Math/CS 466/666: Lecture 1

This course consists of a mixture of mathematics and computer science. The goal is to use computers to treat and understand problems in continuous mathematics, often by finding and analyzing approximate solutions to those problems. In general, the term continuous mathematics refers to a crucial use of the continuum of the real numbers in the formulation of the problem. Note that the set of real numbers includes all rational numbers as well as all irrational numbers. The first time most students encounter any theoretical issues related to the real numbers is in the context of Calculus and the continuity of a function. In that case, the issues are somewhat subtle and often overlooked. In the context of computational mathematics, the problems are clear and immediate: the set real numbers is uncountably infinite and therefore can not be accurately represented on any digital computer, which by nature is finite. Before embarking on a theoretical treatment of the rounding errors present in the approximations of real numbers defined by the floating-point hardware in digital computers, we first discuss some engineering aspects of scientific computing.

Computer Literacy and Scientific Computing

In 1909 the Ford Model T ran on gasoline, kerosene or ethanol and could be repaired by the owner. In fact, to be a viable product all parts of these mass-produced cars were user serviceable using ordinary tools. On the other hand, the operation of a Model T required hand-cranking to even start it. If this were not done with the proper skill, a driver might break an arm or get run over. In comparison the Volkswagen Beetle of 1939 was significantly easier to operate but more complex. The typical owner did not repair the Beetle; however, the car was easily serviced by the corner garage. Today's cars include complex safety, entertainment and emission control systems which are controlled by 20 to 30 individual computer processors [6].

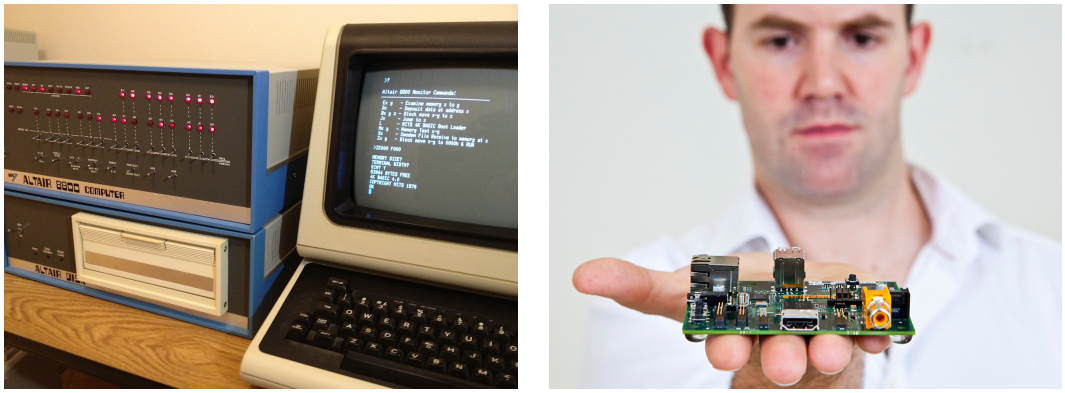
The Ford Model T was mass produced and user serviceable; the Apple II was mass produced and user programmable.

A modern car is literally a rolling computer. Not only is it difficult for the owner to maintain such a car, but the Digital Millennium Copyright Act has been used by manufacturers to make many repairs illegal. Specifically, manufacturers placed encryption in the computer systems because Section 1201 criminalizes the act of understanding this encryption well enough for anyone but the manufacturer to repair the car. In 2015 the Electronic

Frontier Foundation successfully filed for an exemption to the DMCA which allows owners to modify and repair their cars [7]. While this exemption decriminalizes attempts to repair cars, it does not make the repairs themselves any easier. Cars have evolved into complex machines which are easy to operate but have few user-serviceable parts.

In 1977 the Apple II was first introduced. To be a viable product these mass-produced computers were user programmable using a built-in Basic interpreter. In particular, operation of the Apple II required programming skills just to get started. Although programming doesn't directly lead to the difficulties of a broken arm or getting run over, as with automobiles, computers have changed over time to become easier to use. At the same time computers have become more and more complex to understand. Today, the typical owner does not program the computer; however, custom software can still be written by skilled users and experts. Future computers may not be programmable at all but instead via secure boot only run programs downloaded from a vendor approved software library such as the Microsoft Windows Store, the Google Play Store or the Apple iTunes Store.

The ability to read and write a person's native language is referred to as literacy. Computer literacy, by analogy, logically refers to the ability to read and write a computer's native language, that is, to read and write computer programs. In 2006 faculty at Cambridge University noticed a decline of computer literacy among incoming students who wanted to study computer science—fewer and fewer had any programming experience. Imagine the win-win situation in which ordinary people don't have to learn reading and writing and instead a whole class of high-paying jobs are created for professional scribes to do the reading and writing for them. This is the situation with computer literacy achieved by commercial software companies. Unfortunately, it is difficult to train new software engineers if nobody has prior experience reading and writing computer programs. Rather than water down their degree program, the faculty hatched a plan to increase coding skills and algorithmic thinking for new students over the summer.



Microsoft Basic first ran on the Mits Altair 8800; the Raspberry Pi was created to reintroduce programming into the public schools.

In 2012 a $35 computer called the Raspberry Pi was created to familiarize students with programming before they arrived at the university. This simple computer was designed to remove the barriers of complexity that were preventing young people from learning how to program computers. The original plan was to use an updated version of RiscOS with BBC Basic from the 1987 Acorn Archimedes. BBC Basic is a block-structured pro-

gramming language developed for the 8-bit BBC Micro in 1981 and largely responsible for the initial period of computer literacy in the United Kingdom. Since the processor used in the Raspberry Pi was reverse compatible with the Acorn RISC Machine used in the Archimedes, this was a natural idea; however, the same processor is also used in Linux-based Android smart phones. In the end the Raspberry Pi was not a retro project but instead shipped with a modern Unix-like operating system based on the Linux kernel.

