Languages, compilers, options and astronomical data

Keith Shortridge, September 2019.

Contents

Summary

Twelve different languages have been tested on one simple problem intended to see how efficiently they handle a common operation in astronomical data processing - accessing individual elements in a 2D floating point array. In some languages, the best way to code the required program was obvious; in others, various approaches were tried. Where compilers or interpreters provided control over optimisation, a range of optimisations were tested. The most striking result is the sheer range of speeds measured; the fastest code ran over a million times faster than the slowest. That slowest result was an outlier, but even without it, the range of speeds covered many orders of magnitudes, and in some cases just enabling compiler optimisation could improve speeds by over two orders of magnitude. The results are presented in detail here, together with discussions of the code used for each language.

Introduction

Astronomers want the data processing programs they run on their data to be as efficient as possible. Processing large amounts of data takes a long time, and the amount of data produced by modern telescopes is growing all the time. Time spent waiting for programs to run can be spent productively, perhaps by making yet another cup of coffee, but ultimately astronomers want programs to run as fast as possible. (All other things, such as producing the right answer, being equal.)

How do we make programs run as fast as possible? Being clever helps - some algorithms are much faster than others, for example. But given an algorithm, how much difference does the way it's coded make? And what about the way that code is turned into executable instructions for the computer by compilers? Are some compilers better than others? Do the various optimisations applied by the compilers make a lot of difference? Are some programming languages going to produce faster programs than others? Interpreted languages, such as Python are becoming hugely popular - how much speed, if any, is being traded off for the flexibility they give you?

One might think the language doesn't really matter - a good compiler for one language should produce final code that runs as fast as that produced by an equally good compiler for a different language. But how are different languages with multi-dimensional arrays? Most images in astronomy are 2D rectangular arrays, and most image processing involves manipulating the values in those arrays. These values are usually floating point numbers. More extensive astronomical data can have additional dimensions – three-dimensional data with two spatial dimensions and one wavelength dimension, for example. But usually these arrays are multi-dimensional matrices; 2D arrays are 'rectangular' - each row is the same length; 3D arrays are cuboids, where each plane is the same size. They're all very regular shapes.

Interestingly, not all languages have the concept of a matrix. Most languages have the concept of an array, at least as a series of so-many items, say floating point numbers, of the same type. But that's only a one-dimensional array. Some of the old programming languages designed, like Fortran, for numerical processing, had built-in the concept of a multi-dimensional array, or matrix. Fortran supported arrays with up to seven dimensions. Other languages, however, only support multi-dimensional arrays as 'arrays of arrays'. So, for example, ten separate 1D arrays can form the rows of a 2D array, and then the 2D array is just a 1D array each of whose elements is one of those 1D row arrays. The rows can even be of different length, and allocated separately. This is grand if you want to do this sort of thing, but it's drifting away from the concept of a simple 2D matrix.

Does this matter? Well, it's always dangerous to draw analogies from how humans work, but generally speaking, someone who has a good understanding of a problem will do a better job of solving it efficiently than someone who doesn't. Can a compiler for a language that doesn't have the concept of a rectangular array do as good a job generating code as a compiler for a language designed specifically to handle such things? If you think of a program as a way of 'explaining' to a compiler just what the problem is, you might think you'd get better results if the language let you describe the problem clearly.

What's described here are the results of a series of tests that came from wondering about things like this. They all involve timing the execution time of a subroutine that performs a fairly trivial operation on a 2D array. A selection of twelve different languages was used. In some cases, there was one obvious way to code the problem, and others there were different approaches that were used. Some languages, particularly the compiled ones, provide a number of possible options for optimising the code, and a number of these were tried in each case. In all, 24 different ways of tackling the problem were coded in the 12 different languages, and nearly 90 separate tests were run, trying each of the 24 different codes with a variety of optimising options. They were all run on the one machine (my laptop), in as close to identical conditions as possible, and were repeated.

How interesting you find the results will depend on just how fascinating you find computer languages and code generation. Or (more likely) on how interested you are in seeing your code run faster. Probably the headline result is simply the sheer range of execution times. The fastest test ran over one and a half million times faster than the slowest. Admittedly, that slowest time was something of an extreme outlier. but even without it, the range of times covered several orders of magnitude. In one case, simply recompiling with optimisation deliberately maximised produced a program that ran nearly 500 times faster than the same program compiled with default settings. If you want to know which these were, read on!

Caveats

Benchmarking is a black art, and the results shown here will not apply to your code. They apply to one set of test programs, run on one machine with a specific set of compilers and interpreters. The test itself was far too simple to be representative of all the ways data can be processed. Successive runs of the test produced slightly different results, and even different overall 'winners'. All the code was written by one person, whose experience with the different languages ranges from pretty proficient to complete beginner. Not all this code will represent the best each language can achieve, and the very nature of the test is unfair to some languages, particularly the interpreted ones.

