18.330 Problem set 1 (spring 2020)

Submission deadline: 11:59pm on Monday, February 10

The questions are a mixture of hand / analytical calculation and numerical calculation using Julia. You should submit a PDF file. You may photograph / scan your handwritten solutions for the theory parts, or (preferably) type them up as part of a Jupyter notebook that you print to PDF.

Exercise 1: Evaluating polynomials Consider the polynomial function

$$p(a, x) = a_0 + a_1 x + \cdots + a_n x^n.$$

1. Write a function `poly_eval(a, x)` to evaluate $p(a, x)$ in the naive way by just summing the terms as written. This should accept the coefficients $a_0, \ldots, a_n$ as a vector.
2. The **Horner method** is an alternative evaluation algorithm in which no powers of x are (explicitly) calculated.

   For example, for a quadratic $p(x) = a_2 x^2 + a_1 x + a_0$ we have $p(x) = a_0 + x(a_1 + a_2 x)$

   Generalise this to polynomials of degree 3 and then degree n
3. Implement this as a function `horner(a, x)`.
4. Test `horner` to make sure it gives the same result as `poly_eval`.
5. Benchmark the two methods to evaluate the polynomial $p(a, x) = 1 + 2x + 3x^2 + \cdots + 10x^{10}$. Which algorithm is more efficient and by how much?
6. Count (approximately) the number of operations required by each of the two algorithms. Does this agree with your benchmarks? (Feel free to perform more benchmarks to check this, e.g. using polynomials with random coefficients.)

Exercise 2: Calculating $\sqrt{y}$ The Babylonian algorithm for calculating $x = \sqrt{y}$ is the following iteration:

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{y}{x_n}\right).$$

1. *Suppose* that x_n converges to some limiting value x^* as $n \to \infty$. Show that then we must have $x^* = \sqrt{y}$.

2. To show that it does, in fact, converge, suppose (without loss of generality) that $x_0 < \sqrt{y}$.

   Show that we then have $x_n < x_{n+1} < y_{n+1} < y_n$ for all n. Hence the x_n form an increasing sequence that is bounded above, and hence they converge. (This is a theorem that we will just assume for this course.)

3. Write a function `babylon` that implements this algorithm. It takes a tolerance ϵ and iterates until the residual is $< \epsilon$. It should return the sequence (x_n).

4. Test your function to make sure it gives the correct result.

5. Use rational initial values to convince yourself that calculating with rationals is a bad idea and that floating point is a good compromise.

6. Let $\delta_n := x_n - x^*$ be the distance of x_n from the limiting value $\sqrt{y}$. (You may use Julia's built-in `sqrt` function, or take the last data value as the limit.)

7. Plot the (absolute value of) δ_n as a function of n on a suitable combination of linear and log scales. How fast does it seem to be converging?

8. Compare this on the same plot to the bisection algorithm from class. Which is better?

9. With δ_n defined as in [6.], show that $\delta_{n+1} \simeq \delta_n^2$ ($\simeq$ means "is approximately equal to").

   Hint: you may need to use a Taylor series expansion; if so you should calculate this (by hand).

Exercise 3: Collision of two discs [In this question we will finally find an actual use case for solving a quadratic equation!]

Suppose we have two discs located at positions $\mathbf{x}_1$ and $\mathbf{x}_2$, with velocities $\mathbf{v}_1$ and $\mathbf{v}_2$, respectively. The discs each have radius r.

1. Find an expression for the condition that the discs collide (i.e. are touching) at time t, involving the distance between their centres.

2. Rewrite your expression from [1.] as a quadratic equation for the time t. How many real solutions may this equation have? What do different numbers of solutions of this equation correspond to physically?

3. Write a function `collision` that takes all the data and calculates the collision time by using the quadratic formula. Make sure you take account of the answer to [2.]

4. Check that your code works by setting up some combinations of discs where you can do the calculation by hand (e.g. both discs moving at a 45-degree angle).

Exercise 4: Evaluating elementary functions

1. Implement the degree-N Taylor polynomial approximation $\exp_N(x)$ for $\exp(x)$ around $x = 0$. Make sure that you do *not* explicitly calculate factorials in your code.

