Programming Project Pi
CPSC-298-6 Programming in C++
jbonang@chapman.edu

Introduction

In this assignment, you'll use the Leibnitz formula to approximate the value of π (pi), the ratio of the circumference to the diameter of a circle.

The Leibnitz formula for π,

$$\frac{\pi}{4} = \sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1}$$

is an alternating series:

$$1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \cdots = \frac{\pi}{4}$$

The value of π is approximately

3.1415926535897932384626433832795028841971693993751058209749445923

Of course, the digits go on forever.

We can't carry out the expansion to infinity in C++ but we can approximate π using a large value for the upper bound of k.

In the first part of the assignment, you'll write a program that uses a single for loop implementing a truncated Leibnitz formula to approximate the value of pi using an upper bound for k entered by the user.

In the second part of the assignment, you'll reuse your earlier code but encapsulate it in an outer for loop that increases the upper bound of k and observe the computed value for pi approach the actual value.

These programs appear deceptively simple and don't take long to write. However, there's more to them than meets the eye.

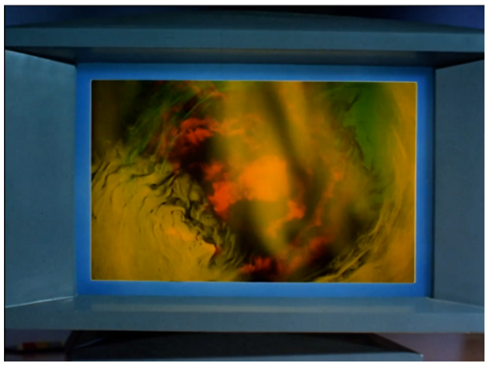
(The Background section that follows is optional; you can skip to the assignment. However, it may contain information that will save you time. It might, just might, be interesting, too.)

Background

Pi in Sci Fi

Computer viruses, a form of malware, began to appear in the early 1970s. The earliest of these, Creeper, was a legitimate experiment in self-replicating programs; it didn't cause any damage beyond printing the message "I'm the creeper; catch me if you can!" as it propagated across the ARPANET, the forerunner of the Internet. Viruses took on a more sinister tone in the next decade, when the Elk Cloner virus appeared on early Macintosh computers in 1982. Written by a fifteen-year-old, it propagated by means of floppy disks. More destructive viruses appeared in the mid- to late-1980s. The most serious of these, the Morris Worm, took down the ARPANET within 24 hours. It's author, Robert Morris, became the first malware author convicted of a crime. Today, malware is ubiquitous.

Science fiction postulated computers being subverted by malware much earlier. For instance, in the 1967 episode "Wolf in the Fold" of the original *Star Trek* television series, a malicious, incorporeal entity inhabits and gains control of the main computer system of the starship *Enterprise*. Just as in today's spacecraft, the computer manages most of the functions aboard the starship, allowing the malevolent entity to wreak havoc.



Captain Kirk and Mr. Spock devise a plan to force the computer to solve an insoluble mathematical problem. After programming the "duotronic" computer's "compulsory scan unit," Mr. Spock compels it to solve the problem by issuing a voice command:

"Computer, this is a class A compulsory directive; compute to the last digit the value of pi."

He comments to Captain Kirk "As we know, the value of pi is a transcendental figure without resolution."

The malevolent entity reacts much as any undergraduate would when given the assignment: "No, no, nooooooo!"

The computation continues, consuming ever more memory, until the malevolent entity can no longer tolerate it and flees the computer.

Now, most of us would be happy if we had *only* one malevolent entity in our computer. Nonetheless, what, exactly, did Mr. Spock do?

The Leibnitz Formula and Summation Notation

Mr. Spock likely used the Leibnitz formula for π,

$$\frac{\pi}{4} = \sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1}$$

which is an alternating series:

$$1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \cdots = \frac{\pi}{4}$$

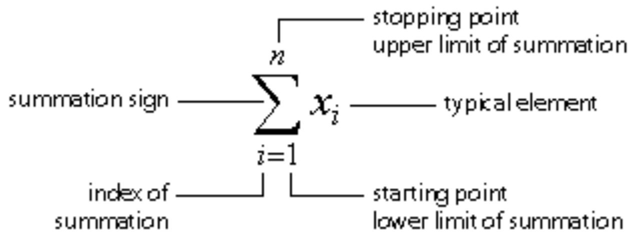
He also probably used arbitrary precision arithmetic. Instead of being limited to the precision available in 32-bit (float) or 64-bit (double) floating-point values, he programmed the computer to use much greater precision. This is actually available in C++ through the Boost Multiprecision Library (https://www.boost.org/doc/libs/1_77_0/libs/multiprecision/doc/html/boost_multiprecision/intro.html).