Once upon a time, someone with a table of instruction times and a listing of the machine instructions generated by a compiler could calculate how quickly the program would run. On a modern machine, it is essentially impossible to know how fast a given instruction will execute; it depends on the state of the cache - both for the main memory and for the instructions themselves - and on the clock speed - which can be boosted or reduced as the machine heats up - and goodness knows what else.

The test

The test used was trivially simple. Pass a 2D array to a subroutine and have that subroutine add the sum of its two index values to each individual element. One example of C++ code for the body of the subroutine is shown below. The idea was to force the compilers/interpreters to access each element individually, rather than making it easy for them to use built-in vector operations to speed things up. However, some compilers turned out to be really quite ingenious in how they handled this.

```
for (int Iy = 0; Iy < Ny; Iy++) {
   for (int Ix = 0; Ix < Nx; Ix++) {
      Out[Iy][Ix] = In[Iy][Ix] + Ix + Iy;
   }
}
```

I've used C++ here because it's fairly clear what's going on here, and I expect most people can understand this. However, C/C++ programmers who take a moment to think may be asking themselves: "if this is in a subroutine, how are Out and In defined? How does the compiler know the dimensions of the arrays?" For all its popularity, C/C++ is not the easiest language to code this problem in, and that's discussed later when I describe the various implementations.

I didn't want to just, say, copy one array into another, for example:

```
for (int Iy = 0; Iy < Ny; Iy++) {
   for (int Ix = 0; Ix < Nx; Ix++) {
      Out[Iy][Ix] = In[Iy][Ix];
   }
}
```

because any C++ compiler worth its salt would realise this is just a big memory copy and would generate a call to an optimised copying routine like memcpy(). I wanted to force it to do something that was different for each element of the array, because I was interested in how efficiently the program could access individual array elements. In the end, I realised I should have made it even harder, because some compilers are remarkably clever nowadays. You'll find out how…

I wanted to put this in a subroutine for a number of reasons. One was that that's how most code is going to be written: you write a general-purpose routine that does a certain operation, and then call it with different data. Another was that I knew that some languages, particularly C/C++ have subtle issues when passing multi-dimensional arrays to subroutines, and I wanted to see how well that was handled. And also, to time such a routine you have to call it many times in a loop from the test program, and I found that if some compilers saw this being called in a such a loop, they realised it just did the same thing each time and only called it once. In fact, for these compilers, it was necessary to compile the subroutine separately, so they didn't realise just what it was doing and so couldn't optimise it away. Such is the lot of the writer of benchmarks.

The languages

The languages were chosen somewhat arbitrarily. In some cases, these were languages I knew very well or at least passably well (C/C++, Fortran, Python, Java, Tcl). In others, they were languages I had dabbled in, or was interested in for various reasons, or had never tried and thought might be interesting (R, Rust, Swift, Julia, Javascript, Perl - yes, I'd never written in Perl).

A practical requirement was that I should be able to get a free compiler for my OS X laptop. One I wanted to try was the Intel Fortran compiler, but although a free short-term demo version was available, I felt that was rather too short-term. Other languages I knew have faded to the point they were probably not relevant (PL/1, Pascal, Forth). In any case, each language added would require a significant amount of time and effort.

And I added hand-coded X86-64 assembler. I had tried this same test a couple of years before, without a great deal of rigour and with just a few languages, and had learned enough X86 assembler to be able to follow the code some of the compilers generated and to try to do better. Back then, I had found I could beat the best C compiler by a factor two by coding in assembler - admittedly, basing my code on what the compiler produced, but removing a number of inefficiencies - and I wanted to see if this was still the case. (I had coded in assembler for various machines over the years, but had never cared for the 32-bit Intel X86 instruction set. The 64-bit X96-64 instruction set is rather more interesting.)

So the full list of languages was:

C/C++,Fortran,Java,Javascript,Julia,Perl/PDL,Python,R,Rust,Swift,Tcl and Assembler.

And the winners are

I'm reluctant to sum up such a complex set of results into a simple list like this, as if this was a horse race, but if I take the best result in terms of speed for each language, I get this list, fastest first, languages on the same line effectively tied:

Assembler, C/C++. Fortran
C++/Boost, Java, Swift
Julia
Rust
Javascript
Python/numpy
Perl/PDL
R
Tcl

But there's a lot more to it than that.

To put the speeds in context, if this had been a horse race, with the fastest horse completing the course in 5 minutes, the tail-ender would have romped in something like two months later. (And that's not even allowing for some of the slower results I got from trying out some of the languages in ways that turned out not to show them at their best.)

The machine and compilers / interpreters

All these tests were run on an OS X laptop: a late-2016 13" MacBook Pro running High Sierra (OS X 10.13.6) with a 2.9 GHz Intel Core i5 processor with 16 GB 2133 MHz LPDDR3 memory.

