Computational Physics, Homework 1

William F. Schiela

good!

10/10

September 27, 2019

The files for this homework can be found at github.com/bschiela/comp-phys/. Python code is located in the `python/` directory. Output data files and plots are located in `python/output/`. This LaTeXdocument is in the `latex/` directory.

Numpy automatically upcasts to double-precision float64 in arithmetic operations between a numpy float32 and a python int. For example, the result of (float32 + int) is a float64. To prevent upcasting, the int must first be cast to float32. To avoid having to explicitly cast every time you want to use an int, I wrote the module `singlep_int` to provide float32 representations of int from 0 to 20, bound to names prefixed by an underscore, e.g. `_2 = float32(2)`.

1 Numerical differentiation

The script that generates the plots and data for this section is `diff_err.py`.

a) Algorithms

Forward-, central-, and extrapolated-difference algorithms for computing the numerical derivative are implemented and documented in `python/lib/diff.py`.

The forward-difference approximation is just the definition of the derivative in terms of a forward finite difference, for finite h:

$$f'(x) \approx \frac{f(x+h) - f(x)}{h} = f'(x) + \mathcal{O}(h)$$

where the second relation follows from the Taylor expansion of the numerator about x. The truncation error is therefore linear to leading order in h.

The central-difference approximation is the same, except the finite difference is taken symmetrically about x:

$$f'(x) \approx \frac{f(x+\frac{h}{2}) - f(\overset{\text{typo?}}{\boxed{h+}}\frac{h}{2})}{h} = f'(x) + \mathcal{O}(h^2)$$

where the second relation again follows from the Taylor expansion of the numerator leading to cancellation of the terms linear in h. The truncation error is therefore quadratic to leading order in h.

The nth-order extrapolated-difference approximation is obtained by fitting an n-order polynomial to the function sampled at $n+1$ points placed symmetrically about x and taking the coefficient of the linear term. The 1st- and 2nd-order extrapolated-differernce approximations simply reduce to the central-difference approximation. The quartic ($n = 4$) extrapolated-difference approximation yields

$$f'(x) \approx \frac{f(x-2h) - 8f(x-h) + 8f(x+h) - f(x+2h)}{12h} = f'(x) + \mathcal{O}(h^4)$$

where the second relation follows as before. Thus the truncation error is quadratic to leading order in h.

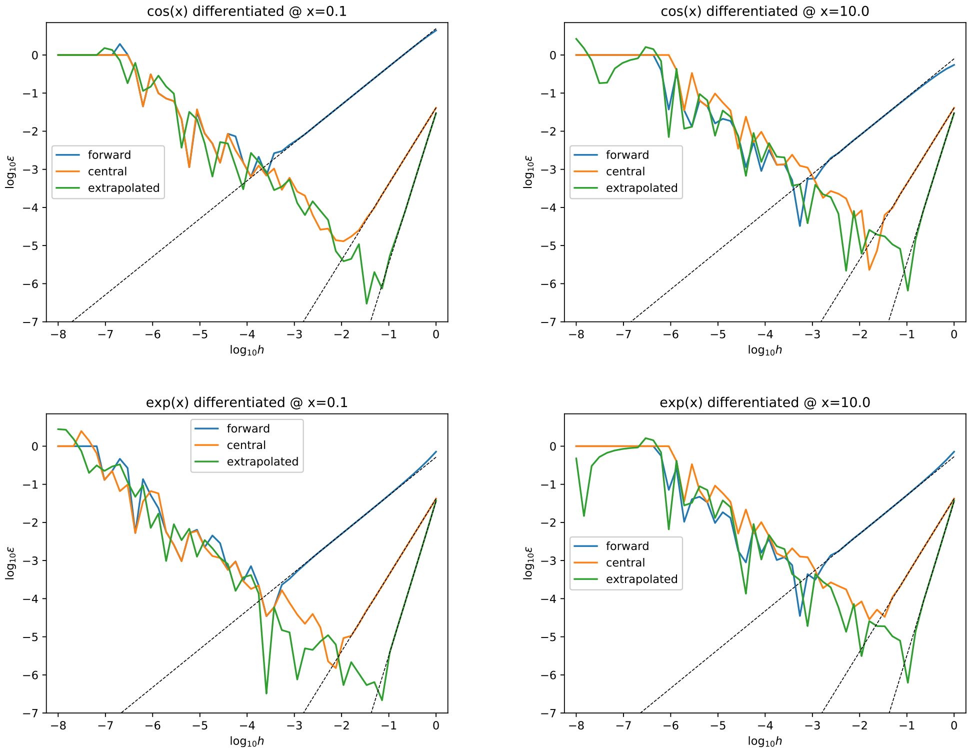
b) Errors vs. step size

Figure 1 shows log-log plots of the relative error vs. step size for each given function at each given point. In each plot, the errors begin to flatten out below $h \approx 10^{-7}$ which is about the machine precision of a 32-bit CPU. As we are limiting our numerical calculations to single-precision (i.e. 32 bits), this is likely an indication that we are bumping up against the limits of single precision.

