and its error value"\
            "is %e" % (s, error)
```

Forward, Central, and Extrapolated Difference

```
"""
Written by Michael Lee

PHYS440: Computational Physics
Project 2, Differentiation, Integration, and Root Finding
"""

import numpy as np

def f(x):
    # We use f(x) as our test function to test our differentiation program.
    return x**3

def forward_difference(x, h):
    a = (f(x+h) - f(x)) / (h)
    forward_error = np.fabs(round(a) - a)
    return {'value': a, 'error': forward_error}

def central_difference(x, h):
    b = (f(x+h/2) - f(x-h/2)) / h
    central_error = np.fabs(round(b) - b)
    return {'value': b, 'error': central_error}

def extrapolated_difference(x, h):
    c = (4*((f(x+h/4) - f(x-h/4)) / (h/2)) - ((f(x+h/2) - f(x-h/2))/ h)) / 3.0
    extrapolated_error = np.fabs(round(c) - c)
    return {'value': c, 'error': extrapolated_error}
```

```
28
29 if __name__ == '__main__':
30
31     i = 4
32     h = 1e-10
33
34     test1 = forward_difference(i, h)
35     test2 = central_difference(i, h)
36     test3 = extrapolated_difference(i, h)
37     Actual_value = 3 * 4**2
38
39     print "The actual value is %f." % Actual_value
40     print "The value of the forward difference is %.8f and its error " \
41                                 "is %e." % (test1['value'], test1['error'])
42     print "The value of the central difference is %.8f and its error " \
43                                 "is %e." % (test2['value'], test2['error'])
44     print "The value of the extrapolated difference is %.8f and its " \
45                                 "error is %e." % (test3['value'], test3['error'])
```

Newton-Raphson Root Finding Method

```
1  """
2  Written by Michael Lee.
3
4  PHYS440: Computational Physics
5  Project 2, Differentiation, Integration, and Root Finding
6
7  """
8  import numpy as np
9
10 def f(x):
11     # Our test function
12     return 2*np.cos(x) - x
13
14 def newton(x, dx, imax, eps):
15     for i in range(0, imax + 1):
16         A = f(x)
17         if (abs(A) <= eps):
18             # Ends program if f(x) < eps
19             print "Root found, tolerance eps = ", eps
20             break
21         print "Iteration # = ", i, ", x = ", x, ", f(x) =", A
22         """
23         The next series of calulations finds the
24         new value x to recalculate f(x).
25         """
26         df = (f(x+dx/2) - f(x-dx/2)) / dx
27         dx = -A/df
28         x += dx
29     return x
30
31 if __name__ == '__main__':
32
33     test = newton(3, 1e-3, 100, 1e-6)
34     print "The root(s) are: %f" % test
```