# Beat the (Linpack) Benchmark! Mathematical Programming with Python

**MATH 2604: Advanced Scientific Computing 4**
**Spring 2025**
**Monday/Wednesday/Friday, 1:00-1:50pm**

https://people.sc.fsu.edu/∼jburkardt/classes/python_2025/benchmark/benchmark.pdf

## 1 Measuring Computer Performance

If you think of a computer as a number cruncher, then it makes sense to evaluate each computer on how fast it can crunch those numbers. If a game machine is not powerful enough, then the player will be frustrated by the stuttering animation. A biomathematician will be impatient if the analysis of a protein folding problem seems to be taking hours or days. Computational scientists seek new methods that will shorten the time required to solve certain problems. Computer vendors want to brag that their machine is the most powerful. The National Science Foundation has funded supercomputing centers across the country, including the Pittsburgh Supercomputing Center, to offer free access to the best computing facilities. What can be "super" about a computer? What speed are we measuring?

We seek ways of estimating computing power, no matter what language or computer or operating system is involved. This is difficult, since a given computer may handle certain types of problems well, but do poorly on others. Things like the amount of memory, the ability to access memory quickly, the clock speed, and the kind of CPU at the heart of the machine all play a role in producing a desired computational result.

A group of numerical analysts decided to create a benchmark, that is, a single problem that any computer should be able to solve. The number of floating point operations, divided by time required to compute the solution, gives a numeric processing rate, known as FLOPS (floating point operations per second), which could be used as a reasonable rating for a certain class of numerical problems.

## 2 A program to solve a linear system

The benchmark problem chosen was the solution of a linear system of the form $A * x = b$. Here, the value $n$ represented the order, or number of variables. This meant that the unknown solution vector $x$ and the right hand side vector $b$ would be vectors of length $n$, while the matrix $A$ would be of dimensions $n \times n$.

Actually, for technical reasons, partly relating to conventions of the Fortran programming language, the matrix was assumed to be stored in a somewhat larger array, of dimensions $lda \times n$, where typically $lda = n+1$. Surprisingly, because of peculiarities of addressing computer memory, choosing this slightly larger row dimension for $A$ could sometimes produce a noticeable improvement in the speed of execution.

In order to construct a linear equation solver that could be employed on any computer, the decision was made to employ the *linpack()* library, written in the Fortran77 language. A compiler for Fortran77 was available for most scientific computing systems, which meant that the same sequence of calculations, no matter where the program was run. The solver was created primarily from a pair of subroutines:

- *dgefa()*: to compute the $PLU$ factors of $A$;
- *dgesl()*: to solve $P * L * U * x = y$ for $x$;

These two subroutines, in turn, called a small number of basic linear algebra subprograms (known as BLAS) to carry out low-level tasks such as vector scaling, row swapping, and finding the maximum entry in a vector.

To save on storage, the memory for the $A$ matrix was overwritten by the LU factors, which the permutation matrix P was stored in an integer vector of length $n$. Moreover, the solution $x$ overwrote the original right hand side vector $b$. Thus, aside from scalar variables, the memory usage for this solver involved an $lda \times n$ real matrix, and two vectors of length $n$, one real and one integer.

The heart of the computation looked something like this:

```
      call matgen ( A, lda, n, b, norma )

      call cpu_time ( t1 )
      call dgefa ( A, lda, n, ipvt, info )
      call cpu_time ( t2 )
      time(1) = t2 - t1

      call cpu_time ( t1 )
      call dgesl ( A, lda, n, ipvt, b, 0 )
      call cpu_time ( t2 )
      time(2) = t2 - t1

      cpu = time(1) + time(2)
```

## 3 Creating a performance rating

If we want to use the information from the benchmark program to determine a performance rating, we need to compute the ratio of work accomplished divided by CPU time elapsed.

When the program executes, it automatically computes the quantity `cpu` for us, measuring the CPU time required by the factor and solve functions that we called.

To compute the ratio, we need to evaluate the amount of work that had been done in the elapsed time. We can imagine that each time we encounter an addition, multiplicaton, or division in the code, a person would have to execute a command on a calculator. We ignore all other activities in the program, such as updating the loop index, and simply count each floating point operation or `FLOP`. For the kind of Gauss elimination and back solution employed by `dgefa()` and `dgesl()` for an $n \times n$ matrix, a good estimate for *op_count*, the number of operations is:

$$\text{op\_count} = \frac{2}{3}n^3 + 2n^2$$

although it's easier to remember the estimate $\frac{2}{3}n^3$.

