**Phys 512 Problem Set 1**

Due on github Friday Sep 12 at 11:59 PM. You may discuss problems, but everyone must write their own code.

**Problem 0:** Please hand-write the numbers 0 through 9, scan, and compress each down to a 28x28 array of pixel brightnesses that go from 0 to 255. Put into a single array of size (10,28,28) and save as 'digits_[your student id]_week1.npy'. You can get a half bonus point if you write out 5 sets and save as a (50,28,28) with the numbers ordered (0..9,0..9,0..9), and a full bonus point if you write out ten sets.

**Problem 1:** We saw in class how Taylor series/roundoff errors fight against each other when deciding how big a step size to use when calculating numerical derivatives. If we allow ourselves to evaluate our function $f$ at four points ($x \pm \delta$ and $x \pm 2\delta$),
a) what should our estimate of the first derivative at $x$ be? Rather than doing a complicated fit, I suggest thinking about how to combine the derivative from $x \pm \delta$ with the derivative from $x \pm 2\delta$ to cancel the next term in the Taylor series.
b) Now that you have your operator for the derivative, what should $\delta$ be in terms of the machine precision and various properties of the function? Show for $f(x) = \exp(x)$ and $f(x) = \exp(0.01x)$ that your estimate of the optimal $\delta$ is at least roughly correct.

**Problem 2:** Write a numerical differentiator with prototype

```
def ndiff(fun,x,full=False):
```

where fun is a function and x is a value. If full is set to False, ndiff should return the numerical derivative at x. If full is True, it should return the derivative, dx, and an estimate of the error on the derivative. I suggest you use the centered derivative

$$f' \simeq \frac{f(x+dx) - f(x-dx)}{2dx}$$

Your routine should estimate the optimal dx then use that in calculating the derivative. If you're feeling ambitious, write your code so that x can be an array, not just a single number. If you do that, you may actually wish to save your code as you might use it in the future.

**Problem 3:** We saw in class that Simpson's rule required us to use an odd number of points, and we got very wrong answers if we used an even number of points. One way we could fix this for an even number of points is to use regular Simpson's rule for the first $n-1$ points, then fit a parabola to the last 3 points, and use the area of that parabola only for the *last* interval.
Part a) For simplicity, assume the last three points have $x-$coordinates $-1, 0, 1$, and that they have $y-$values of $y_l, y_0, y_r$. If we have $y = ax^2 + bx + c$, what are $a, b, c$ in terms of $y_l, y_0, y_r$?

Part b) What is the area of that parabola from 0 to 1, again in terms of coefficients times $y_l, y_0, y_r$?
Part c) Write an integration routine with prototype

```
def flexsimp(y,dx):
```

that returns the numerical integral of $y$ for abritrary length of $y$. If the length of $y$ is odd, it should return Simpson's rule. If the length of $y$ is even, it should return Simpson's rule for $n-1$ points, and add your correction from part b). I encourage you to use recursion here, where for even numbers of points, flexsimp calls flexsimp for an odd number of points, and then adds the correction.
Part d) Evaluate the numerical integral of $\exp(x)$ from $(-1, 1)$ with your integrator for 20 points and for 40 points. Did the error in your answer decrease as expected?
Bonus: Repeat the above exercise, but now use the first interval instead of the last for your parabolic fit. Both answers should be correct to the same order, so the average should be correct to the same order as well. In the average, the [2,4,2,4...] pattern in the interior for one answer will be matched by a [4,2,4,2...] pattern in the other, which means the interior reduces to just the sum of the $y_i$. Write an integrator that takes advantage of this by finding weights for the first/last 3 $y_i$ and gives you $4^{th}$-order accuracy for any length $y$ without using any if-statements. What are the weights for the first/last 3 points?

You could continue this procedure indefinitely, where any high-order accuracy integrator can be expressed as a set of weights times the end regions plus the unweighted sum of the interior. That means that *all* of the information about higher-order behavior of the integral of an arbitrarily large region resides in the behavior of the function over a finite region at the edges of the interval.

**Problem 4:** Take $\cos(x)$ between $-\pi/2$ and $\pi/2$. Compare the accuracy of polynomial, cubic spline, and rational function interpolation given some modest number of points, but for fairness each method should use the same points. Now try using a Lorentzian $1/(1+x^2)$ between -1 and 1.

What should the error be for the Lorentzian from the rational function fit? Does what you got agree with your expectations when the order is higher (say n=4, m=5)? What happens if you switch from np.linalg.inv to np.linalg.pinv (which tries to deal with singular matrices)? Can you understand what has happened by looking at $p$ and $q$? As a hint, think about why we had to fix the constant term in the denominator, and how that might generalize.