The compilers and interpreters were:

```
Clang     : Apple LLVM version 10.0.0 (clang-1000.10.44.4)
Gcc       : g++ (GCC) 9.1.0
Gfortran  : GNU Fortran (GCC) 9.1.0
R         : R scripting front-end version 3.6.0 (2019-04-26)
Python2   : Python 2.7.10
Python3   : Python 3.6.2
Java      : java version "1.8.0_121"
Node.js   : v10.13.0
Tcl       : 8.5
```

```
Swift     : Apple Swift version 4.2.1 (swiftlang-1000.11.42 clang-1000.11.45.1)
Julia     : julia version 1.1.1
Rust      : rustc 1.37.0 (eae3437df 2019-08-13)
Perl/PDL  : perlDL shell v1.357
```

The assembler code was processed using Clang, but gcc would have handled it identically - the code used is nothing to do with the compiler.

Results

A Python script (Run.py) was written to build and run each of the 80-odd tests in turn, timing the execution time for a predetermined number of iterations (calls to the subroutine that did the real work of the test). For each test, the number of iterations was chosen experimentally to produce an execution time of somewhere roughly between one and ten seconds. (The range of execution times can be gauged from the fact that the number of iterations used varied between 3 and 5 million - that's not 3 million to 5 million, that's actually three to 5 million!). The time for just one iteration was also measured to allow for the startup times for the test and for the checking code at the end that made sure the test had worked properly. From these two timings, the time that would have been required for 1000 iterations was calculated and recorded. The whole set of tests was run three times, and the fastest time for each test was used for the final rankings.

For each test, the Python script then output that minimum time to complete 1000 iterations, and that time scaled relative to the fastest test - ie how much slower the combination used for that test was compared the fastest test. Here is that output from one run of the program:

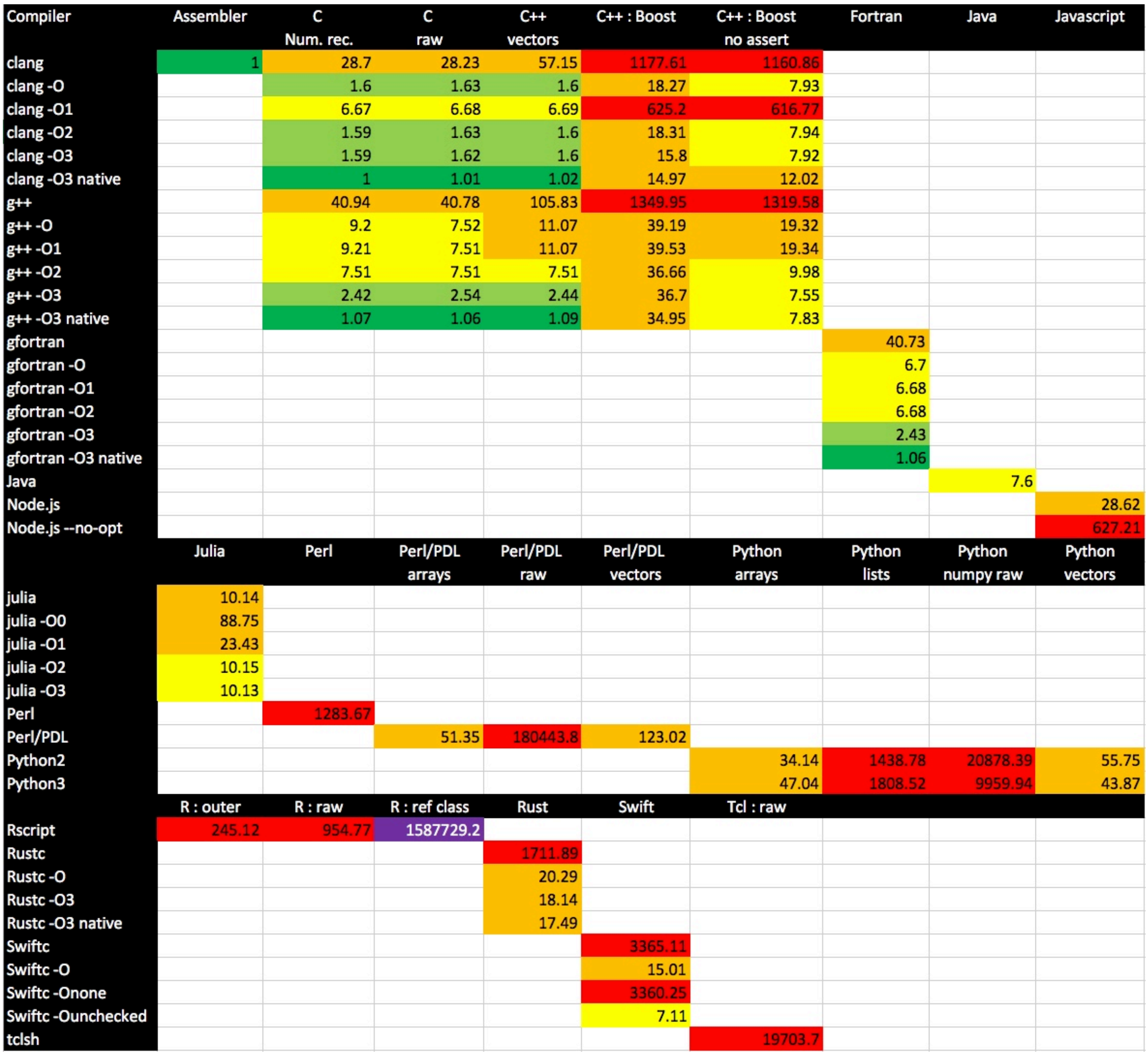
```
Summary of relative speeds:

                   Swift               Swiftc 1K Iter:       6.2, Relative time      3365.11
                   Swift        Swiftc -Onone 1K Iter:       6.2, Relative time      3360.25
                   Swift            Swiftc -O 1K Iter:     0.028, Relative time        15.01
                   Swift   Swiftc -Ounchecked 1K Iter:     0.013, Relative time         7.11
               Assembler                clang 1K Iter:    0.0018, Relative time         1.00
                 Fortran             gfortran 1K Iter:     0.075, Relative time        40.73
                 Fortran          gfortran -O 1K Iter:     0.012, Relative time         6.70
                 Fortran         gfortran -O1 1K Iter:     0.012, Relative time         6.68
                 Fortran         gfortran -O2 1K Iter:     0.012, Relative time         6.68
                 Fortran         gfortran -O3 1K Iter:    0.0045, Relative time         2.43
                 Fortran  gfortran -O3 native 1K Iter:     0.002, Relative time         1.06
                 R : raw              Rscript 1K Iter:       1.8, Relative time       954.77
           R : ref class              Rscript 1K Iter:   2.9e+03, Relative time   1587729.20
               R : outer              Rscript 1K Iter:      0.45, Relative time       245.12
                    Perl                 Perl 1K Iter:       2.4, Relative time      1283.67
          Perl/PDL : raw             Perl/PDL 1K Iter:   3.3e+02, Relative time    180443.80
      Perl/PDL : vectors             Perl/PDL 1K Iter:      0.23, Relative time       123.02
       Perl/PDL : arrays             Perl/PDL 1K Iter:     0.095, Relative time        51.35
          Python : lists              Python2 1K Iter:       2.7, Relative time      1438.78
      Python : numpy raw              Python2 1K Iter:        39, Relative time     20878.39
        Python : vectors              Python2 1K Iter:       0.1, Relative time        55.75
         Python : arrays              Python2 1K Iter:     0.063, Relative time        34.14
          Python : lists              Python3 1K Iter:       3.3, Relative time      1808.52
      Python : numpy raw              Python3 1K Iter:        18, Relative time      9959.94
        Python : vectors              Python3 1K Iter:     0.081, Relative time        43.87
         Python : arrays              Python3 1K Iter:     0.087, Relative time        47.04
                    Rust                Rustc 1K Iter:       3.2, Relative time      1711.89
                    Rust             Rustc -O 1K Iter:     0.037, Relative time        20.29
                    Rust            Rustc -O3 1K Iter:     0.033, Relative time        18.14
                    Rust     Rustc -O3 native 1K Iter:     0.032, Relative time        17.49
                    Java                 Java 1K Iter:     0.014, Relative time         7.60
               Tcl : raw                tclsh 1K Iter:        36, Relative time     19703.70
```

```
                Julia                julia 1K Iter:      0.019, Relative time        10.14
                Julia            julia -O0 1K Iter:       0.16, Relative time        88.75
                Julia            julia -O1 1K Iter:      0.043, Relative time        23.43
                Julia            julia -O2 1K Iter:      0.019, Relative time        10.15
                Julia            julia -O3 1K Iter:      0.019, Relative time        10.13
           Javascript              Node.js 1K Iter:      0.053, Relative time        28.62
           Javascript     Node.js --no-opt 1K Iter:        1.2, Relative time       627.21
              C : raw                clang 1K Iter:      0.052, Relative time        28.23
              C : raw             clang -O 1K Iter:      0.003, Relative time         1.63
              C : raw            clang -O1 1K Iter:      0.012, Relative time         6.68
              C : raw            clang -O2 1K Iter:      0.003, Relative time         1.63
              C : raw            clang -O3 1K Iter:      0.003, Relative time         1.62
              C : raw     clang -O3 native 1K Iter:     0.0019, Relative time         1.01
              C : raw                  g++ 1K Iter:      0.075, Relative time        40.78
              C : raw               g++ -O 1K Iter:      0.014, Relative time         7.52
              C : raw              g++ -O1 1K Iter:      0.014, Relative time         7.51
              C : raw              g++ -O2 1K Iter:      0.014, Relative time         7.51