Unix was developed in 1971 by Thompson, Ritchie, Ossanna and Canaday at Bell Labs and publicly released in 1975. Because U.S. Federal Law prevented Bell Labs from selling products due to its status as a monopoly institution, Unix was made available at no cost with complete source code. In 1982 the Department of Justice broke AT&T into several firms. Corporate interests combined with intellectual property laws soon made the original Unix system very expensive. As a remedy, in 1984 Stallman began the GNU project to create a Unix-like operating system with a license that guaranteed individual programmers the right to extend and modify the software as needed [10]. The remaining kernel needed to complete the GNU operating system was provided by Torvalds in 1991 at the University of Helsinki and called Linux [8]. The Linux kernel consists of about 15 million lines of code while the rest of the GNU operating system adds another 15 million lines. A typical distribution includes an additional 40 million lines devoted to mouse interfaces, web browsers and office applications.

The first public release of Unix ran on the DEC PDP-11; almost all supercomputers run a Unix-like operating system called Linux.

Numerical computation, just like computer programming in general, should not be a nostalgic trip down memory lane, but a modern discipline relevant to current practices in science, technology, engineering and mathematics. The reasons for using Linux in this course are similar to those which motivated the Raspberry Pi Foundation. According to Kernighan and Pike [5], "The Unix programing environment is unusually rich and productive." More than 30 years later the same can be said about the Unix-like programming environment provided by GNU/Linux. As of June 2017 more than 99 percent of the world's top 500 supercomputers run a Linux based operating system [12]. Linux is also widely used for scientific computing on personal computers. Although less relevant for our purposes, it is interesting to note that 87.5 percent of the current smart phone market consists of Android which is based on Linux. Linux also plays a central role in cloud computing.

While GNU/Linux is the clear operating system choice for numerical computing, the

choice of programming language is less obvious. In the beginning almost all numeric codes were written in FORTRAN. FORTRAN, short for formula translator, was developed in 1954 by John Backus at IBM and originally called the IBM 701 Speedcoding System. Though it is not quite the first high-level programming language, it is surely the most successful. Some of the success of FORTRAN comes from tradition. Large libraries of existing well debugged code make it convenient to write new programs. On the other hand, the C programming language plays the unique role of system programming language on Unix and Unix-like systems. In particular, the operating system kernel, device drivers, supporting libraries, windowing system and many of the user-space programs such as `ls`, `cat` and `grep` are written in C.

The original release of Unix from Bell Laboratories included a FORTRAN compiler which was written in C and used the same code generator as the C compiler [11]. New features have been added to FORTRAN only with careful consideration on how they will affect the automatic optimization of numeric codes. Fortran 90 supports vector operations, operator overloading and modules which contain both data and code. Fortran 2000 supports object oriented programming in a way similar to Java and C++. OpenMP and MPI add symmetric and distributed parallel processing capabilities. At the same time C has also been kept up to date with the addition of type-safe prototypes in 1989, stack-allocated variable-length arrays, native complex numbers and aliasing hints to allow automatic vectorization in 1999. The C11 standard includes Unicode variable names while the MIT/Intel CilkPlus extensions add vector operations and a convenient way to do SMP parallel programming. As with FORTRAN, parallel processing is also available by using OpenMP and MPI. Although, modern FORTRAN continues to be a relevant language for numerical programming [8], it is not widely used for other purposes.

This course will focus on C because it is more generally used in the software industry and because modern versions are just as good as FORTRAN from a suitability, availability and performance point of view. It is difficult to talk about C without mentioning the object oriented programming language C++ taught in the introductory computer science classes. Unfortunately, C++ has evolved along different lines than C and currently lacks stack allocated arrays and native complex data types. Fortunately, the languages are close enough that it is not difficult to switch between them. Other C-like programming languages include C#, CUDA, D, Go, Java, OpenCL, Kotlin and Swift. Each has its own merits and drawbacks.

So far we have discussed various computer-engineering aspects of computational mathematics, scientific and high-performance computing that explain why GNU/Linux and the C99 version of the C programming language were chosen for this class. While the abstract theory of computational mathematics and computer science are not tied to particular technologies or tools, this course also strives for practical application of theory. We finish by making a hands-on comparison of some alternative technologies and educational approaches. During this tour we shall consider

- whether the programming environment and tools are general purpose enough to apply to fields outside of numerical mathematics,

- whether the tools are easy to use given a student's background,

- whether the tools are useful for solving problems sized to the practical limits of modern multi-core computing systems,
- whether the tools are likely to change or become obsolete in the near future, and
- whether the tools will still be available to every student after graduation.

While important considerations have doubtlessly been omitted, these are enough to focus the discussion below. As individual needs are different, each student is further welcome to draw their own conclusions.

For a hands-on comparison, we need a problem in continuous mathematics to solve. Consider the harmonic series. A simple Calculus argument shows that

$$\ln(1+n) \leq H_n \leq 1 + \ln n \qquad \text{where} \qquad H_n = \sum_{k=1}^{n} \frac{1}{k}.$$

Our discussion problem is to write a program to compute the Euler–Mascheroni constant

$$\gamma = \lim_{n\to\infty} (H_n - \ln n) \approx 0.577215664901532$$

directly from the definition. There are significantly more efficient ways to approximate γ, for example, by using Ramanujan's formula

$$\gamma \approx H_n - \frac{\ln n + \ln(n+1)}{2} - \frac{1}{6n(n+1)} + \frac{1}{30n^2(n+1)^2} - \cdots. \tag{1}$$

We do not pursue such methods here.

