

Numerical Analysis I – Project 2 – Extra Credit

For up to 25% extra credit (yes, 25!) you can prepare a mini-report analyzing and predicting the performance of your **Composite Trapezium** code (which I refer to as `comp_trap`) on the integrand $f(x)$ given in the main project.

In this part you'll look at an example of how to make predictions for the number of intervals needed to approximate an integral to high accuracy based on statistics of the method's performance at more modest accuracy. This can be a useful technique when the computational cost of a high accuracy computation is great, for instance if the integrand is very slow to compute. In such a case, its derivatives may be either very expensive to compute too, or they may not exist in explicit form. Either way, error analysis may be impractical or may generate predictions that are far too conservative (grossly over-estimating the necessary number of intervals for high accuracy is costly in this scenario).

We will work with an integrand that has explicit derivatives, is easy to integrate explicitly, and for which error analysis is easy, so that we can see a comparison of the techniques. We will also look at how to record the execution time of a calculation in python accurately. To avoid needing to use a CAS if you don't have access, the exact value of the integral you are calculating is 1.6840782256556266187 to 20 decimal places. You can perform the differentiation and analysis of $f(x)$ and its derivatives by hand and by graphing.

TASKS

(1) In addition to the usual recommended imports, import the `clock` function from the `time` library, and the `leastsq` function from `scipy.optimize`. Also, define two empty lists to record the statistics near the top of the file.

```
from time import clock
from scipy.optimize import leastsq

cost = []
errors = []
# error tolerances list
tolerances = [1e-2, 1e-3, 1e-4]
```

(2) For the above tolerances, use error analysis of the quadrature rule to estimate the minimum n that will achieve them. Then verify whether the quadrature

approximation using these n values indeed meet these tolerances. How far is the actual error from the desired tolerances?

(5 points)

(3) For each of the above tolerances find the *actual* minimum value of n that makes each the quadrature method accurate to those tolerances (using the known exact value of the integral). It doesn't have to be exact to the nearest integer, just get the error to round to the tolerance. [Hint: Save time searching blindly by using the fact that approximation error roughly divides by 4 every time you double n in order to get a good estimate for each n (based on a known error and n for the largest tolerance in the list).]

(4) How do your actual n values compare to the estimates from the error analysis?

(5 points)

(5) Define this list of the minimum n values for each of the above tolerances, in order:

```
n_vals = [ ] # << put values for each tolerance in here
```

Given an n value from the `n_vals` list, generate a number of repetitions, r , for each approximation, in the manner described below. Without many repetitions for the smallest n values, the execution time can be so small that the call to the clock function cannot register a meaningful value. Notice that we divide the elapsed time by r afterwards to get an average value for that n . Include this example for a single n value of 2000:

```
repetition_factor = 20000 # This value works well
r = int(ceil(repetition_factor/n))
t0=clock()
for i in xrange(r):
    good_approx = comp_trap(f, a, b, 2000)
t_elapsed = (clock()-t0)/r
```

Compare this "good approximation" to the exact value. It should be accurate to about $1e-5$, but we'll pretend we don't know this.

(6) For each n in your list `n_vals`, call the quadrature function on f and measure the elapsed time for the appropriate number of repetitions. Preferably, you should do this using a *for* loop. After each call, record the approximation error compared to the "good approximation" (our technique is not supposed to depend on the exact value) and the average elapsed time by appending them to the appropriate list.

(5 points)

(7) Plot the approximate errors on the x -axis vs. the cost on the y -axis using a log-log plot. Take the \log_{10} yourself of the lists after converting them to arrays, as in `log_cost = log10(array(cost))` etc.

(8) Using the techniques of Ch. 8 find an appropriate linear fit for this log-log data. You don't need to write code to do this, just use the `leastsq` function we already imported, as shown below. This uses an iterative form of the least squares method you learned in class.

```
def fitfunc(p, x):
    return p[0] + p[1] * x

def errfunc(p, x, y):
    return y - fitfunc(p, x)

pinit = [1.0, -1.0] # any initial values will do
out = leastsq(errfunc, pinit,
              args=(log_error, log_cost), full_output=1)

intercept, slope = out[0] # unpack values from the array
covar = out[1] # we don't care about this covariance
print "Slope is", slope, "Intercept is", intercept

plot( [-6, 0], [fitfunc(out[0], -6), intercept], 'k-')
```

Drawing this line on your log-log plot you should see that it seems reasonable. [Note: If you repeated this on the integrand $g(x)$ you would expect a very similar linear slope, although a different y -axis intercept.] You should get a slope of around -0.5.

(5 points)

(9) Use this fit to generate the nonlinear equation for the relationship between error tolerance and expected execution time on your computer.

```
def time_predict(target_tol):  
    return 10**(intercept)*target_tol**(slope)
```

(10) Use this function to predict how long it will take your computer to calculate the integral correct to 7, 9 and 11 decimal places (that's up to *six* orders of magnitude more accurate than the "good approximation"). Compare the accuracy of these predicted times by direct calculation of the quadrature approximation to these tolerances (as you did in Q3). Are they excellent, reasonably good, or bad?

(5 points)

Final notes

If you try to apply this method beyond 10 or 11 decimal places for the Composite Trapezium or even Composite Simpson's methods, accumulated rounding error and memory management issues within python may begin to distort your predictions, especially regarding execution time. Also, small errors in the fitting of your straight line will be magnified the further away from your data you try to extrapolate the line to, making the predictions increasingly unreliable. Remember that any errors are magnified through the exponential function when you use them to predict actual times!

This project is just a simplistic and naive exploration of some of the issues involved. More sophisticated modeling of the behavior of your computer would be needed in practice. Also, one would normally use methods that are inherently more accurate to extrapolate to high accuracy for smaller n values e.g., adaptive methods and Gaussian quadrature methods.