              C : raw              g++ -O3 1K Iter:     0.0047, Relative time         2.54
              C : raw       g++ -O3 native 1K Iter:      0.002, Relative time         1.06
          C : vectors                clang 1K Iter:       0.11, Relative time        57.15
          C : vectors             clang -O 1K Iter:      0.003, Relative time         1.60
          C : vectors            clang -O1 1K Iter:      0.012, Relative time         6.69
          C : vectors            clang -O2 1K Iter:      0.003, Relative time         1.60
          C : vectors            clang -O3 1K Iter:      0.003, Relative time         1.60
          C : vectors     clang -O3 native 1K Iter:     0.0019, Relative time         1.02
          C : vectors                  g++ 1K Iter:        0.2, Relative time       105.83
          C : vectors               g++ -O 1K Iter:       0.02, Relative time        11.07
          C : vectors              g++ -O1 1K Iter:       0.02, Relative time        11.07
          C : vectors              g++ -O2 1K Iter:      0.014, Relative time         7.51
          C : vectors              g++ -O3 1K Iter:     0.0045, Relative time         2.44
          C : vectors       g++ -O3 native 1K Iter:      0.002, Relative time         1.09
          C++ : Boost                clang 1K Iter:        2.2, Relative time      1177.61
          C++ : Boost             clang -O 1K Iter:      0.034, Relative time        18.27
          C++ : Boost            clang -O1 1K Iter:        1.2, Relative time       625.20
          C++ : Boost            clang -O2 1K Iter:      0.034, Relative time        18.31
          C++ : Boost            clang -O3 1K Iter:      0.029, Relative time        15.80
          C++ : Boost     clang -O3 native 1K Iter:      0.028, Relative time        14.97
          C++ : Boost                  g++ 1K Iter:        2.5, Relative time      1349.95
          C++ : Boost               g++ -O 1K Iter:      0.072, Relative time        39.19
          C++ : Boost              g++ -O1 1K Iter:      0.073, Relative time        39.53
          C++ : Boost              g++ -O2 1K Iter:      0.068, Relative time        36.66
          C++ : Boost              g++ -O3 1K Iter:      0.068, Relative time        36.70
          C++ : Boost       g++ -O3 native 1K Iter:      0.065, Relative time        34.95
C++ : Boost no assert                clang 1K Iter:        2.1, Relative time      1160.86
C++ : Boost no assert             clang -O 1K Iter:      0.015, Relative time         7.93
C++ : Boost no assert            clang -O1 1K Iter:        1.1, Relative time       616.77
C++ : Boost no assert            clang -O2 1K Iter:      0.015, Relative time         7.94
C++ : Boost no assert            clang -O3 1K Iter:      0.015, Relative time         7.92
C++ : Boost no assert     clang -O3 native 1K Iter:      0.022, Relative time        12.02
C++ : Boost no assert                  g++ 1K Iter:        2.4, Relative time      1319.58
C++ : Boost no assert               g++ -O 1K Iter:      0.036, Relative time        19.32
C++ : Boost no assert              g++ -O1 1K Iter:      0.036, Relative time        19.34
C++ : Boost no assert              g++ -O2 1K Iter:      0.018, Relative time         9.98
C++ : Boost no assert              g++ -O3 1K Iter:      0.014, Relative time         7.55
C++ : Boost no assert       g++ -O3 native 1K Iter:      0.014, Relative time         7.83
        C : Num. rec.                clang 1K Iter:      0.053, Relative time        28.70
        C : Num. rec.             clang -O 1K Iter:      0.003, Relative time         1.60
        C : Num. rec.            clang -O1 1K Iter:      0.012, Relative time         6.67
        C : Num. rec.            clang -O2 1K Iter:     0.0029, Relative time         1.59
        C : Num. rec.            clang -O3 1K Iter:     0.0029, Relative time         1.59
        C : Num. rec.     clang -O3 native 1K Iter:     0.0018, Relative time         1.00
        C : Num. rec.                  g++ 1K Iter:      0.076, Relative time        40.94
        C : Num. rec.               g++ -O 1K Iter:      0.017, Relative time         9.20
        C : Num. rec.              g++ -O1 1K Iter:      0.017, Relative time         9.21
        C : Num. rec.              g++ -O2 1K Iter:      0.014, Relative time         7.51
```

```
C : Num. rec.            g++ -O3 1K Iter:     0.0045, Relative time         2.42
C : Num. rec.     g++ -O3 native 1K Iter:      0.002, Relative time         1.07
```