All programs will be run on the same hardware: an Intel Xeon E5-2620 based system with 12 cores and 24 threads clocked at 2.4 Ghz. This hardware is representative of a single computational node in a supercomputer built around 2015. We now compare how different programming tools can be used to solve this problem while reflecting upon the five considerations outlined above. To actively understand these considerations, please follow along using a lab computer, a home computer or a laptop. Note that individual computers will run faster or slower depending on their exact specifications; however, the relative differences between tools should be similar.

The C Programming Language

As a point of reference let us start with the C programing language. The following program computes an approximation for gamma.

```
1 #include <stdio.h>
2 #include <math.h>
3
4 int main() {
5     int n=2000000000;
6     double hn=0.0;
7     for(int k=n;k>=1;k--) hn+=1.0/k;
```

```
8     printf("gamma(%d) = %.14f\n",n,hn-log(n));
9     return 0;
10 }
```

Before proceeding, a few comments on the code are in order. First, lines 1 and 2 tell the compiler that the input/output library and the math library will be used in the program. The loop in line 7 is done in reverse order to reduce rounding error in the floating-point arithmetic. Finally, line 9 returns a value of 0 to the operating system to indicate the program finished successfully. Now, compile and run the code with the commands

```
$ gcc -O2 -o sn0 sn0.c -lm
$ ./sn0
```

to produce the output

```
gamma(2000000000) = 0.57721566514962
```

which is correct to 8 digits.

The compilation command is cryptic and one of the difficulties when writing C programs. The `-O2` flag indicates to optimize the code by performing a data flow analysis and other mathematical transformations to increase the resulting execution speed, the option `-o sn0` tells the compiler to create an executable output file called `sn0` and `-lm` tells the compiler to link with the mathematics library. To avoid repetitive typing a `Makefile` is often used to automate the compilation of C programs. The command

```
$ time ./sn0
```

produces the additional output

```
real    0m9.018s
user    0m9.016s
sys 0m0.000s
```

which indicates the program performs at a speed of

$$2n/T = 2 \times 2000000000/9.018 = 443557329 \text{ FLOPS} = 443.6 \text{ MFLOPS}$$

on our test machine. Here FLOPS stands for floating point operations per second. Since each iteration through the loop performs a division followed by an addition, the speed in FLOPS may be obtained by dividing $2n$ by the time T in seconds. Note that the load and initialization time, the time to compute the logarithm and the output time are included in the results. If you are following along hands-on, please compute and record the FLOPS using your own computer or one of the lab computers.

Due to differences in rounding when running the program on different hardware with different optimizations, the last four digits of the approximation for γ may differ, but those weren't significant anyway. Increasing n would make the computation take longer. Note also that larger values of n would result in more rounding error because of more floating point operations as well as a greater loss of precision when $\log n$ and H_n are subtracted from each other in the end because they are larger when n is larger. As mentioned in the

statement of the problem, the obvious way to improve the speed and accuracy is by using mathematical analysis and a different formula. As this is a comparison of programming tools, we instead write a parallel version of the code. The Intel/MIT CilkPlus extensions to the C programming language are convenient for this purpose. The parallel code is

```
1 #include <stdio.h>
2 #include <math.h>
3 #include <cilk/cilk.h>
4
5 double Hser(int p,int q){
6     double hn=0.0;
7     for(int k=q;k>=p;k--) hn+=1.0/k;
8     return hn;
9 }
10 double Hpar(int p,int q){
11     if(q-p<20000000) return Hser(p,q);
12     int c=p/2+q/2;
13     double r1=cilk_spawn Hpar(p,c);
14     double r2=Hpar(c+1,q);
15     cilk_sync;
16     return r1+r2;
17 }
18 int main() {
19     int n=2000000000;
20     printf("gamma(%d) = %.14f\n",n,Hpar(1,n)-log(n));
21     return 0;
22 }
```

While significantly more complicated than the original program, it is, in fact, simpler than most parallel programs. A recursive parallel approach has been taken that divides the problem of computing one sum into two smaller sums

$$\sum_{k=p}^{q} \frac{1}{k} = \sum_{k=p}^{c} \frac{1}{k} + \sum_{k=c+1}^{q} \frac{1}{k} \qquad \text{where} \qquad c = \left\lfloor \frac{p+q}{2} \right\rfloor$$

that can be computed in parallel until the resulting sums become small enough for one core to efficiently handle. Compiling and running the program yields

```
$ /usr/local/gcc-7.1/bin/gcc -fcilkplus -O2 -o sn0p sn0p.c -lcilkrts -lm
$ time ./sn0p
gamma(2000000000) = 0.57721566515166

real    0m1.006s
user    0m22.024s
sys 0m1.248s
```

The number of FLOPS is now

$$2n/T = 3976143141 \text{ FLOPS} = 3.976 \text{ GFLOPS},$$

which amounts to a factor 8.96 times improvement. As the test machine has 12 cores we conclude the resulting parallel speedup is 75 percent efficient. Loss in parallel efficiency arises when a significant part of the execution time is spent performing serial tasks. In the current case loading and starting the runtime system are serial tasks that are included in the time measurement. If such overhead becomes large in comparison to the total execution time, the apparent efficiency decreases.

Parallel programming techniques are usually employed only when the computational size of the problem is sufficiently large. To mimic solving a larger problem, we switch to 64-bit integers in order to use a larger value of n. The resulting code

```
1 #include <stdio.h>
2 #include <math.h>
3 #include <cilk/cilk.h>
4
5 double Hser(long p,long q){
6     double hn=0.0;
7     for(long k=q;k>=p;k--) hn+=1.0/k;
8     return hn;
9 }
10 double Hpar(long p,long q){
11     if(q-p<20000000) return Hser(p,q);
12     long c=p/2+q/2;
13     double r1=cilk_spawn Hpar(p,c);
14     double r2=Hpar(c+1,q);
15     cilk_sync;
16     return r1+r2;
17 }
18
19 long main() {
20     long n=20000000000;
21     printf("gamma(%ld) = %.14f\n",n,Hpar(1,n)-log(n));
22     return 0;
23 }
```

yields the output

```
$ time ./sn0q
gamma(-1474836480l) = 0.57721566492700
_
real    0m9.173s
user    3m37.556s
sys 0m1.768s
```

Thus, for a larger problem the answer is correct to 10 digits and the resulting speed is

$$4360623569 \text{ FLOPS} = 4.36 \text{ GFLOPS}$$

which translates to a 9.8 times parallel speed up that is 82 percent efficient.