However, we're going to use double precision for this assignment. (Students entering the data analytics field or computer scientists or electrical engineers writing numerical analysis programs may want to keep the Boost Multiprecision Library in mind. Other programming languages have similar libraries.)

The Leibnitz formula uses summation notation. Let's review that.

Summation Notation

Often mathematical formulae require the addition of many variables. Summation or sigma notation is a convenient shorthand used to give a concise expression for a sum of the values of a variable. Here's the general form:



$$\sum_{i=1}^{n} x_i$$

Notice that the notation says that i goes from 1 to n but doesn't indicate what the increment is; it's assumed to be 1.

In the Leibnitz formula, k is used as the index of summation rather than i. Additionally, k begins at 0 rather than 1 (we computer scientists like that). The stopping point for the Leibnitz formula is infinity,∞. We can't add up an infinite number of terms so we'll need to rewrite the equation,

$$\frac{\pi}{4} = \sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1}$$

as an approximation:

$$\frac{\pi}{4} \approx \sum_{k=0}^{n} \frac{(-1)^k}{2k+1}$$

Infinity is replaced by n, a constant upper summation limit and the equal sign is replaced with an approximately equal sign, ≈.

The x_i in the Liebnitz formula is the term

$$\frac{(-1)^k}{2k+1}$$

where, as mentioned earlier, k is used instead of i as the index of summation.

Our approximation,

$$\frac{\pi}{4} \approx \sum_{k=0}^{n} \frac{(-1)^k}{2k+1}$$

expands to:

$$\frac{\pi}{4} \approx \frac{(-1)^0}{2(0)+1} + \frac{(-1)^1}{2(1)+1} + \frac{(-1)^2}{2(2)+1} + \frac{(-1)^3}{2(3)+1} + \frac{(-1)^4}{2(4)+1} + \cdots + \frac{(-1)^n}{2(n)+1}$$

Suppose we choose n to be 5,

$$\frac{\pi}{4} \approx \sum_{k=0}^{5} \frac{(-1)^k}{2k+1}$$

then our approximation becomes

$$\frac{\pi}{4} \approx \frac{(-1)^0}{2(0)+1} + \frac{(-1)^1}{2(1)+1} + \frac{(-1)^2}{2(2)+1} + \frac{(-1)^3}{2(3)+1} + \frac{(-1)^4}{2(4)+1} + \frac{(-1)^5}{2(5)+1}$$

which reduces to:

$$\frac{\pi}{4} \approx 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11}$$

Translating Summation Notation to C++ Code

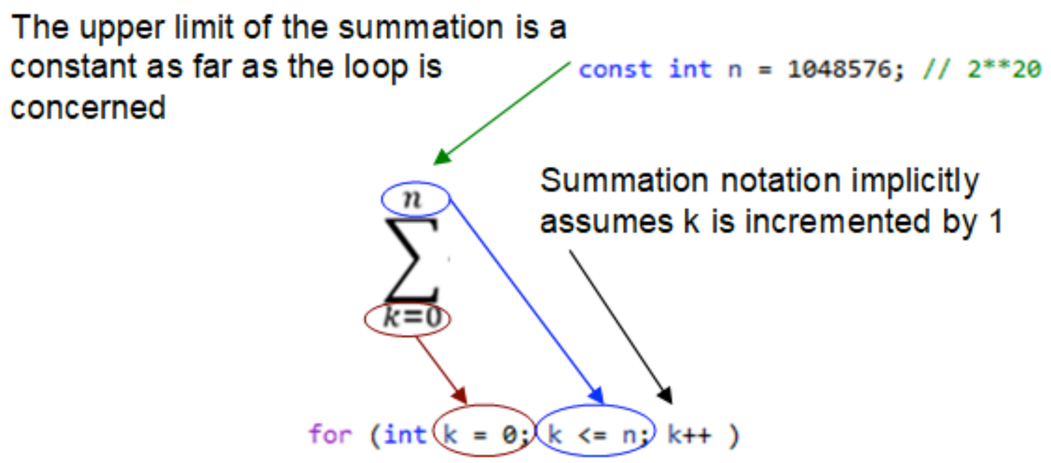
How does summation notation translate to C++?

