Numerical Methods II, Courant Institute, Spring 2016

http://www.math.nyu.edu/faculty/goodman/teaching/NumMethII2016/index.html

Always check the `classes` message board before doing any work on the assignment.

Assignment 2, due February ??

Corrections: (none yet)

1. (*An exercise in Taylor series order of accuracy calculation*) Many computations of fluid flows use *cell averages* rather than *point values*. In one dimension, a *cell* is the interval of length Δx about the grid point x_j. That is

$$C_j = [x_j - \tfrac{1}{2}\Delta x, x_j + \tfrac{1}{2}\Delta x] \; .$$

   Suppose the discrete variable U_j is the cell average:

$$U_j = \frac{1}{\Delta x} \int_{x_j - \frac{1}{2}}^{x_j - \frac{1}{2}} u(x)\, dx \; .$$

   (a) Show that the cell average is a second order accurate approximation to the point value in the cell center. That is:

$$| \, u(x_j) - U_j | \leq C \Delta x^2 \; .$$

   (b) Find a fourth order "reconstruction" of the midpoint values from the cell averages. That means, find values a_{-1}, a_0, a_{+1} and a power p so that

$$u(x_j) = U_j + (\Delta x)^p \left(a_{-1} U_{j-1} + a_0 U_j + a_{+1} U_{j+1} \right) + O(\Delta x^4) \; .$$

2. (*Serious von Neumann calculation*) This exercise calls on you to construct a fourth order explicit time stepping method for the heat equation following a derivation of Lax and Wendroff. The scheme has a five point stencil and finite difference coefficients that are not easy to guess. The method has a time step constraint $\Delta t \leq \lambda D \Delta x^2$. We will see how to find λ.

   (a) "Review" *Richardson extrapolation*[1]. Use Richardson extrapolation to find a fourth order approximation to the second derivative of the form

$$f''(x) \approx \frac{1}{\Delta x^2} \left(b_{-2} f(x - 2\Delta x) + b_{-1} f(x - \Delta x) + b_0 f(x) + b_1 f(x + \Delta x) + b_2 f(x + 2\Delta x) \right)$$

   The error should be order Δx^4.

[1] One source is the beautiful book by Dahlquist and Björk. Another is the notes by David Bindel and Jonathan Goodman linked from the `Resources` page of the class web site.

(b) Write a simply `Python` script to try your formula on $f(x) = e^{2x}$ and $x = 0$. Get a numerical demonstration of fourth order accuracy (not just high accuracy).

(c) Find a second order accurate finite difference approximation to the fourth derivative:

$$f'''' \approx \frac{1}{\Delta x^4} \left(c_{-2} f(x - 2\Delta x) + c_{-1} f(x - \Delta x) + c_0 f(x) + c_1 f(x + \Delta x) + c_2 f(x + 2\Delta x) \right)$$

A simple way to do this is to take the second difference of the second difference, which is a discrete version or taking the second derivative of the second derivative. The numbers should come from row 4 of Pascal's triangle, but with alternating signs[2]: $1, -4, 6, -4, 1$.

(d) Suppose that $u(x, t)$ is a smooth function that satisfies the heat/diffusion equation

$$\partial_t u = D \partial_x^2 u \ .$$

Show that the following is a second order approximation (this is the idea of Lax and Wendroff):

$$u(x, t + \Delta t) \approx u(x, t) + \Delta t D \partial_x^2 u(x, t) + \frac{\Delta t^2 D^2}{2} \partial_x^4 u(x, t) \ .$$

(e) Assume that $\Delta t = \frac{\lambda}{D} \Delta x^2$, where λ is the *dimensionless CFL ratio*. Use the fourth order approximation to the second $x-$derivative and the second order approximation to the fourth $x-$derivative to get a scheme of the form

$$U_{j,k+1} = a_{-2} U_{j-2,k} + a_{-1} U_{j-1,k} + a_0 U_{j,k} + a_1 U_{j+1,k} + a_2 U_{j+2,k}$$

The five coefficients (really just three because of repeats) all depend on λ but otherwise are independent of Δx, Δt, and D. Show that the method is formally fourth order accurate, which means that the residual has the form $\Delta t R_{jk}$, where $R = O(\Delta x^4)$. Hint: This could be hours of repetitive and dull calculation. You will make much less work for yourself if you "show" that the residual is fourth order without calculating an explicit formula for the residual. Mathematical reasoning will make this problem much easier than brute force calculus.

(f) The *symbol* of the finite difference scheme is

$$m(\theta, \lambda) = a_{-2} e^{-2i\theta} + a_{-1} e^{-2i\theta} + a_0 + a_1 e^{i\theta} + a_2 e^{2i\theta} \ .$$

It depends on λ because the coefficients depend on λ. Show that $m(\theta)$ is real, for real θ. Show that if λ is small enough, then $|m| \leq 1$ for all θ. This shows that the scheme is stable for sufficiently small CFL ratio.

[2]The book of Dahlquist and Björk has a beautiful discussion of the *calculus of finite difference operators*.