Therefore, for a given run of the program that took cpu seconds, we can compute a computational rate, known as `FLOPS` (floating point operations per CPU second):

$$\text{FLOPS} = \frac{\text{op\_count}}{\text{cpu seconds}}$$

This number is a measure of the performance of the program. Of course, this number depends on the problem $n$, the operating system, the Fortran77 compiler, the compiler options, the computer model, and other factors that we may want to worry about later.

## 4 Supercomputers and the Linpack benchmark

In 1979, whe the Linpack benchmark was devised, a matrix of size $n = 100$ was considered a large problem, and so this was the size used in benchmarking. Later, it was decided to go to a problem size of $n = 1000$, and later the problem size was allowed to be even larger. Even in 1979, the FLOPS rating for computers tended to be a large number. For example, a Pentium III chip had a clock speed of 750 MegaHertz, that is, 750 million clock ticks per second. Assuming that this chip could perform one floating point operation per clock tick, that would suggest a theoretical top performance limit of

$$\frac{\text{1 floating operation}}{\text{clock tick}} * 750,000,000\,\frac{\text{clock tick}}{\text{second}} = 750,000,000 \text{ FLOPS.}$$

To simplify reports, the term *"MegaFLOPS"* was devised. The Pentium III was said to operate at a theoretical maximum rate of 750 MegaFlOPS. In fact, the actual Linpack benchmark rating for the Pentium III came out to about 138 MegaFLOPS. Around the same time, a computer from the Cray corporation produced a Linpack rating of 549 MegaFLOPS, indicating that it was a much stronger performance for large numerical problems.

As computer performance continued to improve, it was necessary to name even higher levels of performance, in steps of 1,000:

| Rate | $\frac{\text{floating ops}}{\text{CPU second}}$ | First |
|---|---:|---|
| FLOPS | 1 | |
| KiloFLOPS | 1,000 | IBM 704 |
| MegaFLOPS | 1,000,000 | CDC 6600, 1964 |
| GigaFLOPS | $10^9$ | Cray-2, 1985 |
| TeraFLOPS | $10^{12}$ | ASCI Red, Sandia Lab, 1996 |
| PetaFLOPS | $10^{15}$ | Roadrunner, Los Alamos Lab, 2008 |
| ExaFLOPS | $10^{18}$ | Frontier, Oak Ridge Lab, 2022 |

The IBM, CDC and Cray machines were general purpose computers created by private companies, and widely sold as commodities. The higher performance machines were custom built for national laboratories, requiring enormous computer rooms, special air conditioning facitlies, and massive power generators. The FLOPS ratings for all these machines were generated by versions of the Linpack benchmark, and were a source of great pride. Because of changes in scientific computing, especially parallel programming and memory management, the new versions of the Linpack benchmark, although they still perform the same task of solving a big linear system, are completely different in their software details.

## 5 The Linpack benchmark in Python

Although the original benchmark program was written in Fortran77, it is not hard to translate it into other languages. It's natural to make some adjustments for the new language, but it's important that in the new language, the numerical computations occur in the same order, to the same quantities. It might be possible to make a better linear equation solver than Linpack, but we would prefer to begin by simply trying to do the exact same thing in our preferred language.

Here is how the main portion of the code looks in Python:

```
  A = 2.0 * np.random.random ( [ n, n ] ) - 1.0
  x_exact = np.ones ( n )
  b = np.matmul ( A, x_exact )
#
#  Factor the matrix.
#
  t = perf_counter ( )
  ALU, ipvt, info = dgefa ( A, n )
  cpu = perf_counter ( ) - t
#
#  Solve the linear system.
#
  t = perf_counter ( )
  x = dgesl ( ALU, n, ipvt, b )
  cpu = cpu + perf_counter () - t
```

As far as possible, the `dgefa()` and `dgesl()` functions repeat what the Fortran77 codes did. However, for convenience and clarity in programming, the input quantities `A` and `b` were not overwritten by results. Instead, separate output values `ALU` and `x` were used.

## 6 Solving the benchmark with scipy()

The `scipy.linalg()` library also has a general solver for linear systems, called `sp.linalg.solve()`. We can set up the Linpack problem, solve it with `scipy()` and compare the timing with our results from `linpack_bench()`. Since the problem, problem size, operating system, programming language, and computer processor will all be the same, we should just be comparing the performance of the software that the two programs are running.

Ignoring all the extra arguments that are available, the basic calling sequence is:

```
  x = solve ( A, b )
```

where `x`, `A` and `b` are all `numpy` arrays.

To do the timings, we can relay again on the `perf_counter()` function from the `time` library.

It is interesting to note that the documentation indicates that when the matrix `A` is in general form, `solve()` calls `dgesv()`, the appropriate linear solver routine from LAPACK. This indicates that here we are really, in part, benchmarking not just a Python code, but an underlying compiled Fortran package.