$$\frac{\pi}{4} \approx \sum_{k=0}^{n} \frac{(-1)^k}{2k+1}$$

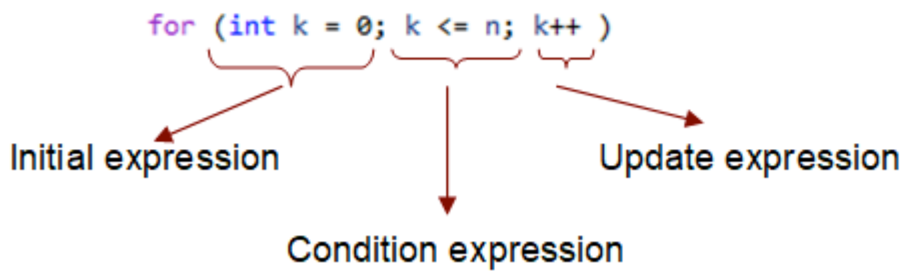
The upper limit of the summation, n, could simply be a constant, 5, say, or 1048576 (which is 2^{20}), in C++:

```
const int n = 1048576; // 2**20
```

The sigma character represents a summation and maps to a C++ for loop. The index of the for loop is k, the upper bound on the for loop is n and the increment (implicit in the summation notation) is 1.


The upper limit of the summation is a constant as far as the loop is concerned
const int n = 1048576; // 2**20
Summation notation implicitly assumes k is incremented by 1
$\sum_{k=0}^{n}$
for (int k = 0; k <= n; k++)

The condition expression for the loop is k <= n. The loop continues to iterate as long as the condition expression is true; the iterations terminate when it is not true. We want n to be "counted", so k <=n is used instead of k < n. The upper limit of the summation is n after all, not n-1. The loop terminates when k is greater than n.



The update expression changes k on each iteration of the loop, incrementing it by one.

Now, we need to accumulate the value for each term as we iterate through the loop. How do we do that?

We introduce a new variable, piOver4, to accumulate the values of each term as the loop iterates.

Our approximation,

$$\frac{\pi}{4} \approx \sum_{k=0}^{n} \frac{(-1)^k}{2k+1}$$

translates to the following pseudo code:

```
const int n = 1048576; // 2**20

double piOver4 = 0.0;

for (int k = 0; k <= n; k++)
{
    piOver4 = piOver4 +    $\frac{(-1)^k}{2k+1}$    ;
}
```

Notice that we haven't yet translated the "typical term", $\frac{(-1)^k}{2k+1}$, to C++ code.

Instead of the assignment operator, as in piOver4 = piOver4 ..., we could use the += addition assignment operator.

```
for (int k = 0; k <= n; k++)
{
    piOver4 +=    $\frac{(-1)^k}{2k+1}$    ;
}
```

Notice that piOver4 is initialized to 0.0 before entering the loop. If it were initialized to something else, such as 1.0, the sum would be incorrect. The accumulator in this case must be initialized to 0.0.

Pow!

For $(-1)^k$, you could use a straightforward approach such as the power function, pow. For example,

pow(-1.0, k)

which raises -1.0 to the k'th power.

To use the pow function, you'll need to include header file cmath.

```
#include <cmath>
```

For the denominator of the typical term, remember that you're computing a floating-point value, so you may want to use 2.0 instead of 2, or 1.0 instead of 1 so that you avoid integer arithmetic.

Output Formatting

Fixed Point Notation for Displaying Floating Point Values

Floating point can be displayed in scientific notation or in fixed point notation.

When displaying our approximations to pi, fixed point notation is more convenient.

