Assignment 1

This assignment will give you an opportunity to familiarize yourself with the computing environment you will be using this semester and to practice writing up computing assignments.

Before starting this assignment you should read the coding standards document and the brief introduction to the Git version management system.

You should track your work using a git repository (you can use the same repository for both problems). Your git repository should have at least 2 commits. When you are finished with the homework, run the command

```
git log -p > hw1-gitlog.txt
```

inside of your Git repository. This command writes the Git log to the text file `hw1-gitlog.txt`. Include this file in your submission.

1. The Pareto distribution has density

   $$f(x|\alpha, \beta) = \frac{\beta\alpha^\beta}{x^{\beta+1}}$$

   for $0 < \alpha < x$ and $\beta > 0$. Write a C function to evaluate the Pareto density. Also write a main program that reads values for the arguments x, α, and β, in that order, from standard input and prints the value of $f(x|\alpha, \beta)$ to standard output. I will test your program using commands of the form

   ```
   echo 4.0 2.0 3.0 | ./paretodens
   ```

   In your written report include the code in an appendix, explain how to compile and run the program, and show a few (two or three) examples of its use. Include the program files as text files with `.c` extensions in your submission as well.

2. Write an R function `dpareto` that computes the density of a Pareto distribution. Your function should behave like other R density functions, such as `dgamma` or `dbeta`.

   Your submission should include a file `dpareto.R` suitable for reading into R with the `source` function. I will test your code using the expressions

   ```
   source("dpareto.R")
   dpareto(x, a, b)
   ```

with various definitions of `x`, `a`, and `b`.

In your written report include the code for the function in line, show an example of its use, and show plots of the density for two different sets of parameters. Include the file `dpareto.R` in your submission as well.

Some points to keep in mind:

- Negative values of the parameters are not meaningful; how is this handled for other distributions?
- Densities are defined on the entire real line; they are zero outside of the support of the distribution.
- Make sure that your code is consistent with the coding standards.
- It is a good idea to compile your C code with all useful warnings enabled and to eliminate any warnings you find. For the `gcc` compiler using the options `-Wall -pedantic` enables the most useful warnings.
- Make sure the text and code of your writeup are easy to read. Use appropriate fonts and margins.
- R functions are generally vectorized, i.e. return vectors of results when the arguments are vectors. Your R function should do this as well.

You should submit your assignment electronically using Icon. Your submission should include

- your writeup as a PDF file
- one or more source code files containing your programs for Problem 1 and Problem 2.
- your Git log file.

Submit your work as a single compressed tar file. If your work is in a directory `mywork` then you can create a compressed tar file with the command

```
tar czf mywork.tar.gz mywork
```

Solutions and Comments

General Comments

- Don't include things not asked for (editor temporary files like `foo~`, `.git` subdirectories, executables, etc). To find what is in your archive you can use

  ```
  tar ztf myfile.tar.gz
  ```

- Avoid using file names that contain spaces. Many tools don't work well with such file names.
- If you are asked to use a particular name for a file or a function you *must* must match that name *exactly*, *including capitalization*. Otherwise code that uses your file or function, or automated tests run on them, will fail.
- I will only use and ask for R code files with a `.R` extension.
- R code files should usually only include code, not tests or examples (unless they are commented out in a useful way).

Problems

1. - It is a good idea to compile your C programs with as many warnings enabled as possible. With the `gcc` compiler on the Linux systems, using the flags `-Wall -pedantic` does this.
   - Programs in any language other than machine language are intended to be read by humans. You should try to make your programs as readable as possible.
     - Proper indentation of looping and conditional structures makes programs more readable. There are even some languages, such as Python, where indentation is part of the language syntax! For C, the `emacs` editor can help you indent your code–hitting the TAB key will make emacs indent to the place it thinks appropriate. If the code is not being placed where you think it should go then you probably have a syntax error, such as a missing or extra parenthesis or a missing semicolon. You can also use the program `indent` to indent your C source code according to a reasonable style.
     - I have a strong prference for indenting by 4 spaces at each level.
     - Spaces around operators and after commas also help make your code more readable.
     - Avoid having tab characters in your code — your editor should be able to replace those with spaces.

