Assignment 3: Newton's Method and Root Finding
due: Feb 19, 2019– 11:59 PM

GOAL: The purpose of Problem Set #3 is twofold: one purpose is to investigate searching and root-finding, but the other is to lay the groundwork for Problem Set #4: Gaussian integration, which has a lot to borrow from root finding, of all things. Read Lecture notes Section 6 to see how this all fits together. (Always go to GitHub latest version of Lectures!)

1 Introduction

The ultimate focus of this assignment looking for the zeroes of Legendre polynomials... this isn't made to scare you, but it's meant to emphasize how far you'll go in just a week. Finding the zeroes requires root finding, and if you're wondering what a Legendre polynomial is, well, it's a function we'll build recursively.

As a warm up to this impressive feat, we'll start of with simpler tasks: using *bisection* and *Newton's method* to find zeros of simple polynomials (which will apply to any continuous function, such as those pesky Legendre polynomials). And as a warm up to the warm up, we'll first point out that root finding is similar to searching in a sorted array!

Let's say you have a *sorted* array `c[0], c[1], ...`, and you want to find which index `i` satisfies `c[i] = b` for some `b`, that is, you want to find where `b` resides in the array. This is similar to finding the zero of a function—you're (discretely) solving:

```
c[i]-b = 0
```

With that in mind, let's get to it!

1.1 Using Makefiles

In this homework we are beginning to get different functions in separate files and/or headers. So it is useful (really essential!) to start to organized compilation using Makefiles. You already saw this in the OpenACC exercise. The Makefile is sort of the beginning of a role your own Unix IDE. (**Almost everyone sets up make files by coping existing Makefiles. The rules are so baroque! Really fancy ones are made with automake tools!**). Try the very simple examples on GitHub in directory `VerySimpleMake`. A very basic one you can run by `make -k` and a fancier one you can run by `make -k -f MakeFancy` becuase I gave it a new name. Give them a spin and I hope you'll find it helpful to role your own in these exercises.

2 Warm Up Exercise #1 : Searching an Array

To see the array version you should run the code `search.cpp` with the dependent `.h` files:

```
#include "search.h"
#include "sort.h"
```

that is on `github`. Read the code so you understand it. First set `Nsizse = 100` to see that it is all working. Then go back `= 10000000` to really see the differences between the different algorithms. This is a fun way to appreciate the advantages of better algorithms: LinearSearch, BinarySearch, and DictionarySearch are O(N), O(Log(N) and O(Log Log N) algorithms respectively. Or if you wish 3 algorithms: `Stupid, Smart, Brilliant!` Each has it corresponding version for root finding!

Modify the code `search.cpp` to run for a range of values of N (no larger than `10^8`) and output the results into a file `search_timing.dat`. The file should have four columns:

1. The value of N.
2. The number of iterations for linear search.
3. The number of iterations for binary search.
4. The number of iterations for dictionary search.

Using `gnuplot`, fit the curve to N, Log N and Log Log N to convince yourself that these estimates are right. (The demos on the gnuplot website might be helpful if you're not sure how to fit in `gnuplot`.) The deliverables for this exercise are:

- Your modified code, `search.cpp`.
- Your data file, `search_timing.dat`.
- Plots of your data with fit curves.

Do not stress too much about the plots of the data with fit curves. Don't worry about error bars and properly weighted fits! Later we will discuss proper weighted fits.

(Also note that MergeSort is a smart fast $O(NLog(N))$ sort. Try using a stupid $O(N^2)$ Selection Sort. It is so terrible you will have to run it over spring break or longer!)

3 Coding Exercise #2: Taking a Square Root

Again, we have a prewritten program for simple root finding. The code in `bisection_vs_newton.cpp` explicitly solves $f(x) = x^2 - A$, giving the positive square root of A. Just compile `bisection_vs_newton.cpp` and run. This program is a contest between bisection search, a `Log(N)` method, and Newton's method (think Dictionary search!), a `LogLog(N)` method. Try it for various values of A.[1]