Again, since this is a hands-on discussion, please run the parallel version on your own and record how many FLOPS are achieved. If you are using the Math/CS Linux image in a virtual machine, you may have to allocate more cores to the virtual machine to see any performance increase. If you are using a different distribution of Linux which you installed yourself, please note that CilkPlus is available in `gcc` since version 5.0. However, you may also have to install additional packages to make the `cilkrts` library available.

The BASIC Programming Language

We now examine a programming language that was originally designed for teaching and later became popular with home computers. BASIC was created at Dartmouth college by Kemeny and Kurtz in 1964 to enable students in fields other than science and mathematics to use computers [4]. The language considered here may be traced back to Microsoft's first product, a BASIC interpreter for the MITS Altair in 1974. Much of Microsoft's early business was selling versions of this interpreter that ran on other computers. Gates [3] wrote a letter to hobbyists in 1976 that opens with the words

> *To me, the most critical thing in the hobby market right now is the lack of good software courses, books and software itself. Without good software and an owner who understands programming, a hobby computer is wasted. Will quality software be written for the hobby market?*

While this letter is famous for the ending that accuses Microsoft's customers of being thieves, the part quoted above makes an interesting statement about how important computer literacy is for the effective use of a computer.

In the 1980's many students entering the university who were interested in numerical mathematics or computer science would have had prior experience programming one of the 8-bit home computers of the time such as the Apple II, the PET or the TRS-80. Note that the Z-80 processor, after which the TRS-80 was named, is currently used in most Texas Instruments graphing calculators. Without further programming experience such a student might write a solution that looked like

```
10 n=1000000:hn=0
20 for k=n to 1 step -1
30    hn=hn+1/k
40 next k
50 print using "gamma(#) = #.##############";n,hn-log(n);
60 quit
```

Although working versions of early home computers are now museum artifacts, by chance there is a compatible BASIC interpreter that runs on Linux.

The popularity of Microsoft's BASIC and its expense led to a number of independent implementations with similar semantics. For example, the bwBASIC interpreter considered

here is very close to the original Altair BASIC, but independent of all Microsoft code. It should be noted that BASIC, like C and FORTRAN, has evolved as a language since those early computers. An early structured version of the language was BBC BASIC which also ran on 8-bit microprocessors. Current versions include Microsoft Small BASIC for teaching children and Visual BASIC for general programming. However, unlike with C and FORTRAN, changes to BASIC have resulted in language versions which have little resemblance to or compatibility with the original.

The legacy code necessary for the operation of corporations and industry tends not to be written using programming languages designed for education, still, the lack of stability in such languages is puzzling. The development of educational materials is non-trivial and the need to update teaching resources because of incompatible changes to a programming language an unwelcome chore. Moreover, such changes can cause confusion for beginning students who may be following a tutorial written for the wrong language version.

Programs written in the version of BASIC discussed here started appearing in school textbooks at about the same time the 8-bit computers on which those programs ran were discontinued. Schools in some countries, possibly for financial reasons, continued to teach BASIC until quite recently. From a certain point of view, those schools generated more computer literacy than schools which switched to word processing after computers stopped shipping with a built-in programming language. Whether misfortune or blessing, the days when BASIC programs appeared in school textbooks and in magazines sold at the grocery store are gone. Our aim, however, is not nostalgia but to compare approaches for teaching numerical mathematics using current technology. Technology changes quickly. In some countries even the mathematics curriculum has undergone radical revision every 10 years. A constant stirring of the pot in hope that the soup would improve has left technologically backwards countries with tastier soup and better mathematics education. Stability coupled with consistent incremental improvement rather than radical change is an important aspect of a successful educational system.

Clearly, the built-in versions of BASIC in early home computers were fixed in ROM and unchangeable; however, except for a few retro computing projects, computers themselves have changed. On the other hand, bwBASIC is stable, easy to use and runs on current computers without difficulty. According to the documentation

> *This program was originally begun in 1982 by my grandmother, Mrs. Verda Spell of Beaumont, TX. She was writing the program using an ANSI C compiler on an Osborne I CP/M computer and although my grandfather (Lockwood Spell) had bought an IBM PC with 256k of RAM my grandmother would not use it, paraphrasing George Herbert to the effect that "He who cannot in 64k program, cannot in 512k." She had used Microsoft BASIC and although she had nothing against it she said repeatedly that she didn't understand why Digital Research didn't "sue the socks off of Microsoft" for version 1.0 of MSDOS and so I reckon that she hoped to undercut Microsoft's entire market and eventually build a new software empire on the North End of Beaumont. Her programming efforts were cut tragically short when she was thrown from a Beaumont to Port Arthur commuter train in the summer of 1986. I found the source code to bwBASIC on a single-density Osborne diskette in her knitting bag and eventually managed to have it all copied over to*

a PC diskette. I have revised it slightly prior to this release. You should know, though, that I myself am an historian, not a programmer.

I've searched online for more information, but never found anything further supporting or denying the above history. At any rate, whether or not the story is true, it seems unlikely that bwBasic will incorporate any incompatible changes in the future.

