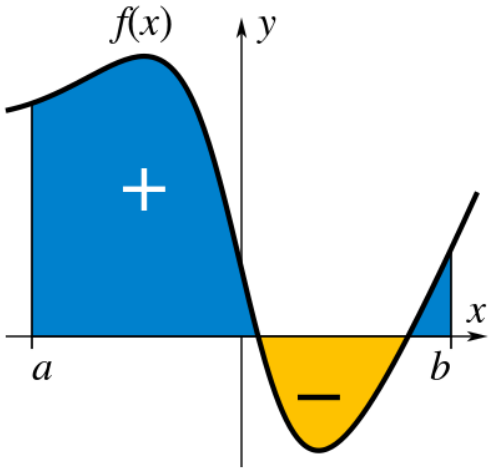
Assignment 4 (Fast python) (30/40 + 5 bonus points)

All solutions should be stored in a directory called `assignment4` in your private repository.

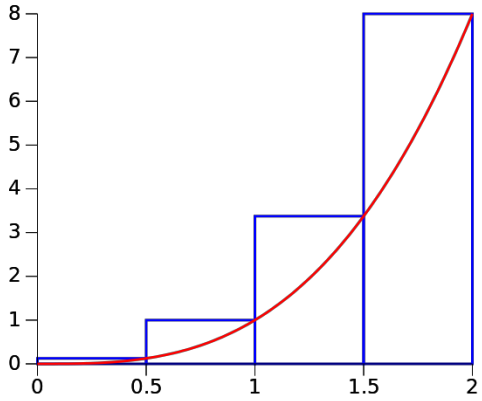
Mathematical background (0 points)

In this assignment, you will make a program which computes the integral of a function numerically. The integral of a function is, essentially, the area under its graph. However, area over the x-axis is considered positive, and area under the graph is considered negative, so the integral of the function in the figure below would be the sum of the two blue areas minus the orange area. The integral of $f(x)$ shown below is written mathematically as

$$\int_a^b f(x)\,\mathrm{dx}$$



A way to compute integrals numerically is as follows: first, divide the interval $[a, b]$ into $N + 1$ points $a = x_0 < x_1 < \ldots < x_N = b$. (In the figure below, $x_0 = 0, x_1 = 0.5, x_2 = 1, x_3 = 1.5, x_4 = 2$.)



Then, on the subinterval $[x_{i-1}, x_i]$, say that $f(x)$ is approximately the same as $f(x_i)$ on the entire interval, so the area under the graph is just $(x_i - x_{i-1})f(x_i)$. To get the total integral, you just sum up all of these terms.

Note that this won't be the exact value of the integral. However, if you use a very large N, you get pretty close[1], as can be seen in this animation.

4.1: Writing test cases (4 points)

Soon, you will create create a Python script which defines a function `integrate(f, a, b, N)` which takes as argument a Python function $f(x)$, the endpoints a, b of an interval and the number of points to use, and returns the numerically computed integral as defined above. First, however, you will create some tests to confirm that it actually works. When doing numerical computation, however, writing tests is slightly tricky for two reasons.

One is that because you'll be doing machine arithmetic on floating point numbers, your answers will have random, small roundoff errors, so instead of checking something like

```
computed_answer == expected_answer
```

it is better to test

```
abs(computed_answer - expected_answer) < 1E-20
```

The other related source of trickiness is that you're fundamentally *approximating* something, not computing it exactly. So even if you know what the right answer is, your program is computing an approximation to it, and even if your program works perfectly, it might still not get the right answer if the approximation method isn't exact. To get around this, you can either make a test looking at a case where the approximation method is in fact exact (for example, a constant function) or choose a function and work out how good the approximation should be for that function.

[1] The error will be at most C/N for some constant C.

Let's do both! Write two tests. In one, verify that if you use the `integrate` function to integrate $f(x) = 2$ from 0 to 1 numerically, you get (up to roundoff errors) 2 no matter what N is.

Call this test `test_integral_of_constant_function`.

In the other, verify that if you integrate $f(x) = 2x$ from 0 to 1 numerically, you get about 1, with the error being about $\frac{1}{N}$.

Call this test `test_integral_of_linear_function`.

Name of file: `test_integrator.py`

4.2: Python implementation (6 points)

Next, create a Python script which implements the function `integrate(f, a, b, N)` and uses it to approximate the integral of $f(x) = x^2$ from 0 to 1. The true value of this integral is $\frac{1}{3}$. Plot the error as a function of the number of points N you use.

Name of files: `integrator.py` (code), `quadratic_error.png` (plot)

4.3: numpy implementation (6 points)

The runtime of your program in 4.2 will probably be about linear in N, meaning that if you double N, the runtime will double too. When N is small, this isn't really a problem, but when N grows large, it becomes a problem.

Make a similar script to 4.1 so that all computationally heavy bits use numpy arrays. Call your function `numpy_integrate`, and add tests for it to `test_integrator.py` [2] Compare the runtime on some input of your choosing. How much is gained by switching to numpy?