> This exercise has two pieces.
>
> First, modify the code `bisection_vs_newton.cpp` to find the n-th root of A, instead of just the square root. This requires modifying the functions `bisection` and `newton` appropriately to take and use an additional numerical argument, `int n`, which you should prompt the user for. Once you are done, the code will find the zero of $f(x) = x^n - A$. For Newton, this requires the iteration
>
> $$x \leftarrow x \times (1.0 - 1.0/n) + A/(n \times \text{pow}(x, n-1)). \tag{1}$$
>
> Next, we'll focus specifically on square roots and third (cube) roots for $A = 2$. Instead of having the tolerance (`TOL`) fixed at the top of the code, make the tolerance a variable. Your goal is to study the iteration count as a function of the tolerance for `bisection` only. Sweep in N, where $N = 1/\texttt{TOL}$, for $N = 10, 100, \cdots 10^{15}$. Plot the iteration count as a function of N using gnuplot, and fit it to the form $c_0 + c_1 \log(N) + c_2 \log(\log(N))$, where c_0, c_1 and c_1 are free fit parameters.
>
> The deliverables for this exercise are:
>
> - Your modified code, `bisection_vs_newton.cpp`, which prompts the user for and computes arbitrary integer n-th roots using both `bisection` and `Newton's` method.
> - Two plots showing the iteration count for computing the square root and cube root, respectively, of 2 using `bisection`. Your plots should include an overlaid fit curve.

In general, the Newton-Raphson method does not always converge to a zero of the function—there are some pathological cases where the method fails. We won't go very in depth on this; for more details, check out:

- `https://en.wikipedia.org/wiki/Newton%27s_method`, namely the section "Failure of the method to converge to the root".

However try finding the zero of $f(x) = x^3(1 + \cos(10x)/2 - 1/4$ starting in the interval $[0, 3]$ starting with $x = 2$. Newton's method fails but bisection works. Explain why this is the case.

[1] There are some weird cases where bisection will beat Newton's method—for example, the way the code is currently written, bisection will win for $A = 1024$. This is a special case due to how we set our initial guess and because 1024 is an even power of 2, namely 2^{10}. These cases are special!

4 Coding Exercise #3: Polynomials

4.1 Part #1: Generating Legendre Polynomials with Recursion

As a first step, we will generate the Legendre polynomials. Recall that an n-th order polynomial can be defined by its coefficients, $a_n[i] \equiv a[n][i]$ for $i = 0, \cdots, n$. To be more explicit, given these coefficients, the n-th order polynomial (such as the Legendre polynomial) can be evaluated as:

$$P_n(x) = a_n[0] + a_n[1]x + a_n[2]x^2 + \cdots + a_n[n]x^n \tag{2}$$

To specifically evaluate a Legendre Polynomial, all we need to do is construct the array $a[n][i]$. After a bit of searching on Wikipedia, we noted that the Legendre Polynomials satisfy a recurrence relation:

$$nP_n(x) = (2n-1)xP_{n-1}(x) - (n-1)P_{n-2}(x) \ . \tag{3}$$

The first two Legendre polynomials are defined as:

$$P_0(x) = 1 \qquad\qquad P_1(x) = x \ . \tag{4}$$

The recurrence relation and the definition of the first two Legendre Polynomials *uniquely* defines all P_n's. As an example, let's demonstrate finding the coefficients of $P_2(x)$. With complete generality, we can write:

$$P_2(x) = a_2[0] + a_2[1]x + a_2[2]x^2 \ . \tag{5}$$

We're looking to define the $a_2[i]$'s. We know the coefficients of the first two Legendre polynomials, so we can write:

$$P_0(x) = a_0[0] = 1, \tag{6}$$

$$P_1(x) = a_1[0] + a_1[1]x = 0 + (1)x, \tag{7}$$

where $a_0[0] = 1$, $a_1[0] = 0$, $a_1[1] = 1$. To find the coefficients of $P_2(x)$, we can plug these expressions into the recursion relation with $n = 2$:

$$\begin{aligned}
nP_n(x) &= (2n-1)xP_{n-1}(x) - (n-1)P_{n-2}(x) \\
2P_2(x) &= 3xP_1(x) - 1P_0(x) \\
2\left(a_2[0] + a_2[1]x + a_2[2]x^2\right) &= 3x\left(a_1[0] + a_1[1]x\right) - 1\left(a_0[0]\right) \\
2\left(a_2[0] + a_2[1]x + a_2[2]x^2\right) &= 3x\left(x\right) - 1\left(1\right) \\
2\left(a_2[0] + a_2[1]x + a_2[2]x^2\right) &= 3x^2 - 1 \\
a_2[0] + a_2[1]x + a_2[2]x^2 &= \frac{3}{2}x^2 - \frac{1}{2}
\end{aligned}$$

We can now match powers of x on each side. This is known as *linear independence*: We thus get $a_2[0] = -\frac{1}{2}, a_2[1] = 0, a_2[2] = \frac{3}{2}$ and conclude:

$$P_2(x) = \frac{3}{2}x^2 - \frac{1}{2}$$

We can generalize this, looking at each power x^i in Eq. 3, and find the recursion relation

$$n\ a[n][i] = (2n-1)\ a[n-1][i-1] - (n-1)\ a[n-2][i] \tag{8}$$

where you **must be careful of the edge effects**. Coefficients are zero out of range:$a[n][i] = 0$ when $i < 0$ and $i > n$. Why? Your task in this programming exercise is to write a function in `C` that finds the coefficients $a_n[0], a_n[1], \cdots, a_n[n]$ for a general n. If you carefully follow the steps I've given for $n = 2$, you'll see I already wrote it for you! Of course, you should assume that $a_n[i] = 0$ if $i > n$. You may find it useful to represent this in `C` as a two-dimensional array.