Now, run the program to compare the performance with our reference C implementation using the commands

```
$ bwbasic sn1.bas
```

Here `sn1.bas` is the file containing the source code

```
10 n=1000000:hn=0
20 for k=n to 1 step -1
30    hn=hn+1/k
40 next k
50 print using "gamma(#) = #.###############";n,hn-log(n);
60 quit
```

The output is

```
Bywater BASIC Interpreter/Shell, version 2.20 patch level 2
Copyright (c) 1993, Ted A. Campbell
Copyright (c) 1995-1997, Jon B. Volkoff

gamma(1000000) = 0.57721616490148
```

The results are correct to 6 digits, the program is easy to understand and there were no complicated compilation steps needed before running it.

Although simplicity is often more important than finding an answer quickly, before drawing any conclusions on the suitability of bwBasic for teaching numerical mathematics, it is informative to measure what the performance tradeoff really is. The command

```
$ time bwbasic sn1.bas
```

produces the additional output

```
real    0m12.215s
user    0m12.212s
sys 0m0.000s
```

and indicates the program performs at a speed of

$$2n/T = 2 \times 1000000/12.215 = 163733 \text{ FLOPS} = 0.163733 \text{ MFLOPS}$$

for our test machine. In particular, bwBasic runs 2709 times slower than C. Note that this program would achieve a speed of only 133 FLOPS using one of the early 8-bit computers. If you are following along hands-on, please compute and record the FLOPS using your own computer or one of the lab computers. You may also want to try different versions of Basic if they are available.

Though not illustrated by our simple example, bwBasic and the original version of Microsoft Basic are both missing features such as complex variables, structured flow control and parallel processing capabilities. While many of these difficulties could be overcome, the fact remains that Basic was designed to be a simple teaching language which doesn't scale to solving problems of a practical size. Further evidence for this is how much faster other programming languages perform our test calculation. Any practical computation would require a different language.

The Python Programming Language

Python was developed under a grant entitled *Computer Programming for Everybody* by the Defense Advanced Research Projects Agency of the United States. As such, it's origins are that of a teaching language and it has been promoted by educators as a natural successor to Basic. Textbooks have been written which teach introductory programming, computer science and numerical methods using Python. In fact, the "Pi" in the educational computer called the Raspberry Pi is frequently said to stand for Python.

Python was originally released in 1991, updated to Python 2 in 2000 and to Python 3 in 2008. Python 3 is not backwards compatible with Python 2. This is consistent with the phenomena observed with Basic, that programming languages designed for education frequently experience incompatible changes over time. The Python 2 code

```
1 import math
2
3 n=100000000
4 sn=0
5 for k in range(n,0,-1):
6     sn=sn+1.0/k
7 print "gamma(%d) = %.13f"%(n,sn-math.log(n))
```

solves the discussion problem. The code is complicated by the `import math` in line 1, the use of 0 in `range(n,0,-1)` in line 5 to mean stop at 1 and the need for a `math` prefix when calling the logarithm function in line 7. Although more complicated than Basic, since there is no need to compile anything it is still simpler than C. The command

```
$ time python sn2.py
```

yields the output

```
gamma(100000000) = 0.5772156699011
_
real    0m16.078s
user    0m14.380s
sys 0m1.696s
```

Thus, Python achieves a speed of

$$12439358 \text{ flops} = 12.4 \text{ Mflops}$$

which is 35.8 times slower than C but 75.7 times faster than bwBasic.

Python overcomes many of the technical issues with BASIC. In particular, it supports structured object-oriented programming and complex numbers. It is also useful for numerical work that involves high-precision arithmetic with greater than 100 digits. On the other hand, it does not support parallel processing and the serial performance is so much slower than C that parallel processing wouldn't help anyway. The fact that Python is 35.8 times slower than C is significant. One would have to go back in time almost 20 years to find Pentium II computers which ran the C code that slow.

Python has been chosen by many for teaching and has been also used for a number of practical applications. One saving feature about Python is that it is possible to call C and FORTRAN subroutines when additional performance is needed. From another point of view it seems strange to learn Python if you have to write the computational part of the code in C anyway. Many, but perhaps not all, of the programming assignments in this course could be done in Python and you are free to do so if you desire.

The Matlab and Octave Programming Languages

Two of the best known FORTRAN subroutine libraries are EISPACK for solving numerical eigenvalue problems and LINPACK for solving systems of linear equations. MATLAB, short for matrix laboratory, was designed at the University of New Mexico by Moler in 1979 to enable students to use LINPACK and EISPACK without having to learn FORTRAN. MATLAB was commercialized in 1984 and as of 2017 costs $2150.00 for a standard single-user license. Restricted licenses for student and home hobbyist use are available at substantially reduced prices. The university has a site license for the DataWorks Remote Desktop and the UNR Grid computing cluster, but not for the individual computing labs.

Over time MATLAB has expanded from being a teaching language to include rapid prototyping along with post processing and data analysis capabilities. Although new revisions have introduced reverse compatibility issues with certain packages such a graphing tools, the language itself has been relatively stable. Half of all currently available introductory texts on numerical methods include code examples written in MATLAB. Many scientists and engineers use MATLAB for research and development in their work. The popularity of MATLAB in both industry and education has led to a number of independent implementations and similar programing languages of which Octave is closest to the original. Note, however, that Octave is not as computationally efficient as MATLAB. In either MATLAB or Octave our discussion problem may be coded as

```
1 n=10000000;
2 sn=0;
3 for k=[n:-1:1]
4     sn=sn+1.0/k;
5 end
6 display(sprintf("gamma(%d) = %.13f",n,sn-log(n)))
```

with output given by

```
$ time octave -q sn3.m
gamma(10000000) = 0.5772157149016

_
```

```
real    0m13.252s
user    0m12.676s
sys 0m0.372s
```

Thus, Octave achieves a speed of

$$1509206 \text{ FLOPS} = 1.5 \text{ MFLOPS},$$

which is 8.3 times slower than Python but still faster than bwBASIC.