Name of file: `numpy_integrator.py`, `report3.txt`

4.4: numba (6 points)

Use numba instead of numpy. Make another function `numba_integrate` script using numba to do the same computations. How much speed is gained by switching to Numba? Can you think of any other advantages to using Numba instead of numpy?

Name of file: `numba_integrator.py`, `report4.txt`

4.5: Cython (INF4331 only: 10 points)

Redo 4.3 using Cython. Compare the runtime as before.

[2] It's fine and in fact smart to reuse the tests from 4.1. Because these two functions are supposed to do exactly the same thing, running all the tests in the same way for both is a good way to verify that.

This problem is only worth the full 10 points if 4.6 and 4.7 are completed. Otherwise it will be worth slightly less.
Name of files: `cython_integrator.py`, `report5.txt`

4.6: Compute smarter, not harder (5 points)

Returning to the introduction, the idea behind our approximation was that on the subinterval $[x_{i-1}, x_i]$, $f(x)$ was approximately constant and equal to $f(x_i)$, its value at the endpoint of the subinterval. However, we could also have chosen to approximate it by its value at the *midpoint* of the subinterval, $f\left(\frac{x_{i-1}+x_i}{2}\right)$. This does, perhaps surprisingly, yield a better result.

For each of the implementation methods from before (pure Python, numpy and numba (and Cython if you did 4.5)), add a function `midpoint_integrate` with the same call signature, but using this alternative approximation scheme.

Then, write a new script which uses all of the methods you now have written to compute the integral of $f(x) = \sin(x)$ from $x = 0$ to $x = \pi$. The real answer is 2. For each of your functions, find the required N such that you get to within 10^{-10} of the actual answer.

Name of files: `integrator_comparison.py`, `report4.txt`

4.7: Making a module (3 points)

Make your implementation into a Python module or package and make a setup script. Your package should include all the methods `integrate_...` and `integrate_midpoint_...`. [3] You are free to make other methods which you think add useful functionality.

Further, make sure your unit tests are packaged with the module in a `test` directory.

4.8: Bonus contest: taking it to the limit (5 bonus points)

Finally, use whichever implementation you liked best to estimate the integrals of some weird functions:

1. $\int_0^\infty \frac{1}{\pi} \frac{\sin(x)}{x} \frac{\sin(x/3)}{x/3} \frac{\sin(x/5)}{x/5}\,\mathrm{dx}$
2. $\int_0^\infty \frac{1}{\pi} \frac{\sin(x)}{x} \frac{\sin(x/3)}{x/3} \frac{\sin(x/5)}{x/5} \frac{\sin(x/7)}{x/7}\,\mathrm{dx}$
3. $\int_0^\infty \frac{1}{\pi} \frac{\sin(x)}{x} \frac{\sin(x/3)}{x/3} \frac{\sin(x/5)}{x/5} \frac{\sin(x/7)}{x/7} \frac{\sin(x/9)}{x/9} \frac{\sin(x/11)}{x/11}\,\mathrm{dx}$

[3] If you want to find a tidier naming scheme like for example `integrate(f, a, b, N, method=''midpoint'', implementation=''numpy'')`, that's fine too.

4. $\int_0^\infty \frac{1}{\pi} \frac{\sin(x)}{x} \frac{\sin(x/3)}{x/3} \frac{\sin(x/5)}{x/5} \frac{\sin(x/7)}{x/7} \frac{\sin(x/9)}{x/9} \frac{\sin(x/11)}{x/11} \frac{\sin(x/13)}{x/13}\,\mathrm{dx}$

5. $\int_0^\infty \frac{1}{\pi} \frac{\sin(x)}{x} \frac{\sin(x/3)}{x/3} \frac{\sin(x/5)}{x/5} \frac{\sin(x/7)}{x/7} \frac{\sin(x/9)}{x/9} \frac{\sin(x/11)}{x/11} \frac{\sin(x/13)}{x/13} \frac{\sin(x/15)}{x/15}\,\mathrm{dx}$

6. $\int_0^\infty \frac{1}{\pi} \frac{\sin(x)}{x} \frac{\sin(x/4)}{x/4} \frac{\sin(x/4)}{x/4} \frac{\sin(x/7)}{x/7} \frac{\sin(x/7)}{x/7} \frac{\sin(x/9)}{x/9} \frac{\sin(x/9)}{x/9}\,\mathrm{dx}$

In each case, actually using $b = \infty$ is hard, so just use $b = 10^7$ instead, which will be sufficient to give you an error of at most 10^{-13}. Also, use $a = 10^{-20}$ or so if you have trouble with division by 0.

Try to push your code as far as it can go and get the best approximation you can. Submit the script you used to get your estimates and a report of your results.

There will be a prize for whoever gets closest to the true answer on 4.8.

Name of files: `contest.py`, and `report.txt`

Clarifications

4.4

- When using numba, it is fine if your program performs poorly when the user passes in a 'weird' function (for example one which won't compile nicely by use of `numba.jit`).

4.8

- The bonus points are awarded just for taking part in the contest. The prize is something else.