The test weren't run in any particular order, so this isn't an easy table to take in. The Python script also put this out in a .csv file that could be fed into Excel, and with a bit of juggling, that produced this:



Compiler	Assembler	C	C	C++	C++ : Boost	C++ : Boost	Fortran	Java	Javascript
		Num. rec.	raw	vectors		no assert			
clang	1	28.7	28.23	57.15	1177.61	1160.86			
clang -O		1.6	1.63	1.6	18.27	7.93			
clang -O1		6.67	6.68	6.69	625.2	616.77			
clang -O2		1.59	1.63	1.6	18.31	7.94			
clang -O3		1.59	1.62	1.6	15.8	7.92			
clang -O3 native		1	1.01	1.02	14.97	12.02			
g++		40.94	40.78	105.83	1349.95	1319.58			
g++ -O		9.2	7.52	11.07	39.19	19.32			
g++ -O1		9.21	7.51	11.07	39.53	19.34			
g++ -O2		7.51	7.51	7.51	36.66	9.98			
g++ -O3		2.42	2.54	2.44	36.7	7.55			
g++ -O3 native		1.07	1.06	1.09	34.95	7.83			
gfortran							40.73		
gfortran -O							6.7		
gfortran -O1							6.68		
gfortran -O2							6.68		
gfortran -O3							2.43		
gfortran -O3 native							1.06		
Java								7.6	
Node.js									28.62
Node.js --no-opt									627.21
	Julia	Perl	Perl/PDL	Perl/PDL	Perl/PDL	Python	Python	Python	Python
			arrays	raw	vectors	arrays	lists	numpy raw	vectors
julia	10.14								
julia -O0	88.75								
julia -O1	23.43								
julia -O2	10.15								
julia -O3	10.13								
Perl		1283.67							
Perl/PDL			51.35	180443.8	123.02				
Python2						34.14	1438.78	20878.39	55.75
Python3						47.04	1808.52	9959.94	43.87
	R : outer	R : raw	R : ref class	Rust	Swift	Tcl : raw			
Rscript	245.12	954.77	1587729.2						
Rustc				1711.89					
Rustc -O				20.29					
Rustc -O3				18.14					
Rustc -O3 native				17.49					
Swiftc					3365.11				
Swiftc -O					15.01				
Swiftc -Onone					3360.25				
Swiftc -Ounchecked					7.11				
tclsh						19703.7			

Discussion

The biggest surprise, for me, was simply the range in speeds that these tests show. It isn't just a factor 10, it really is several orders of magnitude separating the slowest and the fastest. And in some cases, orders of magnitude between the same code when optimised and without optimisation. It's worth knowing that this can be the case.

Is this a fair and representative test? Absolutely not. The test code used here deliberately tried to test how well a system handled access to individual elements of an array, in a way that would be difficult for a compiler to optimise. For example, nested loops that simply copied each element of a 2D array into another array could be turned into a single call to an optimised routine like C's memcpy() that just copies data regardless of its structure, and that wouldn't have tested access to individual array elements. However, it means that the test is probably unrepresentative of many real-world problems, and it certainly allows some languages to shine more than others. In particular, it really penalises interpreted languages like Python, which generally get their speed from packages like numpy, which can work on large blocks of data as quickly as on they do on one element. If your problem is a good match to the bulk processing operations of libraries like numpy, you will find these much more efficient than they may appear here. It's also unfair to new languages – compilers for C/C++ and Fortran have years of development behind them. The evolution of the Swift compiler shows the gains that can come with just a few years of work.

But, all caveats aside, in this particular test there were a clear set of combinations that produced very impressive speeds. These were C/C++ and Fortran with the most aggressive optimisation options enabled. The assembler code is there too, but this was based mainly on the code generated by the C++ compilers, and represented a lot of effort for no tangible gain in performance. In the summary diagram, these are coloured green.

These cases all made use of the vector instructions supported by the X86-64 instruction set, and that may make them unrepresentative. The compilers - to my surprise - were able to work out ways to use these vector instructions to solve what had been intended as a test of access to individual array elements. Whether or not they will be able to do this for more realistic, real-world problems, is unclear, so it may be that these particularly good results are, in their own way, unrealistic outliers.

I tested a number of different ways of setting up 2D arrays in C/C++ (emulation using 1D arrays, the Numerical Recipes array of arrays scheme, STL vectors of vectors, and Boost multi-dimensional arrays). Both the Clang and g++ compilers were able to return almost identical top notch results, using vector instructions, for all of these approaches with the exception of the Boost arrays, where it may be that the template code obscures just a little exactly what the program is trying to do.

Then there were a group of results (yellow in the summary table) which clearly represent very efficient access to single array elements, and these may be representative of what one can expect from very efficient real-world code. These included C/C++ and Fortran at slightly lower optimisation levels, C++ using Boost (with assertions off), Java, Julia, and Swift. (I set the cut-off here at times up to 10x the very fastest test results, rounded - so Julia at 10.13 just got in. Rust didn't, but was close, at ~17.)

Then there was a very wide band (orange in the summary table) ranging from 17 times the fastest code (Rust) to 123 times (Perl/PDL vectors). This included the faster of the interpreted languages, some with help from array-handling packages, which isn't really answering the question about access to single array element access, but which may give realistic estimates of what these languages can do for real problems. In decreasing order of speed, these were Rust, Javascript, Python (with numpy vectors or arrays), Perl (with PDL vectors or arrays). Javascript was faster than I'd expected, an interpreted language coming close to what a compiled language can do (just-in-time compilation blurs the line between what's compiled and what isn't, of course).