To compare Octave with the commercial version of MATLAB the same program was run using the DataWorks Remote Desktop. The DataWorks lab is powered by Intel Xeon E5-2600 processors clocked at 2.7 GHz so the processors are similar but not exactly the same as our test machine. The timing results are

```
>> tic; sn3; toc
gamma(10000000) = 0.5772157149016
Elapsed time is 0.372958 seconds.
```

Thus, MATLAB achieves a speed of

$$53625341 \text{ FLOPS} = 53.6 \text{ MFLOPS}$$

which is 35.7 times faster than Octave but still almost an order of magnitude slower than the C language version. We have neglected the startup time of the MATLAB program by launching the script from within an interactive session. We have done this, in part, because the graphical user interface takes so long to load that it would have been unfair to count that time as part of the total. Note that the same technique with Octave would improve the measured run time by only about 4 percent.

Originally all variables in MATLAB were matrices. Attempts to introduce strings, data structures and object oriented programming have led to a language still not suitable for general purpose programming. At the same time, the appearance of MATLAB code in numerical analysis texts along with the stability of the language and its ease of use recommends it. As MATLAB is not licensed for use in the computing labs except through remote desktop, the only alternative is Octave. Octave is slow and not as complete; however, on occasion we will make use of the built-in matrix operations.

The Maple Programming Language

Maple was developed starting 1980 at the University of Waterloo and commercialized four years later. Originally designed as a computer algebra system, it has since expanded to include numeric capabilities. The price for Maple is similar to MATLAB, again with substantial discounts for student and home hobbyist use. The UNR site license allows installation and use of Maple on any university owned computer used on campus. The DataWorks Remote Desktop can be accessed from outside the university, so Maple is not installed there. As we will make use of Maple in class for the computer algebra features, it is tempting to use it for the numerics as well. The discussion problem in Maple looks like

```
1 kernelopts(printbytes=false):
```

```
2 Digits:=15:
3 n:=10000000:
4 hn:=0:
5 for k from n by -1 to 1 do
6     hn:=hn+1.0/k
7 end:
8 printf("gamma(%d) = %.14f\n",n,hn-log(n));
```

Line 1 prevents the interpreter from periodically writing memory usage statistics to the output. In line 2 the precision of the floating-point arithmetic is set to 15 significant digits, because this is equivalent to the double-precision standard used by the other programming languages. This value could be set to 32 for quad precision or much more for very high-precision calculations. Interestingly, being able to rerun a numerical algorithm using different precision arithmetic can be quite useful when checking convergence and correctness of results. At the same time, double precision is sufficient for most scientific computation and fast because it is implemented in the floating-point hardware.

Like other interpreted languages, running a Maple program doesn't require any additional compilation step. The command and resulting output

```
$ time maple -q sn4.mpl
gamma(10000000) = 0.57721571490660

_
real    0m14.556s
user    0m14.388s
sys 0m0.040s
```

indicate that Maple achieves a speed of

$$1374003 \text{ FLOPS} = 1.4 \text{ MFLOPS}$$

which is about the same as Octave.

Maple can certainly be used for numerics. Sometimes it is the only programming language available, such as, for example, in the UNR Math Center. Other times, doing everything with Maple is convenient because the task at hand requires mixing computer algebra with simple numeric computation. On the other hand, since Maple is neither free nor available on the DataWorks Remote Desktop server, it is difficult for some students to use Maple for homework. In particular, a different computer algebra system originally developed at IBM called FRICAS has been installed on the Math/CS 466/666 Linux image in place of Maple for doing homework using VirtualBox on your home computer.

The Kotlin Programming Language

Kotlin is a new programming language for the Java Virtual Machine. Sun Microsystems released Java in 1995 and soon afterwords multiple groups began to explore its use in high-performance computing. Originally designed for client-server web applications, Java has become popular in business as a replacement for COBOL and is also the primary language used to write applications that run on Android phones. Java tends to be too

complicated and verbose for many people, especially beginners and doesn't conveniently support complex arithmetic. These problems and others have been addressed by the developers of Kotlin. In Kotlin our discussion problem may be written as

```
1 fun main(args: Array<String>){
2     val n = 2000000000
3     var hn = 0.0
4     for(k in n downTo 1) hn += 1.0/k
5     println("gamma(%d) = %.14f".format(n,hn-Math.log(n.toDouble())))
6 }
```

This code is as simple as any of the other languages compared so far and much less verbose than the Java equivalent. The only complication is taking the logarithm of an integer in line 5 where the conversion of the integer to a double precision must be made explicit and the logarithm called with a `Math` prefix similar to Python. While an extension function could be defined to make the notation more concise, that is not the default and would have added more code to the example than necessary. Compile and run the program by typing

```
$ kotlinc -include-runtime sn5.kt -d sn5.jar
$ time java -jar sn5.jar
gamma(2000000000) = 0.57721566514962

_
real    0m11.760s
user    0m15.372s
sys 0m4.048s
```

Thus, Kotlin achieves a speed of

$$340136054 \text{ FLOPS} = 340.1 \text{ MFLOPS}$$

on our test machine, which is nearly as fast as C.

The fact that Kotlin comes this close to C is remarkable considering the above timings include the just-in-time compilation of the Java byte code for the discussion problem as well as the load and startup time for the Java runtime environment. Kotlin includes parallel processing capabilities which perform reasonably well but are considered experimental at this time. The underlying thread pools do not employ the technique of work stealing used in CilkPlus. Therefore it is possible for all threads to block even though there are still scheduled tasks that could make progress. Although Kotlin doesn't have a built-in complex variable type, it allows defining a complex data type using object oriented programming techniques and operator overloading. Unfortunately, when working with arrays of complex numbers the resulting boxing and unboxing operations cause a two to three fold decrease in performance. For this hands-on discussion, please compute the FLOPS for Kotlin using your own computer or one of the lab computers.

