Due Friday, November 13, 24:00

Problem 11 (6 points)

In this problem we will try to learn about the cosmological history of the universe by analyzing supernova data. It is based on this project from Princeton's Physics 209, feel free to read it for more details about physics and many computational hints, but the explanations here are self contained, albeit shorter.

Electromagnetic signals from far away points in an expanding universe are observed to be in longer wavelengths than when they leave their source (cosmological redshift) which can naively be attributed to the fact that the source is moving away from us leading to Doppler shift.[1] If we can measure this change in wavelength as a function of the distance of the source, we can actually learn about how fast the universe has been expanding in the past. This, in turn, is governed by the matter-energy content of the universe, so ultimately we can relate the redshift of cosmological light sources to how much and what kind of energy there is in the universe. Without technical details, the relationship is as follows

$$d_L(z) = (1+z) d_H \overbrace{\int_0^z \frac{dz'}{\sqrt{\Omega_M (1+z')^3 + (1-\Omega_M)}}}^{I(z,\Omega_M)} .$$

Here, d_L is the so-called luminosity distance of the source, $z = \frac{\lambda_{src} - \lambda_{obs}}{\lambda_{obs}}$ is the redshift parameter calculated from the wavelengths at the source and at the observer, d_H is a constant called Hubble length and $0 < \Omega_M < 1$ is a normalized form of matter density in the universe.

(a) You can find the Hubble Space Telescope data for $d_L(z)$ here (or on Blackboard). The columns are z, d_L and the error in the measurement of d_L from left to right. d_L is measured in the common cosmological length unit megaparsecs (Mpc), we will use it in this question. Write a Python function `read_hubble_data(filename)` which reads the above file and returns the three columns as three arrays

```
import numpy as np
def read_hubble_data(filename):
    datafile = open('hubble_low_z.dat', 'r')  # 2nd argument says open for reading
    for line in datafile:            # Iterating over a file goes line-by-line
        (a, b, c) = line.split()         # split() breaks text at whitespace

        # Somehow have to keep a record of a, b, c???
        # Arrays can't be built element-by-element.
        # But lists can...

    datafile.close()                 # Close the data file
```

You can find more help in Physics 209 website. Plot d_L vs z *together with the error bars* (it should go without saying that this plot should appear in your `ps0412.py` file). This can be simply done with `matplotlib.pyplot.errorbar`.[2]

(b) Find d_H using a linear fit of the form $d_L(z) = az + b$ to the data for $z < 0.1$. Is the constant term b around what you would expect it to be? Plot the best fit line together with the data.

(c) For a better analysis we should use the full non-linear equation $d_L(z, d_H, \Omega_M)$ with two fitting parameters d_H and Ω_M. First, write a function which calculates $I(z, \Omega_M)$ as

```
import numpy as np
from scipy.integrate import quad
# calculates the integral expression in cosmological luminosity distance
# \int_0^z dy/(E(y)
def hubble_integral(z, omega_m):
```

[1] Understanding the *real* reason requires general relativity.

[2] Physics 209 suggests using the `pylab` package, but internet says its usage is discouraged these days, so stick with `matplotlib.pyplot`.

We have not learned how to take integrals numerically, but for our purposes it is enough to know that `quad(fun,a,b)` integrates the function `fun` from `a` to `b`. You will also need $I_2 \equiv \partial I/\partial\Omega_M$ which can be implemented similarly to I using `quad`.

Finding the best non-linear fit is equivalent to setting up a nonlinear root finding problem with the above functions (Eq. 3.1.22 in the textbook). There is enough numerics in this problem, so no need to implement this ourselves, use `scipy.optimize.root` to find the best fit values of d_L and Ω_M. A rough guess about the solution is very helpful in nonlinear root finding, so make sure you use the rough value of d_L from the linear fit and the fact that $0 < \Omega_M < 1$.

(d) We did not use our measurements errors, the third column of the data, in our fits. The error can best be understood as the fact that at each given z what we can measure is not *the* value of the luminosity distance, but rather a Gaussian probability distribution[3] for it with the mean given by the second column of the data, and standard deviation by the third column (the error). For example

$$Probability[x < d_L(z = 0.101) < x + dx] = \frac{1}{\sqrt{2\pi \times 45.27^2}} e^{(x-427.89)^2/(2\times 45.27^2)}$$

IMPORTANT: The error bars in the actual data might be quite high which may cause your nonlinear fits to often fail. If that happens, you are allowed to cheat, and use a quarter of the error values in the file.

This immediately gives an idea about estimating the error in our fit parameters due to the error in measurements. Randomly pick a value for $d_L(z)$ from its probability distribution for each z in our data set, and perform a nonlinear fit as above. You can obtain such probability distributions using

```
sigma * np.random.randn(...) + mu
```

Repeat this for many such random picks, and plot the distribution of d_L and Ω_M values you obtain from each fit using a histogram. You will see that the parameters will also have a Gaussian distribution more or less. Calculate the mean and the standard deviation for each parameter which are the central values and the uncertainities in $d_H \pm \Delta d_H$ and $\Omega_M \pm \Delta\Omega_M$.

There is actually a less painful way of estimating $\Delta d_H, \Delta\Omega_M$ in simple cases which we will learn later.

(e) We have done something I advised you not to do: we implemented our own nonlinear least squares solvers even though SciPy has its own: `scipy.optimize.curve_fit`. Try to repeat parts (c) and (d) but this time directly using `scipy.optimize.curve_fit` rather than performing root finding. This function also gives you estimates of the errors in the fitting parameters, so no need for separate error calculations is needed, unlike part (d) (see the **pcov** output in the documentation). How do the results compare to your custom code?