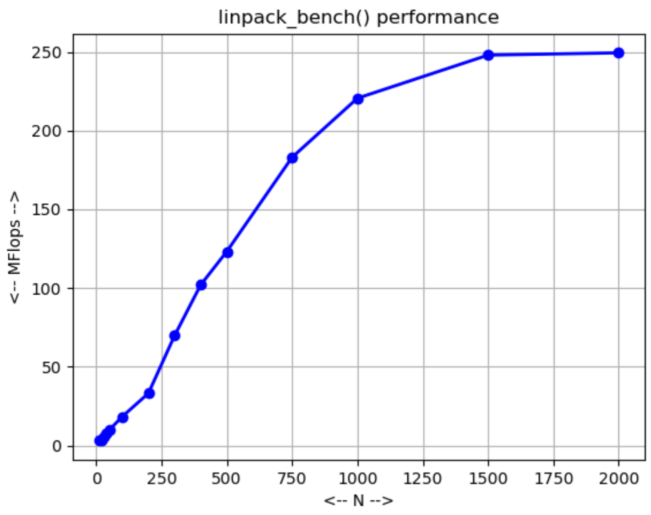
```
def linalg_solve_bench ( n ):

  from scipy.linalg import solve
  from time import perf_counter
  import numpy as np

  A = 2.0 * np.random.random ( [ n, n ] ) - 1.0
  x_exact = np.ones ( n )
  b = np.matmul ( A, x_exact )

  t = perf_counter ( )
  x = solve ( A, b )
  cpu = perf_counter ( ) - t

  r = np.matmul ( A, x ) - b

  a_norm = np.linalg.norm ( A, np.inf )
  r_norm = np.linalg.norm ( r, np.inf )
  x_norm = np.linalg.norm ( x, np.inf )

  ops = ( 2 * n * n * n ) / 3.0 + 2.0 * n * n
  mflops = ops / ( 1000000 * cpu )

  print ( '  Residual norm       = ', r_norm )
  print ( '  Time in seconds     = ', cpu )
  print ( '  MegaFLOPS           = ', mflops )

  return
```

## 7 Behavior over a range of matrix sizes

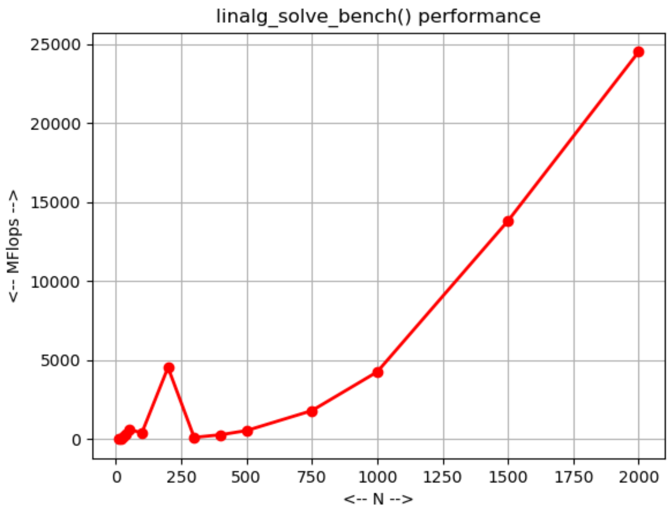
When we run the benchmark programs, we have to pick a value for $n$, the problem size. It is reasonable to choose a value $n$ that seems typical for scientific work; you might think of $n = 100$ or $n = 1000$. We should expect, however, that the FLOPS rating will depend on this value somewhat, especially when we go to very low or high values. In the low case, the work we do on solving the system is so small that other low-level activities will make the performance rating inaccurate. At high values of $n$, we may run into issues that arise when trying to access a huge array, since computer memory can only keep a small set of values nearby, and has to work harder to retrieve data.

Here are some results for a variety of values of $n$:

| n | Linpack MFLOPS | Scipy MFLOPS |
|---:|---:|---:|
| 10 | 3.0 | 3.7 |
| 20 | 3.6 | 62.1 |
| 30 | 5.6 | 180.0 |
| 40 | 7.6 | 332.0 |
| 50 | 9.7 | 635.8 |
| 100 | 18.3 | 405.4 |
| 200 | 33.1 | 4,546.2 |
| 300 | 69.7 | 120.9 |
| 400 | 101.9 | 284.4 |
| 500 | 123.2 | 550.0 |
| 750 | 183.0 | 1,811.0 |
| 1000 | 220.5 | 4,263.9 |
| 1500 | 247.9 | 13,810.0 |
| 2000 | 249.3 | 24,509.2 |

