In science and engineering an important goal is to become a skilled practitioner by doing it yourself. The computer labs provide computational experience related to the analytic theory presented in the lectures.

Solving Linear Systems

This lab is about solving systems of linear equations of the form

$$Ax = b \quad \text{where} \quad A \in \mathbf{R}^{n \times n}, \quad b \in \mathbf{R}^n \quad \text{and} \quad x \in \mathbf{R}^n.$$

Note that A is square and is likely to have an inverse if chosen randomly.

A large amount of scientific and engineering work done by computers amounts to various kinds of linear algebra. For example, systems of linear equations come from the discretization of partial differential equations and are also used in machine learning. Even finding the roots of a polynomial—something that would seem an ideal application for Newton's method—is often better treated as an eigenvalue problem.

The Julia programming language was conceived at MIT in 2009 with numerical computing in mind. The goal was to combine the interactive environment and high-level abstractions available in interpreted languages such as MATLAB and Python with the compiled performance of C and Fortran. Ideally this would alleviate the need for practitioners of numerical methods to know multiple programming languages. All of the experimentation, development, high-performance computing and scientific visualization needed to solve a problem could conveniently be done in Julia.

While creating yet another programming language seems unlikely to remedy the problem that there are already too many, it's difficult to retrofit the new technologies that make Julia possible onto the designs of older programming language. The way dynamic typing works in Python makes that language difficult to compile into fast machine code. As a consequence, just-in-time Python compilers such as Numba and ahead-of-time compilers such as Cython work only with a subset of Python. On the other hand, trying to add an interactive read-evaluate-print loop to Fortran or C also has problems. The syntax of those languages require so much formalism and details that they are inconvenient to use interactively.

Clearly it is possible to make Python just-in-time compiled by removing features or Fortran interactive by simplifying its syntax; however, the results are likely to include historical compromises that detract from the goal of

combining the best of both. Julia was designed from the beginning to be convenient to use interactively subject to the constraint that just-in-time compilation lead to fast machine code.

In achieving the above two goals, Julia has achieved one more: The high-level mathematical abstractions and notation—the built-in operators for vectors and matrices—allow cross-platform performance portability to different types of computing hardware. See, for example,

`https://www.nextplatform.com/2022/01/05/strong-showing-for-julia-across-hpc-platforms/`

and the benchmarking report at

`https://ieeexplore.ieee.org/abstract/document/9652798`

Note that downloading the PDF file from the above link may require authentication through the university library. You are encouraged to download and read these links outside of class.

The notation used by Julia for manipulating vectors and matrices is inspired by MATLAB. However, in order to keep Julia's syntax as logical as possible there are notable differences.

`https://docs.julialang.org/en/v1/manual/noteworthy-differences/`

Unless you are familiar with MATLAB and want to see a summary of the differences with Julia, there is no need to click the above link.

Now start at the beginning with Julia. The left-division operator `\` is used for solving $Ax = b$. To solve the matrix equation

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 3 & 2 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

using Julia as an interactive calculator, type

```
julia> A=[1 0 1; 0 1 0; 3 2 1]
3×3 Matrix{Int64}:
 1  0  1
 0  1  0
 3  2  1
```

```
julia> b=[1,2,3]
3-element Vector{Int64}:
 1
 2
 3

julia> x=A\b
3-element Vector{Float64}:
 -1.0
  2.0
  1.9999999999999998
```

Note the elements in each row of the matrix A are separated by spaces and the rows are separated by semicolons. The elements in the vector b are separated by commas. Finally, x is the approximate solution to $Ax = b$ obtained from the `A\b` Julia command.

The Linpack Benchmark

When A is square and invertible, Julia uses Gaussian elimination with partial pivoting to compute x. This is first algorithm taught in an introductory linear algebra class; however, there is one difference. When performing Gaussian elimination by hand, one generally swaps rows only when the element in the pivot position is zero. When a computer performs the same task it swaps rows each time it chooses a pivot so the magnitude of that pivot is as large as possible. This helps reduce rounding error.

Since 1993 the most powerful computers devoted to science in the world have been ranked by how fast they solve $Ax = b$ as follows: Given a matrix of size $n \times n$, Gaussian elimination requires $(2/3)n^3 + 2n^2$ floating-point operations to arrive at a solution. Divide this quantity by the time T_n in seconds it takes a particular computer to solve the problem using double-precision 64-bit arithmetic. The quotient is a speed defined as

$$\text{FLOPS} = \frac{1}{T_n}\left(\frac{2}{3}n^3 + 2n^2\right).$$

The fastest computers today perform 1×10^{18} FLOPS or floating-point operations per second. This is called exascale supercomputing. Rankings for the fastest 500 supercomputers in the world are listed at

`https://www.top500.org/lists/top500/`

Generally n is taken as large as memory allows. For example, $n = 24\,440\,832$ for the Frontier system at Oak Ridge National Laboratory. For an n that large it takes 4 450 636 GB RAM to even store A. Since the lab computers each have about 16 GB RAM, we will not use such a large n.

In this project, you will use Julia to compute the speed measured in FLOPS of the lab computer and further measure how the time T_n depends on the size $n \times n$ of the matrix A.