(g) (*extra credit, attempt only after the rest is finished*) Find the maximum value of λ for which the scheme is stable.

3. This exercise works through some properties of Fourier analysis. You will know that the algebra is right when the corresponding Python computation agrees with theory. The script `FourierAnalysis.py` will help you get started. The first part of the script illustrates the difference, in Python, between *assignment* and *deep copy*, see below. The second part of the script illustrates the Plancharel formula for the DFT. Note the factor of $n = 6$, which is as it should be. Always use the built-in Python FFT routines, except in part (3a).

   (a) Write a Python script that calculates the DFT sums by direct loops something like (but not exactly)

```
uHat = np.zeros(n)
for m in range(n):
    for k in range(n):
        uHat[m] += np.exp( 2*pi*i*m*j/n)*u[j]
```

   Compare the running speed of this to the built-in FFT code `np.fft.fft`. Describe n values for which one "works" and the other doesn't. You can do the timings informally.

   (b) The *left shift* operator S "operates" on vectors $U \in \mathbb{C}^N$ by *circular* shifting down one index. If $V = SU$, then

$$V_j = U_{j+1} \ , \quad 0 \leq j \leq N-2 \ , \quad V_{N-1} = U_0 \ .$$

   Show that $\widehat{V}_m = s(m)\widehat{U}(m)$ and find a formula for $s(m)$. The numbers $s(m)$ are the eigenvalues of the matrix S. Verify this formula in Python.

   (c) The function

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2}{2\sigma^2}} \ ,$$

   is the probability density of a Gaussian random variable. The Fourier transform of this function (which is called the *characteristic function* in probability) is

$$\widehat{f}(k) = \int_{-\infty}^{\infty} e^{-ikx} f(x)\, dx = \frac{1}{\sqrt{2\pi\sigma^2}} \int_{-\infty}^{\infty} e^{-ikx} e^{-\frac{x^2}{2\sigma^2}}\, dx = e^{-\frac{\sigma^2}{2}} \ .$$

   Approximate the Fourier transform by

$$\widetilde{f}(k) = \Delta x \sum_{|x_j| \leq R} e^{-ikx_j} f(x_j) \ .$$

   There are two approximations, restricting the range of integration and using the trapezoid rule for the integral. Of course, $x_j = j\Delta x$,

but now j can be negative. The FFT will compute the values $\widetilde{f}$ at $n = \frac{2R}{\Delta x} + 1$ points. Because of aliasing, you can find values of $\widetilde{f}(k)$ for a range of k values that is symmetric around $k = 0$. Make a plot of $\widetilde{f}$ in a symmetric part of this range where $\widetilde{f}$ is significantly different from zero. You can modify code from `HarmonicAnalysis.py`. The hard part is scaling – mapping this continuous problem onto the algebra problem where m runs from 0 to $n - 1$. Try with various values of σ, R, and Δx. If the Gaussian is "well resolved", then $\widetilde{f}$ should be an accurate approximation to $\widehat{f}$. "Well resolved" means that Δx is significantly smaller than σ and R is significantly larger than σ. Hand in a few pictures to illustrate what can happen. Make sure to modify the title of the plot to contain the parameters.

Assignment and deep copy: Python has *names* and *objects*. Names are *bound* to objects (or is it objects bound to names?). For example, `x = [2.71,3.14]` creates a name `x` and an object, which is the list of two floating point numbers $[2.71, 3.14]$. If the next line is `y=x`, Python creates a new name, *but not a new object*. Instead, the name `y` is bound to the same object $[2.71, 3.14]$; `x` and `y` are just different names (aliases) for the same object. If the next line is `x[1] = 1.41`, then the object becomes $[2.71, 1.41]$. Both names `x` and `y` are still bound to it. This means that if you print `y[1]` you will get 1.41, not 3.14. Every Python programmer I know has spent days finding a bug due to this. If you want `y` to be bound to a different object that has the same value as `x`, you have to do that explicitly in some way. The script posted does it using `deepcopy`. The statement `y=x` is a *shallow copy*.

The above applies only to *mutable* objects. Roughly speaking, mutable objects have structure – lists, vectors, etc. *Immutable* objects are the simple ones – integers, floating point numbers, character strings. If `x` is bound to an immutable object, then `y=x` creates a new object with the same value as `x` and binds it to the name `y`. This is a deep copy. The example above involved a list, which is mutable. The lines:

```
x = 2.71
y = x
x = 1.41
print y
```

will give 2.71.

Other languages talk about *side effects* of a statement; that happen when the statement is executed that are not obvious from looking at the statement itself. Changing a mutable object can have side effects, if there are other names bound to the same object. Changing an immutable object cannot have such effects.

The terminology immutable/mutable comes from the way assignments work in Python. The statement `x =` (immutable v_1) will create an object o_1 with that value and bind the name `x` to it. If `x` had been bound to a different object,

o_0, then this re-binding does not change o_0, just "forgets" it. Python creates the new object o_1 rather than changing o_0 because o_0 was immutable.