Problem Set 2

Due Thursday Feb. 13, 12:30 pm

Comments

- I haven't fully worked these problems myself, so if you run into any strange issues, please post on Ed as there could be mistakes/oversights on my part.
- Please submit as a PDF to Gradescope.
- Please generate the PDF using Quarto. Feel free to work in a Jupyter notebook and then convert to Quarto before rendering to PDF. Other formats that look professional and are designed for working with code and math notation may also be fine - just check with me via a public post on Ed first.
- Remember to note at the start of your document the names of any other students that you worked with on the problem set (or indicating you didn't work with anyone if that was the case) and then indicate in the text or in code comments any specific ideas or code you borrowed from another student or any online reference (including ChatGPT or the like).
- In general, your solution should not just be code - you should have text describing how you approached the problem and what decisions/conclusions you made, though for simple problems, this can be quite short. Your code should have comments indicating what each function or block of code does, and for any lines of code or code constructs that may be hard to understand, a comment indicating what that code does.
- You do not need to (and should not) show exhaustive output, but in general you should show short examples of what your code does to demonstrate its functionality. The output should be produced as a result of the code chunks being run during the rendering process, not by copy-pasting of output from running the code separately (and definitely not as screenshots).
- I do not recommend writing initial answers using a ChatBot, as I think you are likely to fool yourself in terms of how much you are learning about Julia and programming concepts/skills more generally. But it's up to you to decide how heavily to rely on a ChatBot. And refining your initial answers using a ChatBot seems like a good strategy. Using your own knowledge and information online to check the results of a ChatBot and using a ChatBot to check your own coding can both be important/useful.

Problems

1. This problem explores declaring types for function arguments and thinking about subtypes.

    a. Write a function that sums the elements of a dictionary, but only if the values are all integer or floating point scalar numbers, handled by setting the type of the argument to

your function (i.e., it should be Dictionary with types specified for the keys and values). Hint: you will need to use the `<:` syntax. The keys can be any type.

b. Add another method for the function that, for a dictionary whose elements are all either strings or character literals, concatenates the strings or character literals.

2. Let's enhance your Newton function from PS1.

   a. Allow the *objective function* to take arbitrary additional arguments, passed along from your function into the objective function.

   b. Robustify your Newton function. Add error trapping using Julia's exception system. This should catch divergence as well as convergence to a local maximum (the goal of the function is to find a local minimum). Add useful messaging with `@info`, `@warn`, `@error`.

   c. Add handling for the following situations:

      i. When the next point has a objective value than the current point. In that case you can backtrack toward the current point, perhaps first halfway back, then halfway on the remaining half, etc., until you find a new point whose value lower than that of the current point.

      ii. When the next value is outside the range of the previous values. In that case you can use the technique of bisection to find a new point within the valid interval and then proceed with Newton iterations from there.

   d. Set up a few tests for your Newton function using Julia's testing framework (to be discussed in class Thursday Feb. 6).

   Tips: Here are some thoughts about functions you can use as you develop your code and for formal tests. One function you can test on is the sine or cosine function. If the starting point is near enough to a local minimum it should converge to that minimum, but if it is further away it should converge to the nearby local maximum instead. And here's a function that diverges in some cases:

$$x\tan^{-1}(x) - 0.5 * \log(\left|1 + x^2\right|),$$

   because the first derivative is $\tan^{-1}(x)$ (`atan()` in Julia), which flattens out as $|x|$ goes to infinity. However, there are lots of others you could come up, so feel free to work with others that may be better than these.

3. Write a function constructor that creates a version of a function that reports how many times it has been called. Ideally this would work regardless of how many (if any) positional or keyword arguments are used by the function. Additionally, try to manipulate `kwargs` so that the wrapper only reports the number of times run when the wrapped function is called as `myfun(<regular_arguments>, _report = true)`.

   If you want a real challenge, try to do this using a macro. I was able to do this using some meta programming (code manipulation) tools in Julia but it took some Googling and experimentation, and it was helpful that I have experience with meta programming in R.

4. Consider the overdispersed binomial likelihood calculation below.

   The following is the probability mass function for an overdispersed binomial random variable:

$$P(Y = y; n, p, \phi) = \frac{f(y; n, p, \phi)}{\sum_{k=0}^{n} f(k; n, p, \phi)}$$

$$f(k; n, p, \phi) = \binom{n}{k} \frac{k^k (n-k)^{n-k}}{n^n} \left(\frac{n^n}{k^k (n-k)^{n-k}} \right)^{\phi} p^{k\phi} (1-p)^{(n-k)\phi}$$

where the denominator of $P(Y = y; n, p, \phi)$ serves as a normalizing constant to ensure this is a valid probability mass function.

a. Write two functions, (a) one using looping and (b) one using vectorization, to implement the calculation of the denominator above and compare speed, taking care that the time for JIT compilation is not included. (Some reasonable values to use are $n = 10000$, $p = 0.3$ and $\phi = 0.5$.) In your implementations, avoid unnecessary calculations (e.g., simplify/combine terms where possible). Also assess the time for the JIT compilation.

   Tips: Remember that $0^0 = 1$ and that for numerical stability the sorts of calculations done in calculating $f()$ should be done on the log scale before exponentiating at the end.

b. What happens in terms of speed if you do the calculation outside of a function? In this case the JIT compilation is presumably not possible.