Tutorial 1+2

The core exercises are considered part of the course material, and you are advised to at least finish these. For additional practice and interesting applications, advanced exercises are available as well – feel free to pick and choose from these.

Unless otherwise specified, we expect you to create your own algorithms for the exercises in each tutorial. This means that you should **not use libraries (e.g. `SciPy`, `numpy`)** for the methods that are part of the course material.

You are free to write and execute code on your own laptop, but please be aware that all code you hand in needs to run on the vdesk machines[1] out of the box (e.g. no installing additional packages), so you may want to get used to this already.

Core exercises

1. **Numerical errors**

   Let's look at the (unnormalised) sinc function $\mathrm{sinc}(x)$, defined as

   $$\mathrm{sinc}(x) = \frac{\sin(x)}{x}. \tag{1}$$

   Its power series expansion around zero is given by

   $$\mathrm{sinc}(x) = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n}}{(2n+1)!} = 1 - \frac{x^2}{3!} + \frac{x^4}{5!} - \frac{x^6}{7!} + \cdots, \tag{2}$$

   where '!' denotes the factorial function.

   (a) Write a piece of code that implements the $\mathrm{sinc}(x)$ function in two ways: one using the power series expansion, and one using a library function, e.g. np.sin(x). Clearly, the accuracy of the answer given by the power series expansion depends on the number of terms included. Which kind of error are we dealing with in such an example?

   (b) Compare the two implementations for a different number of terms. Let's look at the value of sinc(7). How does the error change when more terms are added? Does the error accumulate preferentially in one direction, or does it oscillate?

2. **A short timing test**

   Suppose that you need to calculate the Schwarzschild radius R_S of a distribution of black holes of mass $M_\bullet$. R_S is given by

   $$R_S = \frac{2GM_\bullet}{c^2}, \tag{3}$$

   where G is Newton's gravitational constant and c is the speed of light.

   (a) Generate the masses of 10000 supermassive black holes by drawing from a Gaussian distribution with mean 10^6 $\mathrm{M}_\odot$ and standard deviation 10^5 $\mathrm{M}_\odot$ (you can use e.g. np.random.normal).

   (b) Using the `timeit` module, calculate the Schwarzchild radius of the black holes directly as given by equation 3.

   (c) Calculate R_S and use `timeit` again, but this time instead of dividing by c^2, multiply by a predefined variable `c_inv2 = c_inv * c_inv`, where `c_inv = 1/c`. Compare the results.

[1] `https://helpdesk.strw.leidenuniv.nl/wiki/doku.php?id=manuals:virtualdesktopserver`

3. **Interpolation**
An application of interpolation is image resizing (*rebinning*). When expanding the size of an image, the values for new pixels can be obtained by interpolating the values of the original image in 2-steps (such as bilinear or bicubic interpolation). For this exercise, take `https://home.strw.leidenuniv.nl/~daalen/NURfiles/M42_128.jpg`. Read the image using `from matplotlib.image import imread`, followed by `image=imread(''M42_128.jpg'')`, which will load in the image as a numpy array with pixel values.

   (a) Write a linear interpolator and apply it to the first row of the image for 201 equally spaced points. Implement bisection to find the enclosing two grid points for each x you want to interpolate at. Plot both the measurements from the image and the interpolated results.

   (b) Now write a polynomial interpolator (Neville's algorithm) and overplot these results as well.

   (c) Expand each of the dimensions of the image by a factor 2 using any two of the three interpolation methods you have developed in the previous question as your base for 2-step interpolation – or, use true bilinear interpolation as shown in the lecture. Compare the times the different methods took (you may have to run them many times in a row to get a meaningful run time). Which one is better and why? Can you suggest any other method to enlarge an image? Is the total flux density conserved? How can you ensure flux density conservation?

Advanced exercises

4. **Compute the area of a polygon**
Suppose we have some polygon with vertices specified by the coordinates (x_0, y_0), (x_1, y_1),...,(x_{n-1}, y_{n-1}), numbered either in a clockwise or counter-clockwise fashion around the polygon. If the polygon is non-self-intersecting (i.e. it is simple, does not intersect itself, it has no holes, etc), the area of the polygon can be computed by just knowing the boundary coordinates:

$$A = \frac{1}{2}\left|\sum_{i=0}^{n-1}(x_i y_{i+1} - x_{i+1} y_i)\right| \tag{4}$$

   (a) Write a function `polygon_area(x, y)` that takes two coordinate lists with the vertices as arguments and returns the area. For this function, use for loops to operate over each entry in the array every time. Test the function on a triangle and a pentagon where you can calculate the area by alternative methods for comparison. Convince yourself that your function returns the correct area of some regular pentagons.

   (b) Vectorize equation (4) such that there are no Python loops in the implementation. Make a test function that compares the scalar implementation with the new vectorized implementation for some chosen polygons. *Hint: `numpy.dot` could be useful to vectorize the code.*

   (c) Use timeit to compute how much time your code takes to compute all the areas using the vectorized implementation versus the for-loop implementation. Vary the number of vertices for your tests here like you did in (a), using from 3 up to 20. When is the vectorized implementation faster? Is it always faster? If the differences are not immediately apparent, test both approaches on a large number of polygons, e.g. 10 000.

5. **Binary operators**
For this exercise, it is convenient to visualize the input and results both in decimal and binary notation. In `python` you can use the built-in function `bin()` to print your results in binary format.

   (a) An integer number n can be divided by 2 when its least significant digit is 0. Similarly, it is divisible by 4 when its two least significant digits are 0, etc. Write a code to check whether n is divisible by 2^m, with $m > 0$. The solution must use the binary operator &. *(Hint: first create the appropriate binary mask.)*

(b) Solve the same problem using only the shift-left (<<) and shift-right (>>) operators.

(c) An identity operation is such that when applied to a number it produces the same number. Which are the identities for the | (bitwise OR) and for the & (bitwise AND) operators? Produce an example for each. Can you find an equivalent for the ^(XOR) operator (you may need more than one XOR)?

6. **Akima interpolation**
Write an Akima sub-spline interpolator. Recall that if the known "knots" are (x_i, y_i), then the slope of the straight line segment between (x_i, y_i) and (x_{i+1}, y_{i+1}) is p_i, and the derivative of spline segment i at x_i must satisfy:

$$y'(x)|_i (x_i) = \frac{w_{i+1}p_{i-1} + w_{i-1}p_i}{w_{i+1} + w_{i-1}},$$

with weights $w_i \equiv |p_i - p_{i-1}|$. See the lecture slides for the edge constraints. Similar to the regular cubic spline, solving for the four cubic polynomial coefficients in each interval is done by imposing the constraints that the segments and their first derivatives (given above) are continuous. Apply your interpolator to the first row of the image of exercise 3.