Each figure also contains a linear least-squares fit in the region of large h where truncation errors dominate. The slopes obtained from each fit are tabulated in Table 1. For forward-, central-, and quartic extrapolated-difference appoximations we obtain slopes $m \approx 1$, $m \approx 2$, and $m \approx 4$, in agreement with the power-law error scaling discussed in Section a).

define <u>m</u> first, the state values.

We can also estimate the scaling of the errors by considering the number of significant digits in our approximation, i.e the number of digits that



✓

good.

Figure 1: Log-log plots of the fractional error ε vs. step-size h obtained for forward-, central-, and extrapolated-difference derivatives, for functions $\cos(x)$ and $\exp(x)$ each evaluated at $x = 0.1$ and $x = 10$. The dotted lines are linear least-squares fits in the regime of large h, where truncation errors dominate.

$f(x)$	x_0	forward	central	extrapolated
$\cos(x)$	0.1	0.99773043	1.9976448	3.956284
	10	1.008445	1.9913602	3.9520204
$\exp(x)$	0.1	1.0064834	2.003053	4.044973
	10	1.0125173	2.0078945	4.0401845

Table 1: Slopes obtained from least-squares fits of the linear regions in each plot of Fig. 1. Reproduced from `python/output/deriv-errs_fit.tsv`.

x_0	$\left.\frac{\mathrm{d}}{\mathrm{d}x}\cos(x)\right\|_{x_0}$	$\left.\frac{\mathrm{d}}{\mathrm{d}x}\exp(x)\right\|_{x_0}$
0.1	-0.09983342	1.105171
10	0.5440211	22026.465

Table 2: True single-precision values of the derivatives.

agree with the true single-precision values of the derivatives tabulated in Table 2. Taking the derivative of of $\cos(x)$ at $x = 0.1$ as an example, we consider the values obtained for the numerical derivative tabulated in `derivs_cos-0.1.tsv` for each approximation, compared to the true single-precision value in Table 2, rounded to 1 significant digit, 2 significant digits, and so on. We see that the forward-difference approximation gains a significant digit for roughly every 6 logarithmic decrements of h. Similarly, the central- and extrapolated-difference approximations gain a significant digit for every 3 and 1.5 logarithmic decrements (on average), respectively. In other words, the truncation error of the central-difference approximation decreases twice as fast as that of the forward-difference approximation on a logarithmic scale, and that of the extrapolated-difference approximation decrease four times as fast, as we expect. ✓

c) Truncation and roundoff error

As previously discussed, truncation errors are dominant in the linear regions of Figure 1 when h is large. As h decreases, the errors reach a minimum value. Further decreasing h beyond this point increases roundoff errors, and the relative error begins to increase again in the regime of small h (left side of the plots).

2 Numerical integration

The script that generates the plots and data for this section is `integ_err.py`.

a) Algorithms

Midpoint, trapezoid, and Simpson's rule for computing numerical integrals are implemented and documented in python/lib/integ.py.

According to the midpoint rule for an integral I of a function $f(x)$ on the interval $x = a$ to $x = b$ subdivided into N bins,

$$I \approx \frac{b-a}{N} \sum_{i=1}^{N} f(m_i) = I + \mathcal{O}(N^{-2})$$

where m_i is the midpoint of bin i.

According to the trapezoid rule:

$$I \approx \frac{b-a}{2N} \left(f(x_0) + f(x_N) + 2 \sum_{i=1}^{N-1} f(x_i) \right) = I + \mathcal{O}(N^{-2})$$

where x_i are the boundaries of the bins.

According to Simpson's rule:

$$I \approx \frac{b-a}{3N} \left(f(x_0) + f(x_N) + 4 \sum_{\substack{i=1 \\ i+=2}}^{N-1} f(x_i) + 2 \sum_{\substack{i=2 \\ i+=2}}^{N-2} f(x_i) \right) = I + \mathcal{O}(N^{-4})$$

where the index of the two summations each increment by 2.

I initially implemented these algorithms by using numpy to implicitly map each sum over an array of bin values. When performing the integral

$$I = \int_0^1 e^{-t} \, \mathrm{d}t \tag{1}$$

with $N = 2^{31}$, my code took over an hour to run and eventually hit a segmentation fault:

```
[3]- Segmentation fault (core dumped) python integ_err.py
```

hopefully you're not creating a 2^{31}-sized array! no need.