If everything goes as planned, you will find that $\Omega_M \sim 0.4$ (the correct value is $\Omega_M \sim 0.3$, but this is old data and we ignore certain points). This means more than half of the energy in the universe behaves like the energy density of vacuum. This is the famous dark energy that was thought to be nonexistent until this data came out in 1998. Physicists who obtained the data and interpreted it this way shared the Nobel Prize in 2011. This also seems to imply that our universe will expand forever in an exponentially accelerating manner. There is even more to the story in relation to the curvature of the universe which we assumed to be flat here. The universe is indeed flat as shown by the same data (which is discovered by a more general fit that we did not attempt), and basic considerations suggest that it should be very very curved unless something strange happened shortly after the big bang.

[3] In reality the error does not have to be Gaussian, but we will assume this simplest case here

Problem 12 (6 points)

We will use Runge's function

$$R(x) = \frac{1}{1+25x^2}$$

in the interval $[-1\ 1]$ to demonstrate the ideas about interpolation we discussed in class. Before we start, implement this function as `runges_function(x)` (you can add it to `customfunctions.py` if you want, so all the functions we have been implementing ourselves are in the same place).

(a) Pick $n = 4, 8, 16, 32, 64$ equidistant points in $[-1\ 1]$ and make an exact polynomial fit, R_n, to values of R at these points. Plot R and all your fits on the same plot (you should plot on the whole range, not just the interpolation points). Do you get better fits with more points and higher degrees of poynomials? What you see is called Runge's phenomenon. You can use the builtin `numpy.polynomial.polynomial.polyfit` for the polynomial fit and `numpy.polynomial.polynomial.polyval` to evaluate the polynomials for given coefficients.

To quantify how good the interpolation function is, we can generalize the Frobenius norm to continuous functions as

$$E(n) \equiv \left[\frac{1}{2}\int_{-1}^{1} |R(x) - R_n(x)|^2\right]^{1/2} .$$

Plot $E(n)$ vs n, and interpret the results. You can use `scipy.integrate.quad` to take the integral.

(b) Now repeat the previous part with splines. SciPy has `scipy.interpolate.spline`, for splines of any degree, but for general purposes `scipy.interpolate.interp1d` is more versatile. Use it with $kind = 'linear'$ (default) and $kind = 'cubic'$ options, which are the linear and cubic splines (cubic being the common choice). Comment on the results.

(c) You should not completely disregard Lagrange polynomial fits due to Runge's phenomenon, part of the blame is on our insistence to keep equidistant sampling points. Repeat part (a), but this time interpolate the function based on the n points at $x_j = \cos\left(\frac{2j+1}{2n}\pi\right)$ for $j = 0, 1, \ldots, n-1$. Comment on the results. These are called the Chebyshev nodes, and they are related to the Chebyshev polynomials. Geometrically, these nodes are equivalent to picking equidistant points on a unit circle and then projecting them to the x axis. They are not the unique choice, other interpolation points distributed with density $(1-x^2)^{-1/2}$ near the edges can also rectify the shortcomings of naive polynomial interpolation.

Be wary of this last approach unless you really learn its details. There are numerical sources of instability even when things are fine at the analytical level. The following is from the documentation of `numpy.polynomial.polynomial.polyfit`: *Polynomial fits using double precision tend to "fail" at about (polynomial) degree* 20. *Fits using Chebyshev or Legendre series are generally better conditioned, but much can still depend on the distribution of the sample points and the smoothness of the data. If the quality of the fit is inadequate, splines may be a good alternative.*

(d) Lastly, instead of a perfect fit at the n given points, consider a linear least squares fit with x^m for $m \leq \sqrt{n}$. This fit will not pass through all n sample points, but it might still improve the overall fit as measured by $E(n)$ above. Repeat the steps of part (a) with this method for $n = 16, 64, 256, 1024$, and comment on the results.[4]

[4] You can read about this and other methods to avoid Runge's phenomenon at Wikipedia and more detailed references therein.

Optional for those who enjoy coding (no points, bonus or otherwise)

Other than reinventing the wheel, we made another very bad choice in Problem 11: we tried to solve the minimization problem by turning it into a root finding problem using partial derivatives. Remember that there are already specialized algorithms that solve the minimization problem directly (remember gradient descent?), and they are usually much more effective (at worst, they can go back to root finding for the partial derivatives). `scipy.optimize.curve_fit` uses such algorithms, and you can choose between a few. Just as in linear least squares fitting, there are other minimization functions in Python, for example `scipy.optimize.least_squares`. does a similar thing, and I am not sure of the difference, but `scipy.optimize.least_squares` documentation says "textttscipy.optimize.curve_fit [:] Least-squares minimization applied to a curve-fitting problem." You can try and see if this functions makes anything better or worse. Also, you can try changing the method utilized in these fitting functions to see if any of them perform better in terms of accuracy or speed.

For even more specific optimization problems, you can probably find specialized modules that are faster within a small spectrum. However, in any case, know that optimization (i.e. minimization) problems can be very tricky in real life research. They may end up giving you a local minimum rather than the global one, or not find any good solution at all. This is why optimization is a huge sub-field of applied mathematics on its own. Try to provide everything you know to the fitting functions, e.g. any known bounds on the parameters to be fit.