Last updated: 12/29/19

MATH 563, SPRING 2019
SOME GUIDELINES FOR COMPUTATIONAL PROBLEMS

1. Preparing homework with code/data

Follow these guidelines for computational work. In short: write with purpose; save trees!

The written part:

a) **Data/plots are part of the answer:** Use computational results - including plots and table as needed - to support your answers. The relevant data should be referenced from your solution and easy to find (on the same or nearby page, clearly labeled). Your data should make sense on its own, without the code used to generate it. Try to keep figures/data to a reasonable size - save trees and have several figures on one page (they rarely need to span the whole page).
See section 3 and section 4 for suggestions on data/plots.

b) **Save more trees!** For typed homework only, you can submit via pdf with the code; in this case just submit no written work. If handwritten, turn in the hard copy unless you are confident your scan is high quality (not recommended).

c) **Algorithms:** Do not reference code directly or submit it with written work (see next section) unless necessary. Code snippets are okay as needed. To discuss an algorithm, it's usually best to use pseudocode, which is easier to digest.

d) **No magic numbers:** A computational result (e.g. $x_{31} = 1.56739$) is meaningless on its own. The source of this number - the algorithm and parameters used to generate it - should be clear in the solution. Ask yourself: could someone in the class reading the solution reproduce your result? *Note: you don't need to report every last detail - just enough that the mathematical steps could be reproduced.*

The code:

a) **Submission:** To Sakai; the assignment will state what must be submitted. The format will be noted on Sakai (follow the instructions there!).

b) **Functionality:** The code should run out of the box. There must be a 'main' function (or main file in Matlab) that runs and includes a typical working example. In general, it does **not** also need to include all the calculations you used in the homework - think of it as a way to show how your code works.

c) **Language:** The code should be written in one of the supported languages for the course - either `python` or `matlab`. If using `octave` instead of `matlab`, note this in a comment at the top (it *should* run in `matlab`, but there are incompatibilities).

d) **Code/work separation:** Your solutions to problems must be complete. The code should not contain this work (e.g. answers, discussion). The only exception is that if using `jupyter` notebooks in python, you may mix your solutions and code.

2. Programming guidelines/suggestions

Debugging: The implementation can be subtle in numerical computing; protect your sanity and time by practicing good coding habits. A few suggestions:

a) **Verbose output:** Write code to output of your algorithm's progress such as

```
printf("iter %d: %.3f %.3f\n", iter, x(j), y(j))
```

This is useful to a user and for discussing the code. The debug version can/should have more detailed output. Switch off the detailed debugging before submitting!

b) **The debugger is your friend:** The debugger gives you a complete view of your code's inner workings at the breakpoint of your choice. Use it - doing so reduces the amount of temporary print statements you need to shove into the code to test.

c) **Ask others!** If you run into a strange bug or are not sure how to get around one, ask a friend or come to office hours - this can save you time. If coming to office hours to ask about code, make sure that code is available (best: bring a laptop; possible: send a (private) message to me on Piazza with your code).

d) **Translate between math/code:** When your code implements an algorithm, you have the 'math' version and the 'code' version. Try to keep the notation similar. You should have a clear way to 'translate' between the two.

Comment your code to note how the code relates to the algorithm, e.g.

```
x = a*x + b*y; # update x_(n+1) = a*x_n + b*y_n
```

for a recurrence that overwrite x_n with x_{n+1}.

Writing out the relations by hand can help too, e.g. a correspondence like

$$x_1, x_2, \cdots, x_n \iff \mathrm{x}[0], \mathrm{x}[1], \ldots \mathrm{x}[\mathrm{n}-1]$$

for a vector $\mathbf{x} = (x_1, x_2, \cdots, x_n)^T \in \mathbb{R}^n$. Protect yourself against indexing errors! It's worth keeping close track of 'math indexing' vs. 'code indexing'.

e) **Document code (but be concise)!** Use comments to denote key steps. However, avoid over-commenting. Clear code does not need too many comments.

f) **Document function inputs** It is good practice to describe the input parameters to your function (ask: does the user know how to call your function?). Matlab:

```
function y = fzero(f,x0)
% FZERO(f,x0) compute a root of a scalar function f(x)
% inputs: f (function handle), x0 (starting guess)
```

Python:

```
def fzero(f,x0):
    """ Computes a root of a scalar function f(x)
        Args:
            f: the function (double -> double)
            x0: the initial guess
        Returns:
            y: the estimated root
    """
```

This is particularly important in a language like Matlab/python where types and returns are not stated in the function definition.

g) **Short names are okay:** Unlike in general software design, we often use one or two letter variable names like `xn` for x_n. If the short name makes the code easier to read (knowing the math context), it's usually okay (e.g. `xn` vs. `nth_iterate`).