The Julia Programming Language

Like Kotlin, Julia is a relatively new programming language. Work on Julia was started in 2009 by Bezanson, Karpinski, Shah and Edelman at MIT. Their idea was to create a

better MATLAB. Julia is better for us in two ways: free and faster. The simplest code in Julia to solve the discussion problem is given by

```
n=100000000
hn=0.0
for k in n:-1:1
    hn+=1.0/k
end
@printf("gamma(%d) = %.14f\n",n,hn-log(n))
```

Unfortunately, this code runs 28 times slower than the more complicated code in which a function has been defined to perform the calculation.

```
function Hser(p,q)
    hn=0.0;
    for k in q:-1:p
        hn+=1.0/k
    end
    return hn
end
n=1000000000
@printf("gamma(%d) = %.14f\n",n,Hser(1,n)-log(n))
```

Typing the command

```
$ time julia sn6s.jl
```

results in a speed of

$$395961195 \text{ FLOPS} = 395 \text{ MFLOPS}.$$

The difference in performance between the two codes is confusing. Isolating the computation as a function call apparently allows the just-in-time compiler to work more efficiently. Hopefully this issue will be fixed in upcoming versions.

Julia has built-in parallel programming tools which work for both distributed and shared memory systems. In this case, the program needs to be divided into a library and a driver. The library is `sn6l.jl` with contents

```
module sn6l
export Hser, Hpar

function Hser(p,q)
    hn=0.0;
    for k in q:-1:p
        hn+=1.0/k
    end
    return hn
end
function Hpar(p,q)
    if q-p<20000000
```

```
13          return Hser(p,q)
14      end
15      c=trunc(p/2+q/2)
16      r1=@spawn Hpar(p,c)
17      r2=@spawn Hpar(c+1,q)
18      return fetch(r1)+fetch(r2)
19  end
20
21  end
```

and the driver is `sn6p.jl` with contents

```
1  using sn6l
2  n=2000000000
3  @printf("gamma(%d) = %.14f\n",n,Hpar(1,n)-log(n))
```

The parallel Julia code may be run using the command

```
$ time julia -p12 -L sn6l.jl sn6p.jl
gamma(2000000000) = 0.57721566515166

_
real    0m6.006s
user    0m22.720s
sys 0m4.788s
```

The resulting speed is

$$666000666 \text{ FLOPS} = 666 \text{ MFLOPS},$$

which, despite its beastly appearance, is less than double the performance of the serial code and represents a use of the 12 processor cores that is only 14 percent efficient.

Because of its generality, there is significant overhead when starting Julia's parallel runtime environment. Increasing n by a factor of 10 gives the result

```
gamma(20000000000) = 0.57721566492700

_
real    0m15.598s
user    1m59.256s
sys 0m6.552s
```

Thus, for this larger problem size Julia achieves

$$2564431337 \text{ FLOPS} = 2.56 \text{ GFLOPS}$$

or in other words, Julia makes 54 percent efficient use of 12 processor cores.

While the resulting speed is somewhere between C and Kotlin, the Julia language has the convenience of an interpreted language and syntax similar to MATLAB for matrix and vector operations. Julia is a special purpose programming language not likely to be used outside the field of scientific computation, but likely a better alternative than Octave. Julia

is undergoing rapid development that frequently breaks compatibility between versions. Nevertheless, it is used for numerical mathematics courses at a number of universities.

The Go Programming Language

Weinberg's Second Law [2] states

> *If builders built buildings the way programmers wrote programs, then the first woodpecker that came along would destroy civilization.*

The introduction of object oriented programming with multiple inheritance now allows programmers to build skyscrapers, also called frameworks.

The last programming language we shall discuss is Go, developed in 2009 at Google by Griesemer, Pike and Thompson. Note that Pike and Thompson were also involved with the development of Unix and C at Bell Laboratories many years ago. Interestingly, Go deemphasizes object oriented techniques by omitting object inheritance and operator overloading. At the same time, Go aims to be a general purpose programming language which solves the problems of concurrency and parallel processing posed by networks of multi-core servers. In Go our discussion problem becomes

```
1  package main
2  import("fmt"; "math"; "os")
3
4  func main(){
5      n:=2000000000
6      hn:=0.0
7      for k:=n;k>=1;k-- {
8          hn+=1.0/float64(k)
9      }
10     fmt.Printf("gamma(%d) = %.14f\n",n,hn-math.Log(float64(n)));
11     os.Exit(0)
12 }
```

Go requires explicit type conversion similar to Kotlin as well as prefixes on calls to library functions that live in separate namespaces. The code is slightly more verbose than the first C example. The program may be compiled and run as

```
$ go build -o sn7 sn7.go
$ time ./sn7
gamma(2000000000) = 0.57721566514962
_
real    0m8.977s
user    0m8.976s
sys 0m0.008s
```

which indicates Go achieves a speed of

$$445583156 \text{ FLOPS} = 445.6 \text{ MFLOPS}.$$

This is, to within limits, exactly the same as the speed of the C code. We now modify the program for parallel execution to obtain

```
1  package main
2  import("fmt"; "math"; "os")
3
4  func Hser(p int64,q int64) float64 {
5      hn:=0.0
6      for k:=q;k>=p;k-- {
7          hn+=1.0/float64(k)
8      }
9      return hn
10 }
11 func goHpar(p int64,q int64, ret chan float64) {
12     if q-p<20000000 {
13         ret<-Hser(p,q)
14         return
15     }
16     r1r2:=make(chan float64,2)
17     c:=p/2+q/2;
18     go goHpar(p,c,r1r2)
19     goHpar(c+1,q,r1r2)
20     r1,r2:=<-r1r2,<-r1r2
21     ret<-(r1+r2)
22 }
23 func Hpar(p int64,q int64) float64 {
24     ret:=make(chan float64,1)
25     go goHpar(p,q,ret);
26     return <-ret
27 }
28
29 func main(){
30     n:=int64(20000000000)
31     fmt.Printf("gamma(%d) = %.14f\n",n,Hpar(1,n)-math.Log(float64(n)));
32     os.Exit(0)
33 }
```