The function should be declared as:

```
int getLegendreCoeff(double* a, int n);
```

where:

- `a` is an array of `double`s, already allocated, of length `n`+1.
- `n` indicates the order of Legendre polynomial.
- The return value is 1 if there are no errors, 0 if there are errors. Some errors include:
  - `n`< 0, since we have only defined the Legendre polynomials of integer order. (Yes, there are non-integer extensions. Check them out in Mathematica!)
  - `a` is a null pointer.

This function should be included in a file `legendre.c`, with a main function that prompts the user for a value n, allocates an array, finds the coefficients of $P_n(x)$, and prints the polynomial. As an example, here's how the code may run, where > indicates lines with user input. Assume we've started from a bash shell.

```
> ./legendre
What order Legendre polynomial?
> 5
7.875 x^5 + 0 x^4 + -8.75 x^3 + 0 x^2 + 1.875 x^1 + 0 x^0
```

We're not stressed about how many decimal points your output exhibits (as long as it's at least 6 if there are more than 6 non-zero digits).

The deliverables for this exercise are:

- Your own code file `legendre.c` as defined above with the function `getLegendreCoeff`.

4.2 Part #2: Finding Zeros of Legendre Polynomials

As a next step, we need to find the zeros of the Legendre polynomial. There are many different ways to do this, but for this assignment we will focus on the Newton-Raphson's method. Luckily, the roots of the Legendre polynomial are "well-behaved," in large fact because they are orthogonal polynomials. They are so well-behaved that there are high-quality approximate expressions for the roots of general Legendre polynomials that serve as good initial guesses to Newton's method.

We will state without proof that the kth root of $P_n(x)$ is approximated by the expression:

$$\xi[n][k] \simeq \left(1 - \frac{1}{8n^2} + \frac{1}{8n^3}\right) \cos\left(\pi \frac{4k-1}{4n+2}\right). \tag{9}$$

Since the roots are almost equally spaced in $[-1, 1]$, these approximate roots lead to a quick convergence. For this assignment, write a routine:

```
int getLegendreZero(double* zero, double* a, int n);
```

where:

- `zero` is a pre-allocated array of length n where the (sorted!) zeroes will go when the function returns.
- `a` is a pre-allocated array of length $n + 1$ which contains the coefficients $a_n[i]$ (which you'd compute from `int getLegendreCoeff(double* a, int n)`, I imagine)!
- `n` is, well, n.

As noted, use the "smart" initial guesses given above to populate the array `zero` with the zeroes of the given Legendre polynomial. You might want to test your code for $n = 5, n = 6$, and $n = 7$. While there are an infinite number of Legendre polynomials, your computer (as well as mine) doesn't have an infinite amount of memory, so you can have your code print an error if n is bigger than, say, 30, and of course print an error if n is negative.

Here's a sample output:

```
> ./getZeros
What order Legendre polynomial?
> 5
-0.9061798459386640 -0.5384693101056831 0.0000000000000000
0.5384693101056831 0.9061798459386640
> ./getZeros
What order Legendre polynomial?
> 31
C'mon man, don't ask me for an order greater than 30.
```

You can verify the program to the known values for the zeroes, listed (for example) here. In Problem Set # 3, we'll reuse these functions to construct arbitrarily accurate numerical integration routines using Gaussian quadrature.

The deliverables for this exercise are:

- Your own code file `getZeros.c` as defined above with the function `getLegendreZero`.

4.3 Submitting Your Assignment

This assignment is due at 11:59 pm on Monday, February 6. Please e-mail a **tarball** containing the assignment to the class e-mail, bualghpc@gmail.com. Include your name in the tarball filename.

If you're not familiar with tar, here's a sample instruction that perhaps I would use:

```
tar -cvf rich_brower_hw3.tar [directory with files]
```

You may want to include other files. **Do NOT include a compiled executable!** That's a dangerous, unsafe practice to get into.