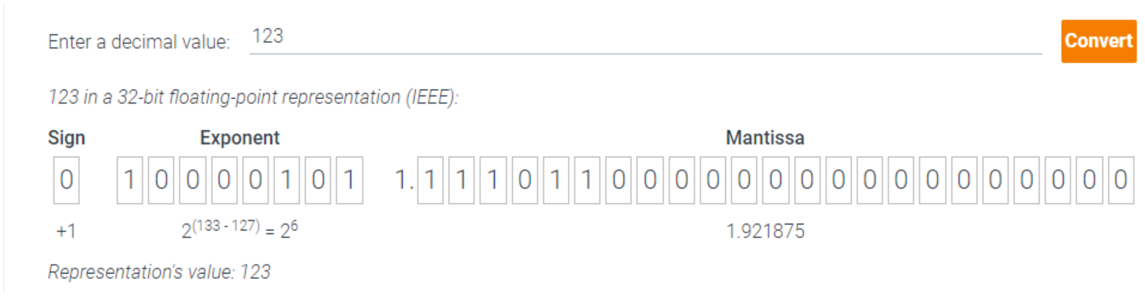
To specify fixed point when outputting floating point values using the std::cout character output stream object, used the std::fixed flag as follows:

```
std::cout << std::fixed;
```

Precision

C++ float and double data types have limited precision. They have only 32 and 64 bits, respectively, to represent floating point numbers. Of these bits, only 24 and 53 are mantissa bits, respectively, (the bits that represent the value), the remainder are the exponent bits and the sign bit.

The bit pattern for decimal 123 is shown in the figure below for a float data type.



123 in a 32-bit floating-point representation (IEEE):

Sign: 0 — +1

Exponent: 1 0 0 0 0 1 0 1 — $2^{(133 - 127)} = 2^6$

Mantissa: 1. 1 1 1 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 — 1.921875

Representation's value: 123

The implicit 1 in front of the 23 mantissa boxes in the figure above counts as a mantissa digit, giving a total of 24 mantissa digits for a float data type.

So we have 24 and 53 bits of precision for floats and doubles. That's the precision in base 2 digits. What about the precision for base 10?

15 decimal places is the precision in decimal digits of a 64-bit C++ double precision floating point value (`double`). It's the value of $\log_{10}(2^{53})$ rounded down, where 53 is the number of mantissa digits in a `double`.

The header file cmath (and cfloat) defines the macro constant DBL_DIG to be the number of decimal digits of precision of a `double`, 15. When setting the precision to be displayed for this assignment, you can pass the DBL_DIG macro constant to the std::setprecision() function. For example,

```
std::cout << std::setprecision(DBL_DIG);
```

The cmath header file also defines the macro constant `DBL_MANT_DIG`, which indicates the number of mantissa digits in a double precision floating point value (`double`), which is 53.

When using the GNU Compiler Collection (GCC) C++ compiler, you may need to include float.h (or, equivalently, cfloat) to obtain the definitions of DBL_DIG and DBL_MANT_DIG.

```
#include <cfloat>
```

The Assignment

The assignment consists of two parts; each part is an independent program.

Part I

In the first part of the assignment, you'll write a program that implements the Leibnitz formula to approximate the value of π using an upper bound for k entered by the user. Call the integer variable for the upper bound n, and the integer variable of summation k. Use a single for loop. Implement this program in a C++ source file named leibnitz_pi.cpp.

Change the Leibnitz formula for π,

$$\frac{\pi}{4} = \sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1}$$

so that it is an approximation and so that it computes π directly, rather than π/4:

$$\pi \approx 4 \sum_{k=0}^{n} \frac{(-1)^k}{2k+1}$$

Represent π as a variable named pi.

When submitting the assignment, enter 1048576, which is 2^{20}, as the value for n, the upper bound for index k.

Print the value of pi out to 15 decimal places and display it in fixed point notation. Use the value `DBL_DIG` defined in header cmath (or cfloat) when setting the precision to 15 decimal places. (See the Output Formatting subsection in the Background section for details.)

After displaying the value of pi to 15 decimal places as approximated by our truncated Leibnitz formula, display the actual value of π to 15 decimal places (it's 3.141592653589793).