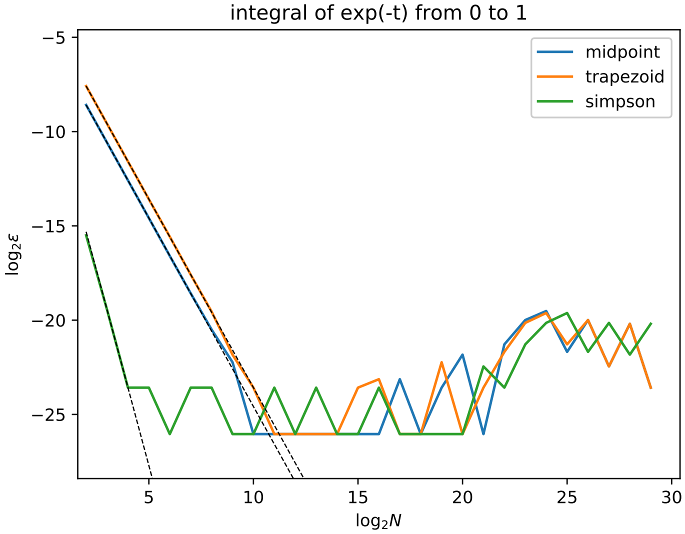
Thinking this may be due to memory allocation issues, I tried an alternative implementation of each algorithm using an accumulator variable and looping over each bin, rather than allocating a potentially large number of bins. I wrote the script integ_timer.py to compare the speed of each implementation, and found that the accumulator method was orders of magnitude slower (the second column is in seconds):

```
# output of integ_timer.py
N = 2^22
integ_mid 0.14608599999701255
integ_mid2 6.010423799998534
integ_trap 0.10156339999957709
integ_trap2 6.294668399998045
integ_simp 0.0769916999997804
integ_simp2 6.620162600000185
```

As a result, the domain of Fig. 2 is limited to $\log_2 N < 30$.

Way more than enough.

b) Errors vs. bin size



good.

Figure 2: Log-log plot of the relative errors ε of the numerical integration (1) as a function of number of bins N using the midpoint-, trapezoid-, and Simpson's rules. The dotted black lines are linear least-squares fits in the region of small N where truncation errors dominate.

A log-log plot of relative error vs. bin size is given in Fig. 2.

✓ you could just state expectations.

	midpoint	trapezoid	simpson
slope	-1.9988084064251928	-1.9990198028537096	-4.132480906723929

Table 3: Table of slopes obtained from linear least-squares fits in the regime of small N where truncation errors dominate. Reproduced from `integ-errs_fit.tsv`

c) Discussion

As in the case of differentiation, truncation errors dominate for large h, which corresponds to small N. The regions dominated by truncation error are linear on a logarithmic scale, as shown in Fig. 2. A table of slopes obtained from a linear least-squares fit for each integration method is given in Table 3, from which it is clear that the error-scaling obtained agrees with those stated in Section a).

At large N, corresponding to small h, roundoff errors ✓ dominate and we begin to see an increase in the relative error. There is a significant region in the middle of the plot where the errors oscillate between two values. This is due to the limits of single precision, and the two values of the error correspond to the two single precision values of the integral obtained, 0.63212055 and 0.6321206, on either side of the true double precision value of the integral, 0.6321205588285577. (These values can be obtained by inspecting `integs.tsv`.) In order to avoid divide-by-zero errors when taking the logarithm of the error in this region where the deviation of the single-precision approximation from the true single-precision value is identically zero, I compared the the single-precision approximate values to the double-precision true value. This is why the values in Table 3 are also double-precision.

3 Baryon acoustic oscillation

The script that generates the plots and data for this section is `bao.py`.

I used numpy's implementation of Simpson's rule to compute the correlation function. First I interpolated $P(k)$ using `scipy.interpolate.interp1d`, an implementation of linear interpolation. I then generated $75,000$ bins spaced logarithmically in k, and used these to evaluate the integral to obtain the correlation function. The result of this analysis is shown in Fig. 3, and

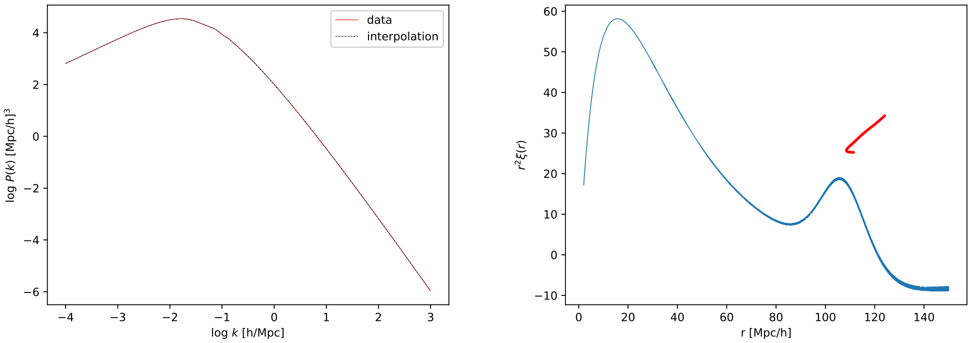


Figure 3: (left) A log-log plot of the power spectrum, overlaid with the interpolation used to compute the correlation function. The baryon wiggles are visible around $\log(k) \approx -1$ h/Mpc. (right) The correlation function scaled by r^2. The baryon acoustic oscillation peak due to the baryon wiggles is visible around $r \approx 104.28$ Mpc/h

the scale r of the BAO peak due to the baryon wiggles in the power spectrum is about 104.28 Mpc/h.