The term Linpack refers to a collection of Fortran subroutines created in the 1970s for solving linear algebra problems. Over the years these subroutines have been refactored into two parts.

- A machine-dependent part known as the BLAS (the basic linear algebra subprograms) that perform vector-vector, matrix-vector and matrix-matrix arithmetic.
- A machine-independent part (still called Linpack) which uses the BLAS to solve systems of linear equations, eigenvalue-eigenvector problems and other linear-algebra problems.

Since the computers in the lab are not equipped with a GPU or vector accelerator, Julia uses Linpack with OpenBLAS when performing the left-division `A\b` to solve $Ax = b$. This is an intrinsically parallel task; however, a bit of Linux shell magic is required before starting Julia to specify the number of CPU cores to use. Note that slightly different commands may be required for Microsoft Windows and Macintosh.

Since many CPU cores support simultaneous hardware threads, the parallelism is actually determined by the number of threads. To check know how many threads are available type `nproc` in a terminal. For example,

```
$ nproc
4
```

indicates four threads are available. In this particular case there were two CPU cores each with two threads. The lab computers may have more.

Sometimes, one might want to leave some CPU cores idle for other tasks or users. For the purpose of seeing how fast the lab computer is, we tell OpenBLAS to use all the threads and then start Julia with

```
$ export OPENBLAS_NUM_THREADS=4
$ julia
               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 1.6.7 (2022-07-19)
 _/ |\__'_|_|_|\__'_|  |  Official https://julialang.org/ release
|__/                   |
julia>
```

Note that the `export` command to set the number of threads is specific to each terminal window. So if you open another, you will again need to type

```
$ export OPENBLAS_NUM_THREADS=4
```

to enable use of all available threads. At this point we are ready to solve $Ax = b$ using all threads of all CPU cores.

Since solving $Ax = b$ for large values of n tends to stress the hardware, it can happen that an unexpected hardware malfunction occurs during the computation. It is important to detect such malfunctions. Truthfully speaking, verifying that a computer is functioning correctly is probably the more important use of the Linpack benchmark these days.

To detect errors, it is best to solve a problem to which the answer is already known. To this end, we construct a linear system with a known solution by starting with x the desired answer—usually the vector with all ones—and setting $b = Ax$. Then solving with b on the right side yields an approximation x^* of x. If things go well, $\|x^* - x\|$ should be small.

The program `flops.jl` shown below solves $Ax = b$ for $n = 2000$ and returns the maximum measured speed among five trials at the end.

```
1 # flops.jl
2 using LinearAlgebra
3
```

```
4 n=2000
5 flmax=0
6 println("n = ",n)
7 for i=1:5
8     A=rand(n,n)
9     b=A*ones(n)
10    Tn=@elapsed x=A\b
11    flops=1/Tn*(2/3*n^3+2*n^2)
12    global flmax=max(flmax,flops)
13    println("Tn = ",Tn)
14    println("Error = ",norm(x-ones(n)))
15    println("Flops = ",flops)
16 end
17 println()
18 println("Maximum Flops ",flmax)
```

This program can be run by typing `include("flops.jl")` in the REPL. The output should look similar to

```
julia> include("flops.jl")
n = 2000
Tn = 1.763931903
Error = 4.4429052618528094e-11
Flops = 3.0280836376104326e9
Tn = 0.386692098
Error = 3.2498676403918044e-10
Flops = 1.3812884620500658e10
Tn = 0.381068803
Error = 1.4049416519342912e-10
Flops = 1.4016716381092297e10
Tn = 0.386438117
Error = 1.972178603916605e-10
Flops = 1.3821962944026384e10
Tn = 0.381153727
Error = 3.2211930596149373e-10
Flops = 1.401359334821179e10
```

```
Maximum Flops 1.4016716381092297e10
```

If in doubt that all CPU cores are being used, monitor the system status by running `top` or `htop` in another terminal window. Then increase the value of n until the run of `flops.jl` takes long enough to confirm all CPU cores are active. On a computer with 16 GB RAM a matrix with $n = 20\,000$ is possible. Don't increase n too much or you may run out of memory.

To complete this lab add an additional loop to the `flops.jl` program such that for each value of n given by $n = 1000, 2000, 3000, 4000$ it solves $Ax = b$ five times. Report the maximum measured speed in FLOPS among all of the twenty trails at the end.

Submitting Your Work

A single PDF file should be submitted for grading that contains

- A program the finds the maximum FLOPS among twenty trials.
- The output from running that program.

After debugging and making sure your program runs correctly, you may prepare your submission by typing

```
$ export OPENBLAS_NUM_THREADS=`nproc`
$ julia flops.jl >flops.out
$ j2pdf -o submit03.pdf flops.jl flops.out
```

For convenience, the `export` above sets `OPENBLAS_NUM_THREADS` automatically to the number of threads reported by the `nproc` command. Make sure to use the back quote in the top left corner of the keyboard or it won't work. It is also fine to set `OPENBLAS_NUM_THREADS` manually as described earlier.

Before uploading, check `submit03.pdf` using the PDF previewer with

```
$ evince submit03.pdf &
```

to make sure the FLOPS output looks correct and everything else. Please reboot into Microsoft Windows before leaving the lab.