Optimization Theory Assignment

Due Sunday 23 December

1. Recall the Collatz function from the first Python assignment. Based on the Python implemention, reimplement in C++, noting similarities and fundamental differences.

2. Write a function `isPrime(int number)` that inputs an integer number and outputs a Boolean which is true iff the input is prime. Now write a function `printPrimes(int upto)` that prints all primes at most `upto`. Test your two functions in one program, choosing sensible test cases.

3. Recall natural-number powers of an integer q, defined by

$$\begin{aligned} q^0 &= 1 \\ q^{n+1} &= q \cdot q^n \ . \end{aligned}$$

   The following functions should test validity of input, and in the error case send a message to `std::cout` and return 0.

   (a) Write a function `pow_iterative(int q, int n)` which uses iteration to evaluate q^n.

   (b) Write a function `pow_recursive(int q, int n)` which instead uses recursion to evaluate q^n.

   (c) A naive algorithm for q^n performs $n-1$ products. Can you suggest a more efficient implementation (in either the iterative or recursive case, whichever you prefer)? [Hint: $q^4 = (q^2)^2$.]

4. Equations that have only one solution in exact arithmetic may have several such solutions in floating-point arithmetic. Consider, e.g., the equation $(1+x) = 1$ for $x \in \mathbb{R}$. In exact arithmetic, the only solution is $x = 0$, but in floating-point arithmetic this equation usually has several further solutions, i.e., all numbers that are too small to have an effect when added.

   Write a program to get used to floating-point arithmetic in C++. It should read in a floating-point number x of type `float` resp. `double` from `std::cin`. Then calculate and print $x \oplus 1$.

   (a) How small do you have to choose x so that you obtain exactly 1?
   Note: Rembember to increase the output accuracy by using `std::setw` and `std::setprecision` (you may search online on how that is done).

   (b) Is this the smalles positive number that can be represented by `float` resp. `double`?

5. Consider the tent map, which maps the unit interval $I := [0, 1]$ onto itself via

$$f: I \to I, \quad f(x) := \begin{cases} 2x & \text{if } x \in [0, 0.5) \\ 2 - 2x & \text{else} \end{cases}$$

This is one of the simplest examples of a nonlinear function describing a dynamic system. For $x_0 \in [0, 1]$ we define the sequence $(x_i)_{i \in \mathbb{N}}$ via

$$x_i = f(x_{i-1}), \quad i \in \mathbb{N}$$

The tent map is a chaotic system, i.e., small changes of x_0 can have large influence on later values and it is apparently not possible to predict values in the sequence. Unfortunately that's not the case if x_0 is a finite binary number. It is therefore not possible to reproduce this behavior on a computer directly.

In this exercise you investigate how the tent map behaves for floating-point numbers, and how the chaotic behavior can be implemented regardless.

(a) Write a C++ program that uses the value $x_0 = 0.01401$ and calculates the terms x_i for $i = 1, \ldots, 100$, printing them to the terminal and producing something similar to:

```
$ g++ -o tent_map tent_map.cc
$ ./tent_map
0.01401
0.02802
...
```

Redirect those numbers to a file `tent.dat`:

```
$ ./tent_map > tent.dat
```

Visualize the data using some plotting tool, e.g., `gnuplot` or Python.

(b) At first the values look chaotic, but a pattern forms rather soon. One can prove that this always happens when using fixed-point or floating-point arithmetic. One can also show that the sequence $(x_i)_{i \in \mathbb{N}}$ never becomes periodic if x_0 is irrational. However, that doesn't help us with floating-point numbers. Instead, implement a slightly changed function:

$$f: I \to I, \quad f(x) := \begin{cases} 1.999999 \cdot x & \text{if } x \in [0, 0.5) \\ 1.999999 \cdot (1 - x) & \text{else} \end{cases}$$

and visualize the resulting sequence as in the previous task.

6. In one of the previous exercises you calculated whole powers of numbers. Here we want to consider a somewhat related problem: we are looking for

the n-th root of a positive number $q \in \mathbb{R}^+$, defined by

$$q^{1/n} = a \in \mathbb{R}^+ \iff a^n = \prod_{i=1}^{n} a = q.$$

In contrast to the power exercise, we now use floating-point numbers, since the root $q^{1/n}$ is not a whole number for most combinations of q and n, even if both are integer. A formula for approximate computation of $q^{1/n}$ is

$$a_{k+1} \coloneqq a_k + \frac{1}{n} \cdot \left(\frac{q}{a_k^{n-1}} - a_k \right),$$

where a_0 is an initial guess, e.g., simply $a_0 \coloneqq 1$, and the sequence of values $a_0, a_1, a_2, a_3, \dots$ produces better and better approximations of the true value $q^{1/n}$ (i.e., converges to it).

Consider the recursion above and argue why it is a special version of Newton's method. Which function is set to zero by this approach (Newton's method interpreted as a root finding scheme)? Which function is minimized by it (Newton's method as a nonlinear optimization method)?

The following functions should always test for valid input, and in the error case write a message to `std::cout` and return zero.