Display pi and n on the same line; set the width for n to 7. Print the actual value for π on the following line. Your calls should look similar to the following:

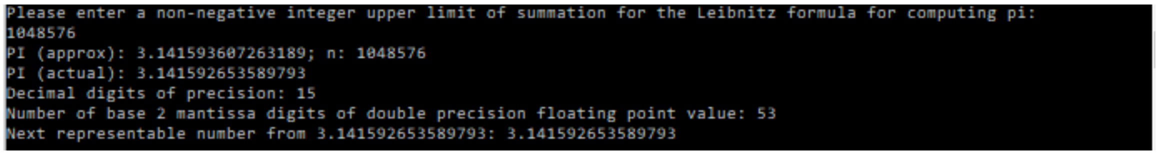
```
std::cout << "PI (approx): " << pi << "; n: " << std::setw(7) << n << std::endl;
std::cout << "PI (actual): " << "3.141592653589793" << std::endl;
```

Next, print out the value of the `DBL_DIG` and `DBL_MANT_DIG` macro constants. `DBL_MANT_DIG` indicates the number of base 2 mantissa digits in a double, based on the IEEE 754 Floating Point standard (it's 53 for `double`).

Finally, use the std::nextafter method to determine if the value for pi to 15 digits could have been represented exactly. This gives some idea of how the limited number of representable floating point values might affect our calculations. Your code should be similar to the following.

```
std::cout << "Decimal digits of precision: " << DBL_DIG << std::endl;
std::cout << "Number of base 2 mantissa digits of double precision floating point value: "
          <<  DBL_MANT_DIG << std::endl;
std::cout << "Next representable number from 3.141592653589793: "
          << std::nextafter(3.141592653589793, 3.14) << std::endl;
```

The output of your program should appear similar to the following screen capture:



```
Please enter a non-negative integer upper limit of summation for the Leibnitz formula for computing pi:
1048576
PI (approx): 3.141593607263189; n: 1048576
PI (actual): 3.141592653589793
Decimal digits of precision: 15
Number of base 2 mantissa digits of double precision floating point value: 53
Next representable number from 3.141592653589793: 3.141592653589793
```

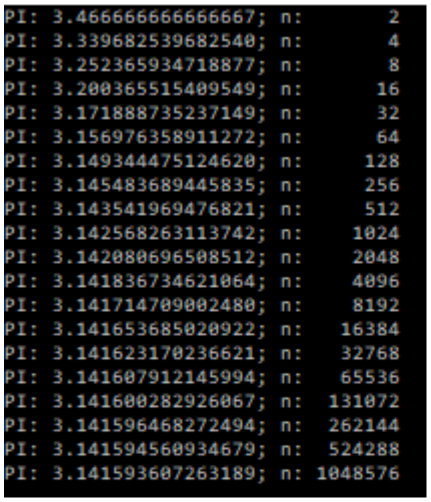
Part II

In Part II you'll reuse much of your code from Part I. Name the source file for this program leibnitz_pi_converge.cpp

Encapsulate the for loop to calculate pi from Part I in another, outer, for loop whose index is n, where n is the upper bound passed to the inner for loop.

n should start at 2 and double in value on each iteration until it reaches 2^{20}, or 1048576.

The output of your program should appear similar to the following.



```
PI: 3.466666666666667; n:       2
PI: 3.339682539682540; n:       4
PI: 3.252365934718877; n:       8
PI: 3.200365515409549; n:      16
PI: 3.171888735237149; n:      32
PI: 3.156976358911272; n:      64
PI: 3.149344475124620; n:     128
PI: 3.145483689445835; n:     256
PI: 3.143541969476821; n:     512
PI: 3.142568263113742; n:    1024
PI: 3.142080696508512; n:    2048
PI: 3.141836734621064; n:    4096
PI: 3.141714709002480; n:    8192
PI: 3.141653685020922; n:   16384
PI: 3.141623170236621; n:   32768
PI: 3.141607912145994; n:   65536
PI: 3.141600282926067; n:  131072
PI: 3.141596468272494; n:  262144
PI: 3.141594560934679; n:  524288
PI: 3.141593607263189; n: 1048576
```

Notice how the value of pi approaches (or converges on) the actual value of π as n increases.

In Part II, you won't need to prompt the user for the value of n.