Then there were the rest, a set of languages - or ways of using those languages - that aren't really at their best used for this sort of problem: Python and Perl using their built-in array types, R, and Tcl, and a lot of cases where code that could run much faster was run without optimisation (notably Swift and C++ with Boost arrays).

And the one outlier, R with reference class objects, which I really should have simply forgotten about, except that it gave the attention-grabbing headline about a more than 1 million to one speed range from the fastest to the slowest code tested. I apologise to R, which really was not meant to be used that way, and, used properly, runs far, far, faster.

Algorithms matter. Other than turning on optimisation, you're more likely to improve the speed of your code by changing the algorithm you use than by changing any of the things described here.

But, ultimately, how important is speed? If you don't feel constrained by the speed of any given program, be happy! Leave it alone, so long as it gets the right results. If you really would like something to run faster, first see if you're using the right optimisation flags - switching on optimisation is almost a free win. (And, personally, it's been a long time since I saw a correct program fall prey to an actual optimisation bug - although optimisation can make a miscoded program run differently, and may bring out problems hidden until now.) Most compiled languages, with optimisation enabled, are pretty close to one another. I feel there's little in these results to justify changing from one compiled language to another just to get speed. Most compiled languages are able to call individual routines in C, so you might justify recoding a particularly slow section in C and calling it from your language of choice. This is more relevant to interpreters; you don't have to give up the convenience of an interpreter to get the speed of C, in most cases, since you can link in C modules and run them from your interpreted code. (Most interpreters are written in C anyway.)

Finally, a list of one-line takeaways from all this:

The range of speeds for programs doing the same thing can be huge.
Coding in assembler really isn't worth it any more.
Having the right optimisation flags makes a big difference to compiled languages.
For the C++ Boost libraries, optimisation make an enormous difference.
Most compilers don't optimise by default.
Some obscure optimisations (-march=native, for example) can help.
Modern mature compilers are really good.
Being able to understand assembler - and your hardware - may be useful.
Compiled code beats interpreted code. Duh.
Libraries like numpy speed things up a lot.
Memory access can be a bottleneck.
At maximum optimisation C/C++ and Fortran were the fastest languages.
Your program will have different trade-offs.

Details

The following sections get into rather more technical detail about the handling of 2D arrays in general, and the code used for the individual tests in the various languages.

Array layout in memory

We imagine that a 2D array will have all its data allocated contiguously in memory. This is the case when a Fortran compiler creates an array using something like:

```
  real a(rows,columns)
```

It's really only a convention to say that the first dimension specified is the number of rows in the matrix and the second is the number of columns. The data isn't actually laid out in memory in two dimensions, because computer memory is one-dimensional, just one memory byte after another. How does the compiler map the rows and columns onto the linear memory in practice?

A Fortran compiler will allocate enough memory to hold the array - if each element of a 'real' array (single precision floating point) takes 4 bytes, then it will allocate (rows * columns * 4) bytes for the array. (I'm using Fortran because things are marginally more complicated with C. Bear with me.) But will it put the data for the array so with all the data for the first row, then all the data for the second row etc, or will it put all the data for the first column, then all the data for the second column, and so on?

The first element in memory will presumably be a(1,1). (Fortran numbers its array elements starting at 1.) Will the next elements be a(1,1), a(2,1), a(3,1) etc, (that would be all the data for column 1 first), followed by a(1,2), a(2,2), a(3,2) etc (all the data for column 2).
Or will it be a(1,1), a(1,2), a(1,3) etc then a(2,1), a(2,2), a(2,3) and so on?

Does it matter?

At first glance, no it doesn't. As long as the compiler knows where the data is, why should we care?

But it does matter. Mostly, it matters when you write nested loops that work through all the elements of the data. Something like (sorry about the Fortran, but I'm sure it's obvious what's happening here):

```
do Iy = 1,Ny
   do Ix = 1,Nx
      Out(Ix,Iy) = In(Ix,Iy) + Ix + Iy
   end do
end do
```

Actually, Fortran puts the data in memory with the data for the first column, then the second, and so on: a(1,1), a(2,1), a(3,1) etc. So, the code here is working along the data column by column, working through the first column - the elements with iy = 1, and then moving on the next column - the elements with iy = 2, and so on. The way this is usually put is:

"In Fortran, the first index varies fastest" as you work along the memory from the first element to the next.

So, this code is working its way through the arrays in the order they're laid out in memory, varying the first index fastest for each array. Instinctively, you'd assume that this is the most efficient way to do things, and you'd be right. The memory sub-system will normally pull in a whole block of memory into cache at a time, assuming that, if you wanted the contents of one location, there's a good chance you'll want the contents of the next pretty soon as well. So, memory access speeds up if you go through the array the way it's arranged in memory. (This is even more important if you have arrays so large they won't fit in physical memory and have to be stored on disk as virtual memory - the virtual memory sub-system will make the same assumptions, and will read blocks of memory from disk at a time, assuming you'll work through it in that order. If you get the loops in the wrong order on a virtual memory system - if you 'go against the grain' of the virtual memory - it starts to run very inefficiently and slowly, thrashing as you keep demanding array elements from widely separated places on the disk.)