Algorithm design:

a) **Be general...** Your algorithms should be 'generic' in that they should take inputs as general as the algorithm allows. Problem specific values should not be in the function. For instance, a root finder fzero valid for real functions $f : \mathbb{R} \to \mathbb{R}$ with an initial guess x_0 and a max. number of iterations N might be

```
function y = fzero(f,x0,it)
```

where `f` is some general function (e.g. `f(x) = @(x) sin(x)`). Note that max. number of iterations is not hard-coded here since it often needs to be changed.

b) **...but not too general:** You do need to make decisions about scope: your algorithm can only solve certain problems. At a certain point, generalizing just makes a mess, and it's better to just have two separate pieces of code for two cases.

c) **Use helper functions (non-Matlab):** Write simple sub-routines to do basic tasks (for instance, `norm1(x)` to calculate $||\mathbf{x}||_1 = \sum_{k=1}^{n} |x_k|$ for a vector $\mathbf{x}$). Rule of thumb: if you write a piece of code twice, it wants to be made into a sub-routine. If three times, you really should write one.

Sub-routines help to break up your code which makes debugging easier - plus you can recycle sub-routines for later. Python is great for this (it is easy to write versatile, short functions).

d) *Note on matlab and sub-routines:* You can either write helper functions in their own `.m` files (not ideal, but that's what Matlab wants), or write local functions inside your algorithm (just be careful with scope, since nested functions can share variables with their parent). For this reason, there's more incentive to write algorithms as 'one function'.

3. Plotting

Here are some guidelines to follow when making plots. The short version is: your plots should be clear and used for some purpose in your answer. A good plot can be most of an argument; a bad plot only confuses the reader.

a) **Plot size:** Keep plot images to a reasonable size. **Include key plots in your written solutions** (they are a part of the answer) - this does mean printing them out. Unless the figure is quite complex, it should not take up an entire page.[1] Also make sure that your text (labels etc.) have the right font size in submitted work

b) **Plot resolution:** Save plots as `eps` or `pdf` if possible. Other non-vector formats like `png` work too but they are not scalable (do not scale up a png plot!). Use the `print` command (in MATLAB) with the `-dpdf` or `-depsc` commands.[2]

c) **Labels:** Include axis labels and so on. Legends and captions are useful. However, a good description in text and a 'see Figure N' can remove the need for a caption (the description should be somewhere, but you don't need to repeat yourself). *Note: It's fine to sketch annotations on the plot by hand (doing so by code can be annoying).*

d) Ask yourself: What point is this plot trying to make? The plot should be designed to illustrate this point (see below). Unmotivated data is not useful data.

e) **Data visibility:** The trend of the data should be clear in the plot. Choose axis limits and scales correctly. Use log-log and semi-log plots when appropriate. Make sure key features are visible at a glance.

4. Presenting data

You will often need to show a table or sequence of data. The main point is that you should be concise and organized - the data is used to make a point and must be comprehensible.

a) **Formatting data:** Organize data into readable tables. You can format by hand (copy, paste, etc.) or use your code output. The best choice is the formatted output command (Matlab/C: `fprintf/printf`; Python: `print`), e.g.

```
fprintf("n = %d \t x = %.4f \n", n, x);
```

b) **Don't show too much:** Most of the time, you can get away with showing a subset of the data. For example, suppose a problem asks you to use 100 iterations to compute a sequence (x_n, y_n) converging to (x^*, y^*) and report the results.

Then you can just show the first few / last few iterations to report the data. To show the rate etc., of course, you'd want to have a plot with all the data (for large sets of data, plots are much better than tables).

c) **Comparing data:** Often, data is reported to compare. In this case, show what you need to make that comparison. Thjs means (a) report enough digits to see differences and (b) show only the data that matters.

[1] In particular, if using Matlab this means changing its inexplicably huge default size to something better.

[2] For python: use `pyplot` (see https://matplotlib.org/users/pyplot_tutorial.html).

5. An example solution

We'll study this in detail later. The point is to give an example of an answer motivated by 'numerical evidence' (do the computation; analyze the results).

