HW 11

Derek Walker

1:

(a):

```
def eval_comp_trap(a, b, f, N):
    h = (b - a) / N
    xi = np.linspace(a, b, N+1)
    Ihat = (h/2)*(f(a) + 2*np.sum(f(xi)) + f(b))
    return Ihat

def eval_comp_simpsons(a, b, f, N):
    h = (b - a) / N
    x = np.linspace(a, b, N+1)
    y = f(x)
    Ihat = (h/3)*(y[0] + 4*np.sum(y[1:-1:2]) + 2*np.sum(y[2:-2:2]) + y[-1])
    return Ihat
```

I wrote these functions to eval comp trapezoidal and comp simpsons quadrature rules. I used N = 50 for each initial evaluation. The results are listed below:

```
Eval Composite Trap: 2.7620875579289037
Eval Composite Simpsons: 2.7468017380097285
```

This is fairly accurate as the actual value of the integral is: 2.746801533890032.

It seems that simpson's rule is more accurate and efficient than trap.

(b):

I used the error terms:

$$-\frac{(b-a)h^2 f''(\mu)}{12}$$ and $$-\frac{b-a}{180} h^4 f^{(4)}(\mu)$$

from class for comp trap and comp simpsons respectively. I then solved for h. The code looked like:

```
def choose_n_by_trap_error(a, b, f2p, tol):
    eta = find_eta(a, b, f2p, tol)
    if np.isnan(eta): return -1
    h = (np.abs(tol*12/((b-a)*f2p(eta))))**(1/2)
    return int((b-a) / h)

def choose_n_by_simpsons_error(a, b, f4p, tol):
    eta = find_eta(a, b, f4p, tol)
    if np.isnan(eta): return -1
    h = (np.abs(tol*180/((b-a)*f4p(eta))))**(1/4)
    return int((b-a) / h)
```

Where find_eta looks for the largest value of the function it's given within a and b according to the given tolerance:

```
def find_eta(a, b, f, tol):
    new_eta = a
    x = np.arange(a, b, tol)
    for eta in x:
        if np.abs(f(eta)) > np.abs(f(new_eta)):
            new_eta = eta
    return new_eta
```

f2p and f4p were calculated by hand like such:

```
a = -5
b = 5
tol = 1e-4
f = lambda s: 1/(1+s**2)
f2p = lambda s: 2*(3*s**2 - 1)/(1+s**2)**3
f4p = lambda s: 24*(5*s**4 - 10*s**2 + 1)/(1+s**2)**5
```

For the number of nodes required, I got:

```
N for trap (0.0001): 912
N for simpsons (0.0001): 48
```

And the results I got were:

```
Eval Composite Trap: 2.7476446922577815
Eval Composite Simpsons: 2.7468008110222586
```

These are not accurate to the given error of 1e-4.

(c):

In running scipy.integrate.quad, I get much more accurate values if I input a tolerance of 1e-4 and 1e-6 respectively:

```
SciPy Quad (0.0001): I=2.746801533909586, nevals=63
SciPy Quad (1e-06): I=2.7468015338900327, nevals=147
```

This is much better than my code, so the number of nodes calculated by a given error tolerance is probably not completely accurate.

The number of evals seem about right for the simpson's rule but way lower than the number of evals required for trap.

2:

The result I get is: `Eval Composite Simpsons (5 nodes): -0.00689462356222182`

If I put 50 nodes, I get a much more accurate result so I believe I'm doing this right.
I did the simple change of variables and ended up with asymptotic limits of a = 0 and b = 1 with an integrand of cos(1/t)*t. I simply put that cos(1/0)*0 = 0 because that's what the limit would evaluate to in my opinion. It then seemed to work.

3:

In general, if you want two error terms to be gone, you need three different versions of the same integral in order to perform Richardson Extrapolation on them. Essentially, the following equations must hold:

$$I = aI_n + bI_{n/2} + cI_{n/4} + O(n^{-5/2})$$
$$a + b + c = 1$$

Since $I_n + O(n^{-3/2}) = I_{n/2} + O(n^{-3/2}) = I_{n/4} + O(n^{-3/2})$, you need all of the coefficients to be added up to unity simply because you want exactly one approximation. The idea behind Richardson Extrapolation is to extrapolate the error to higher orders by removing error terms. Hence, it decreases the error in the approximation. You can do this as many times as you'd like.

Let's look at each of these approximations separately:

$$I \approx I_n + \frac{C_1}{n^{3/2}} + \frac{C_2}{n^2} + \frac{C_3}{n^{5/2}}$$
$$I \approx I_{n/2} + \frac{2^{3/2}C_1}{n\sqrt{n}} + \frac{2^2C_2}{n^2} + \frac{2^{5/2}C_3}{n^{5/2}}$$
$$I \approx I_{n/2} + \frac{2^{3/2}C_1}{n\sqrt{n}} + \frac{2^2C_2}{n^2} + \frac{2^{5/2}C_3}{n^{5/2}}$$

Since they're all equal, we can linearly superpose them and divide by the resulting coefficient to get the original integral but this time with less error terms.

Let's superpose them like in the first equation and solve for the coefficients by looking just at the error terms:

$$E = \frac{C_1(a+2^{3/2}b+4^{3/2}c)}{n^{3/2}} + \frac{C_2(a+2^2b+4^2c)}{n^2} + O(n^{-5/2})$$

Therefore, we now have three equations and three unknowns:

$$a + b + c = 1$$
$$a + 2^{3/2}b + 4^{3/2}c = 0$$
$$a + 2^2b + 4^2c = 0$$

The resulting coefficients can be computed like such:

$$A = [[1,1,1],[1,2\sqrt{2},8],[1,4,16]]$$
$$b = [1,0,0]$$
$$x = A^{-1}b$$

This was computed in python using the following code:

```
print("\nQuestion 3")
A = [[1,1,1],[1,2*np.sqrt(2),8],[1,4,16]]
b = [1,0,0]

x = np.linalg.solve(A, b)
print("x: ", x)
```

```
Question 3
x:  [ 2.06255755 -1.2448636   0.18230605]
```