  - Review the coding standards document[1] and try to follow them.
  - The `indent` program can help; a reasonable result is obtained with
    `indent -kr -nce -nut foo.c -o foo-indent.c`
- The logical *and* operator in C is the double ampersand `&&`. Similarly the logical *or* operator is `||`. The single character operators `&` and `|` are bitwise operators and are very different.
- I used to following code to test your programs:

```
echo 3 2 1  | ./paretodens   # 0.2222
echo 1 2 3  | ./paretodens   # 0.0
echo 3 -2 1 | ./paretodens   # error
echo 3 2 -1 | ./paretodens   # error
```

2.
- Densities are usually defined for all real arguments. If a distribution has limited support then the density is zero outside the support.
- Programs in any language other than machine language are intended to be read by humans. You should try to make your programs as readable as possible.
  - Proper indentation of looping and conditional structures makes programs more readable. There are even some languages, such as Python, where indentation is part of the language syntax! For R, the `emacs` editor with the ESS package can help you indent your code–hitting the TAB key will make emacs indent to the place it thinks appropriate. If the code is not being placed where you think it should go then you probably have a syntax error, such as a missing or extra parenthesis or a missing semicolon.
  - Spaces around operators and after commas also help make your code more readable.
  - Review the coding standards document[2] and try to follow them.
  - The `formatR` package can be useful, for example:
    `Rscript -e 'formatR::tidy_source("foo.R")'`
- If there are range restrictions on your parameters then you should check for those and do something to signal violations in some way. You can either return `NA`, and perhaps signal a warning, or signal an error by calling `stop`. Do not just return whatever value the formula happens to compute — it will be invalid. Also do not return an error message as a text string as this makes life much harder for code that uses your function.
- Functions like `dgamma` return `NaN` and signal a warning for invalid parameters. It is best to follow convention unless there is a compelling reason not to.

[1]https://stat.uiowa.edu/ luke/classes/STAT7400-2023/coding.html
[2]http://www.stat.uiowa.edu/ luke/classes/STAT7400/coding/coding.html

- You should take advantage of vectorized arithmetic whenever possible. Loops and looping functions like `lapply` and friends will be much less efficient.
- To match the behavior of other density functions your function should be vectorized. Your handling of bad parameter values and restrictions on the support of the distribution should also take vector input into account. The `ifelse` function can be useful for this.
- The standard density functions all support a `log` argument. Yours should do this as well.
- If you support the `log` argument, it is almost always better to compute on the log scale and return `exp` of the result than the other way around.
- When plotting a function, line plots for the continuous parts make more sense than point plots. Some thought is needed on how to handle discontinuities.
- When plotting two related densities it is a good idea to help the reader compare the results either by plotting two densities on the same plot or by placing two plots with identical axes next to each other.
- Be consistent in using `=` or `<-` for assignment; `<-` is strongly preferred.
- Make sure to distinguish between the logical operators `&&` and `||` and the vectorized functions `&` and `|`.
- Calls to `return()` are only needed for early exit.
- I used the following code to test your functions:

```
dpareto(3, 2, 1)     # 0.2222222
dpareto(1, 2, 3)     # 0.0
dpareto(3, -2, 1)    # error
dpareto(3, 2, -1)    # error
dpareto(3 : 5, 2, 1) # 0.2222222 0.1250000 0.0800000
dpareto(1 : 5, 2, 1) # 0.0 0.0 0.2222222 0.1250000 0.0800000
dpareto(6, 2 : 4, 1) # 0.05555556 0.08333333 0.11111111
dpareto(3, 2, 1, log = TRUE) # -1.504077
```