(a) Write a function `double root_iterative(double q, int n, int steps)`, that calculates an approximation of $q^{1/n}$. Here `steps` is the number of steps (and therefore approximations) calculated by the program, with the last one used as return value.

(b) For each iteration $a_k \to a_{k+1}$ you need the $(n-1)$-th power of a_k. You have already written such a function; you just have to note that the data types for input and output need to be modified.

(c) Write a function `void test_root(double q, int n, int steps)`, which tests the accuracy of your calculation. Calculate an approximation $\tilde{a} \approx q^{1/n}$ using your user-defined method, compute the power $\tilde{a}^n$, and print the following values: q, n, $\tilde{a}$, $\tilde{a}^n$ and $q - \tilde{a}^n$. Compare with what you know about the convergence of Newton's method.

7. So far our programs have input and output individual numbers. But often programs store and manipulate larger amounts of data. For this C++ uses containers. In the following you will get to know the most important container type:
`std::vector<T>`,
an indexed list of entries of type `T` (almost any data type, e.g. `int` or `double`). You can create an `std::vector` in several ways:

```
#include <vector> // make vector available in your program

int main(int argc, char** argv)
{
  // an empty vector of integers
  std::vector<int> v1;
  // a vector of 10 integers
  std::vector<int> v2(10);
  // a vector with entries 3, 8, 7, 5, 9, 2
  std::vector<int> v3 = {3, 8, 7, 5, 9, 2};
}
```

As an object a vector has so-called methods, which are special functions that may change the object. For a complete reference see `https://en.cppreference.com/w/cpp/container/vector`.

The main methods for this task are:

```
std::vector<int> v = {3, 8, 7, 5, 9, 2};
// returns the number of entries
std::cout << v.size() << std::endl; // 6
// changes the length of the list
v.resize(42);
```

To access an entry of the vector, write the index of the entry in square brackets after the variable's name. The numbering of the entries starts at 0 as in Python. To change an entry, simply assign a new value to the entry:

```
// accessing individual entries
std::cout << v[2] << std::endl; // 7
v[0] = v[0] * 2;
std::cout << v[0] << std::endl; // 6
```

(a) Create a `vector<int>` using any of the methods described above and enter all entries using a `for` loop. What value do entries have for which you have not specified an explicit value?

(b) Write a function that finds the largest and smallest value in a vector and writes them to the standard output. Test the function with an appropriate choice of vectors.

(c) Write a function `reverse (std::vector<int> v)` which returns the reverse of its input. Test your function on appropriate vectors.

(d) Write a program that takes all the entries in an `std::vector<double>`, rounds them to whole numbers and then saves them again in the same vector. To round numbers use:

```
#include <cmath>

int main()
{
  double x = 2.71;
  double x_rounded = std::round(x);
}
```

(e) Write a program that reverses the order of entries in a vector, and saves the result in the same vector.

(f) Modify the previous program so that it can swap individual entries using the function `std::swap(a, b)`.

8. Recall that the convex closure (or convex hull) of a set of points in say $\mathbb{R}^2$ is defined to be the smallest convex set containing all the points.

   Write a program that calculates the convex hull of a set of points. It should read the points in the following format from the standard input:

```
4
0.3  0.7
1.2  3.4
9.3  4.8
2.8  7.2
```

   The first line indicates the number of points and subsequent lines give the x and y coordinates of the points. On the web you will find an example file in that format:
   `https://conan.iwr.uni-heidelberg.de/data/teaching/ipk_ws2017/convex-hull-points.txt`

   Test your program on the file, using the input redirection:

```
./convex-hull < convex-hull-points.txt
```

   (a) Each of the points should be stored in an `std::array` of size 2.

   (b) Write a function `read_point()` that reads two double from `std::cin` (the redirected file), and returns such an `std::array`.

   (c) Write a function `read_problem()` that reads the number of points from `std::cin` fills an `std::vector` of arrays with that many points by using the function `read_point()`, and returns that vector.

   (d) Calculate the convex hull of the points. You can do so by using one of the algorithms on the Wikipedia page for convex hull, most simply the Graham-Scan. For that you need to determine whether a point C is left or right of the line through two other points A and B. That you can do with the relation:

$$\begin{vmatrix} x_B - x_A & y_B - y_A \\ x_C - x_A & y_C - y_A \end{vmatrix} = \begin{cases} < 0 & C \text{ is to the right of } AB \\ = 0 & C \text{ is on } AB \\ > 0 & C \text{ is to the left of } AB \end{cases}$$

(e) Write the indices of the points on the convex hull to the standard output in the same format as the input (described above). Since indices are 0-based, the matching coordinates in the vector read in at the beginning can be looked up. The points themselves must be sorted in ascending order either clockwise or counterclockwise on the boundary of the convex hull.

To find out if your program is working properly you can download a small Python script from the website
`https://conan.iwr.uni-heidelberg.de/data/teaching/ipk_ws2017/plot-hull.py`
which allows you to plot the calculated convex hull as a PDF file.

For that script you must have the python packages `numpy` and `matplotlib` installed. After that you can test your convex-hull program in the following way:

```
./convex-hull < input.txt > output.txt
python plot-hull.py input.txt output.txt
```