2. Make an interactive visualization using the `Interact.jl` package, showing $\exp_N$ and exp as N varies.

3. Make another visualization showing the **truncation error**, i.e. $\exp(x) - \exp_N(x)$. How does it behave?

4. Calculate a bound for the truncation error using the Lagrange remainder. Use Stirling's approximation to estimate how many terms you need to take for the error to be of a certain size ϵ.

5. Implement **range reduction** for the exponential function: instead of blindly applying a Taylor expansion (which will require many terms for large x), calculate $\exp(x)$ by reducing to the calculation of $\exp(r)$ for $r \in [-0.5, 0.5]$ using the relation

$$\exp(2x) = \exp(x)^2.$$

6. Make an interactive visualization of the Taylor polynomial approximation to $\log(1+x)$, showing visually that it fails outside a certain interval. Which interval is that, and why?

Exercise 5: Exactly representing irrationals using symbolic computing
Here we will represent certain real numbers in the computer *exactly*. Effectively we will use a type of *symbolic* computation, in which we explicitly keep the symbol $\sqrt{2}$, rather than approximating it by a floating-point number.

(Although this is not really the subject of the course, it's useful to remember that there is a whole world of symbolic computation that can be applied to certain problems, and that with a bit of work you often *do not need* an expensive commercial tool to do this!)

Consider the subset of the real numbers given by $S := \{a+b\sqrt{2} : a, b \in \mathbb{Q}\}$, i.e. the set of numbers of the form $a + b\sqrt{2}$ where a and b are rational.

1. Write down formulae for the sum, difference and product of $a_1 + b_1\sqrt{2}$ and $a_2 + b_2\sqrt{2}$, showing that the results are in S.

2. Show that $1/(a + \sqrt{b})$ is also in S by supposing that it equals $c + d\sqrt{2}$ and finding explicit equations for c and d. What type of equations are they? Solve them to find explicit values for c and d in terms of a and b.

[This shows that we can also do division (by non-zero elements) and remain within the set, i.e. that S is a **field**, namely an **extension field** of $\mathbb{Q}$.]

3. Define a type `FieldExtension` to represent these number pairs, and the corresponding operations. Also define a `show` method to print them nicely using a `√` symbol (typed as `\sqrt<TAB>`).

4. Find formulae to represent $1/(1+\sqrt{2}+\sqrt{3})$ as elements of the corresponding set.

Some Julia tips

Package installation

A package such as `BenchmarkTools` may be installed by running the following code a *single* time (only once in your current Julia installation):

```
using Pkg
Pkg.add("BenchmarkTools")
```

Load the package in each Julia session with

```
using BenchmarkTools
```

Benchmarking

Simple benchmarking (i.e. timing how long an operation takes) may be done using the `@time` macro:

~~~
```julia
@time f(x)
```
~~~

However, if the time taken is too short then this is not accurate. Instead, use the `BenchmarkTools.jl` package, with the syntax

~~~
```julia
using BenchmarkTools

@btime f(1, $x)
```
~~~

Note that any variables you pass in must be given `$` signs like this.

Plotting

The `Plots.jl` package is used as follows after loading it with `using Plots`:

- `plot(x, y)`: plots the data with given x coordinates and y coordinates, joining those points with lines. These should be vectors.
- `plot(y)`: specifying only one argument plots the data against the numbers $1, 2, \ldots$
- `plot(-1:0.1:1, f)`: you may pass in a range and a function instead of data.
- `plot!`: adding `!` adds a new plot to an existing one.
- `scatter`: plots points instead of lines
- Add `yscale=:log10` inside the plotting command to use a logarithmic scale on the y axis.

Note that there are small delays when first loading the package and for the first plot. Later plots will be quick.

Interactivity

The `Interact.jl` package enables us to generate simple interactive visualizations using a slider (and some other "widgets").

To generate a single slider, wrap a `for` loop in the `@manipulate` macro, e.g.

```
@manipulate for i in 1:10
    i^2
end
```

To manipulate a plot, put a plot as the result at the end of the `for` loop:

```
@manipulate for i in 1:10
    plot(-5:0.01:5, x -> sin(i * x))
end
```

You can generate multiple sliders by using a joint `for` loop:

```
@manipulate for i in 1:10, j in 0.1:0.1:0.9
    i + j
end
```