Problem: The 'forward difference' approximation to $f'(x)$ is

$$f'(x_0) \approx D(f,h) = \frac{f(x_0+h) - f(x_0)}{h}.$$

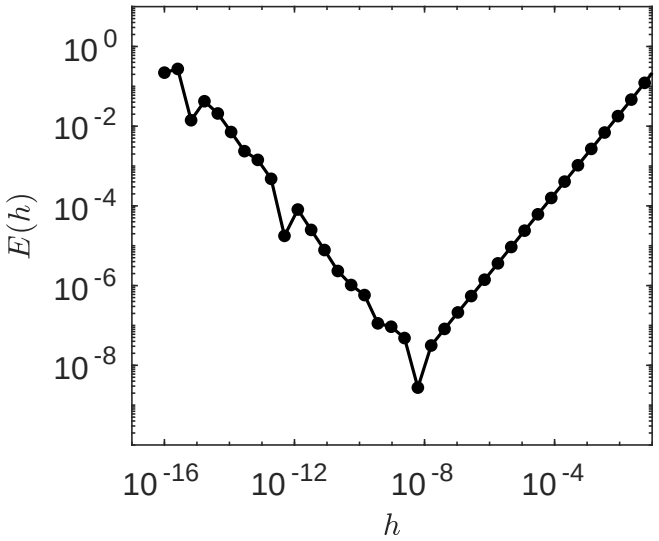
Suppose f is computed to an accuracy of $\eta = 2 \times 10^{-16}$ (rounding error for a `double`) and $f(x) = e^{2x}$ and $x_0 = 0$. How does the approximation depend on the choice of h? Is it true that the error improves as h becomes smaller?

Solution: Since $f'(0) = 2$ is known exactly, we can compute the error

$$E(h) = |f'(x_0) - D(f,h)|. \tag{1}$$

A table of selected values is below along with a plot of the error (1).

h	$D(f,h)$	$E(h)$
10^{0}	6.389056	4.39×10^{0}
10^{-2}	2.020134	2.01×10^{-2}
10^{-4}	2.000200	2.00×10^{-4}
10^{-6}	2.000002	2.00×10^{-6}
10^{-8}	2.000000	1.00×10^{-8}
10^{-10}	2.000000	1.65×10^{-7}
10^{-12}	1.999956	4.42×10^{-5}
10^{-14}	1.998401	1.60×10^{-3}
(exact)	2.000000	0



There are two clear lines of slope 1 and -1, above/below some minimum at $h^* \approx 10^{-8}$. A linear fit to each half of the log-log data (Figure 1) confirms this. We conclude that the error scales like h when h is above the threshold ('convergence') but grows like $1/h$ when h is below it - the error grows as h decreases if h is too small! That is,

$$E(h) \sim \begin{cases} Ah & \text{for } h \gg h^* \\ B/h & \text{for } h \ll h^* \end{cases}$$

Remark: At this point, we would do the analysis and confirm that the theory predicts both behaviors and this 'optimal' value h^*. The details are discussed elsewhere.

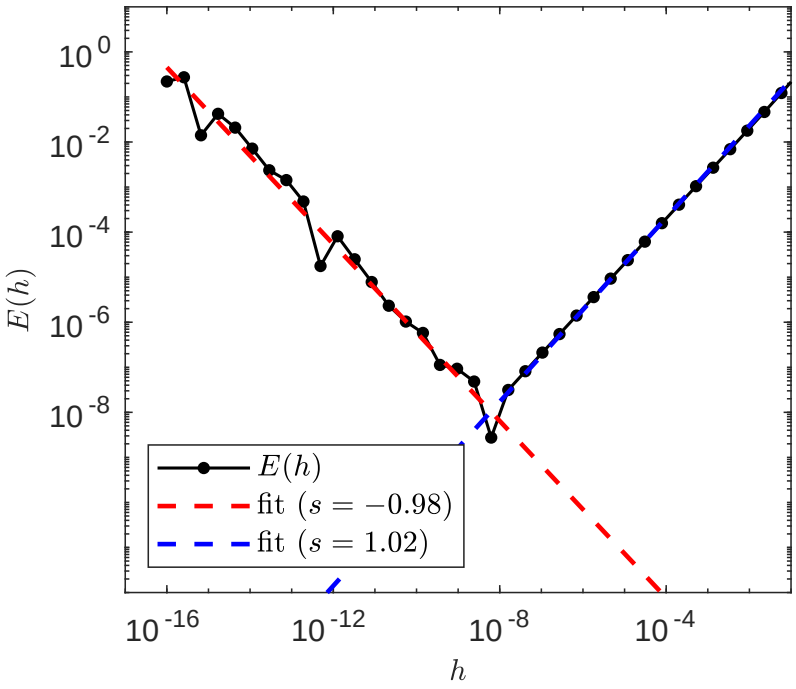


FIGURE 1. Derivative estimate (1) and fits to lines in each 'half'.