Compile the program and run by typing

```
$ go build -o sn7p sn7p.go
$ time ./sn7p
gamma(20000000000) = 0.57721566492700
_
real    0m9.168s
user    3m36.776s
sys 0m0.000s
```

Thus, for the large problem size, Go has a parallel speed of

$$4363001745 \text{ FLOPS} = 4.36 \text{ GFLOPS}$$

which 9.8 times faster than the serial code and identical in efficiency to the parallel code written in C using the CilkPlus extensions. Note that the resulting code is slightly more complicated than the parallel C code. This is because Go supports a more general form of concurrency that includes the client and server model.

The Go programming language is numerically efficient, stable and feature complete. It includes native complex data types as well as primitives for slicing matrices and arrays in ways that are similar to MATLAB. On the other hand, it is a new language and not nearly as popular as C. Moreover, because C is nearly a subset of C++, students with a C++ background will have an easier time programming in C than in Go. To finish the hands-on discussion, please run both the serial and parallel versions of the Go code and compute the resulting number of FLOPS using your own computer or one of the lab computers.

Concluding Remarks

After trying C, Basic, Python, Octave, Maple, Kotlin, Julia, Go and many not considered here, my conclusion is that C best satisfies the five considerations mentioned earlier. Two other languages stand out: Julia and Go. Because of its newness, Julia is still undergoing rapid and sometimes incompatible development. It is also a special purpose language that is unlikely to be used outside of numerical mathematics. At the same time, Julia is particularly suitable for numerics and we will make occasional use of it in class. Go is a general purpose programming language suitable for numerical mathematics that is in many ways better than C. However, since C is more widely used, knowing C well allows greater opportunities both inside and outside the field of numerical mathematics.

All the languages discussed here except MATLAB are available on our Linux image. You are free to use which ever language you prefer in your class work; however, most of my examples will be in the C programming language. Donald Knuth, sometimes called the father of computer science, is attributed to having said

> *The most important thing in a programming language is the name. A language will not succeed without a good name. I have recently invented a very good name, and now I am looking for a suitable language.*

Although FORTRAN is certainly not the name Knuth was thinking of, some people like it and it is also available in the Linux image as well as ALGOL, C++, Java, Pascal and R. If the Linux image does not contain your preferred programming language please let me know whether it is freely available to install.

As computers have increased in capability, the problems people attempt to solve have become larger. To solve large problems, it is important to use a computationally efficient algorithm as well as a fast computer. Note that the algorithm used in our hands-on discussion for computing the Euler–Mascheroni constant γ was not very efficient. Taking n in the millions only led to 6 or 7 significant digits. While there exist problems for which even the most computationally efficient algorithm known is still not fast enough, finding γ is not one of those. A simple modification based on Ramanujan's formula (1)

leads to an algorithm that makes finding more significant digits possible, even on the first home computers made 40 years ago. In a tribute to those first computers, we express this improved algorithm using bwBasic as

```
10 n=1000:hn=0
20 for k=n to 1 step -1
30    hn=hn+1/k
40 next k
45 r=(log(n)+log(n+1))/2+1/(6*n*(n+1))-1/(30*n^2*(n+1)^2)
50 print using "gamma(#) = #.###############";n,hn-r;
60 quit
```

Running this code immediately yields

```
$ bwbasic sn1r.bas
Bywater BASIC Interpreter/Shell, version 2.20 patch level 2
Copyright (c) 1993, Ted A. Campbell
Copyright (c) 1995-1997, Jon B. Volkoff

_
gamma(1000) = 0.57721566490153
```

The answer is correct to 14 significant digits. Even better algorithms [1] have been developed and used for computing billions of digits of the Euler–Mascheroni constant.

References

1. Richard Brent and Edwin McMillan, Some new algorithms for high-precision computation of Euler's constant, *Math. Comp.*, Vol 34, No 149, 1980.
2. Murai Chemuturi, *Mastering Software Quality Assurance: Best Practices, Tools and Techniques*, J.Ross Publishing, 2009.
3. Bill Gates, An Open Letter To Hobbyists, *People's Computer Company*, Vol 4, March–April 1976.
4. John Kemeny and Thomas Kurtz, *A manual for Basic, the elementary algebraic language designed for use with the Dartmouth Time Sharing System*, Dartmouth College Computation Center, 1964.
5. Brian Kernighan and Rob Pike, *The Unix Programming Environment*, Prentice Hall Software Series, 1984.
6. Steve Mertl, How Cars Have Become Rolling Computers, *The Globe and Mail*, March 2016.
7. Meriel Jane Walssman, Let Us Hack Our Cars, *Wired*, Vol. 01, 2015.
8. Matt Welsh, Lar Kaufman, Matthias Kalle Dalheimer, *Running Linux, Third Edition*, O'Reilly Media, 1999.
9. Lee Phillips, Scientific computing's future: Can any coding language top a 1950s behemoth?, *Ars Technica*, 2014.
10. Sam Williams, *Free as in Freedom: Richard Stallman's Crusade for Free Software*, O'Reilly Media, 2012.

11. S Feldman and P Weinberger, A Portable Fortran 77 Compiler, *Bell Laboratories*, 1978.
12. `https://www.top500.org/statistics/list/` retrieved July 5, 2017.