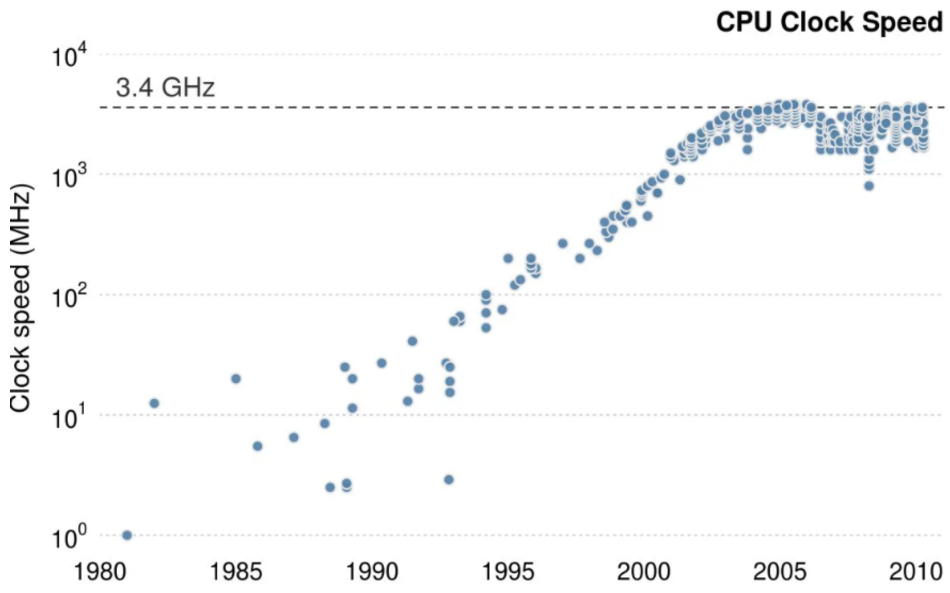


The results for the Linpack version of the benchmark look reasonable: as the value of $n$ increases, the performance rate is improving, although it seems to be leveling off at about 250 MegaFLOPS.



There is something strange going on in the Scipy results: performance improves until we reach $n = 100$, then it drops down, only to soar back up. Suppose we believe these results are meaningful, and even at $n = 2000$, the performance rate is still rising. It could suggest that inside of the Scipy function there is a switch which uses one algorithm up to around $n = 100$, then another afterwards. We might assume the second algorithm is only efficient for large problems, and so that's why we see the "stutter" in the performance rate. The mystery is how the algorithm has reached 24 GigaFLOPS, and is still rising. We will see that in some sense, if we keep our simple model of computer execution, this result is impossible.

## 8 Can you beat the clock?



Around 2010, computer chip clock speeds hit an upper limit that cannot be exceeded.

One way that computers were speeded up was simply to upgrade the internal hardware so that it could run at a faster clockspeed. The exact same calculations would occur, perhaps one per clock tick, but the clock tick faster and the hardware could keep up. Around the year 2010, this kind of improvement stopped entirely. For basic physical laws, it was not possible to make the clock tick any faster.

The upper limit of clock speed, is about 4 GigaHertz. We have so far assumed that a computer could produce (at most!) one floating point result per clock tick. And that means that the best performance possible for any such computer was 4 GigaFlops, that is, 4 billion floating point results per second. But we have already seen for our `linalg_solve_bench()` results what seems to be a performance of 24 GigaFLOPS...and rising. How is that possible?

Developers had already begun alternative ways to speed up calculations. In earlier models, some operations, such as division by a floating point number, actually took several clock ticks (including essentially a quick call to Newton's method). By anticipating that a division might be necessary, new hardware could anticipate some of the calculation, so that the result was ready in one tick.

The CPU was divided up into smaller calculating sections which could independently request memory values, add, multiply, invert, and do other chores, to keep things moving.

Because many calculations involved the same operation repeated on many items of data, a kind of vectorization was developed which allowed for extra fast movement of the memory into pipelines that churned out results one per tick.

Perhaps the most powerful approach involved parallelism, that is, the rearrangement of a calculation into parallel strands which could be independently computed on separate processors, and brought together late in the procedure. This could happen inside a single chip, across several cooperating chips in one machine, or across many servers in a single massive cluster.

If you go into a scientific computing center these days, you will see what looks like stacks and stacks of laptops, with wires and fans everywhere. At any time, some or all of these laptops are talking to each other, working together on a single problem. These new computing monsters are the devices by which the computing performance ratings are still rising into what is known as the “exascale”, such as the Frontier computer at Oak Ridge.

The Frontier computer at Oak Ridge National Laboratory.