Also, if you work through the array as it's stored in memory, the give the compiler more opportunity to optimise the code. The vector optimisations used by the fastest compilers in these tests only work if the code works along the arrays in the order they are in memory. There's no point in filling a vector register from memory if you only end up processing the first element in it and then move on to some other array element that wasn't loaded into the vector.

So, at some level of worrying about performance, you need to know if your language is 'column-major' - stores the arrays column by column, like Fortran (and Julia, and R, incidentally) - or 'row-major' - stores the arrays row by row.

C is a row-major system.

If you declare a C array as

```
   float a[rows][columns];
```

then C stores this in memory as a[0][0], a[0][1], a[0,2] etc, followed by a[1][0], a[1][2], a[1][3] etc.

In C the second index varies fastest as you go through the memory layout. And C starts counting at 0, of course. This is partly because C thinks of a 2D array as an array of 1D arrays. So a[0] is the address of the start of the first row, a[1] is the address of the start of the second row. It pretty much follows automatically that a[0][0] - the first element of the first row - has to be followed immediately by a[0][1], so all the row elements are contiguous in memory.

In general, languages that use an a[0][0] syntax as opposed to an a(0,0) or a[0,0] syntax are thinking in terms of arrays of arrays, and in these languages the second index is always going to vary fastest as you go up through memory.

When you declare an array as shown above, with a specific size, C allocates memory for it as a block, just as Fortran does, except that it lays out the contents of the array row by row - in 'row-major' order, unlike Fortran. Although the a[iy][ix] syntax looks as if a is a 1D array whose elements contain the addresses of the start of each row, that array doesn't actually exist. C doesn't actually create a second array with one element for each row that holds the address of the start of each row. C knows that when it sees a[n] this is the address of the start of the data in the nth row of that 2D block and it can generate code to calculate that because - in the block where a is declared - it knows the length of each row. But when it passes the array a to a subroutine, C has no way of passing that dimension information over to the subroutine. If C did actually create two arrays when it allocates the memory for a[rows][cols], one for the data in the array and one to hold the row start addresses, it could pass that second array to the subroutine and the subroutine would be able to do the index calculations needed to access individual elements. But this is getting ahead of ourselves a bit - I come back to this in the section on C in the part of this document that describes the individual languages in more detail.

Other languages have learnt from this, and have all worked out ways to let subroutines know the dimensions of arrays they are passed. A general rule seems to be that languages that use an array syntax like a[m][n] are treating 2D arrays as rows of vectors, and are using row-major order to store the data. As described above, this follows naturally from the 'arrays of arrays' way of thinking of things. Languages that use a[m,n], or a(m,n) are more likely to use column-major order, although they could quite logically use either order.

The bottom line, after all this, though, is simply that to write efficient code, the order in which your traverse the element of an array needs to match the order in which they are stored in memory. And to do that, you need to know whether your language stores in row-major or column-major order.

And once you know that, remember in your loops:

Row-major: 2nd index varies fastest.
Column-major: 1st index varies fastest.

Individual languages

This section goes into quite a lot of detail about the individual languages, how the code worked, their occasional quirks and the results of the tests. Some are fairly straightforward, usually because there was one obvious way of coding the problem. Some - look at the section on C/C++ - are very long, usually because there wasn't one obvious standout way to tackle the problem in that language. Don't feel obliged to read them all!

Assembler

Once upon a time, if you wanted speed, you had to code in assembler, writing individual lines of code that mapped directly onto the instructions the processor would execute. That was quite a while ago. Once upon an even earlier time, if you wanted to code at all, you had to do it in assembler. But those days are gone, thank goodness.

Optimising compilers have put the assembler writer out of business. A few years ago, I could get a factor two improvement over the C++ compilers by coding this problem in assembler. One of the disappointments of this current set of tests is that this is no longer the case; the compilers have really become very good indeed. (You may not find that disappointing, of course.)

Nevertheless, I think there are still some advantages to knowing enough about the processor you're using to be able to understand assembler. Not to write it, not anymore, but it's often the case that the more you know about the underlying system you have, the more efficiently you can make use of it and the tools that come with it. If you set the -S flag in a C/C++ compilation, the compiler produces a .s file with the assembler code it's generating, and you can sometimes (OK, not very often) learn something useful from that. In this case, I looked at the code Clang generated with the -O3 flag and saw it using the 128-bit vector instructions supported by my core i5 processor. I wondered why it wasn't using the 256-bit instructions the processor supported, did "man c++" and looked carefully at the optimisation flags, and discovered the "-march=native" option. I used that, and the compiler generated 256-bit instructions that more than halved the execution time. Maybe that's very much a one-off, and maybe I should have known about the option in any case, but I found that satisfying.

This is the one language section where I'm not going to list the code for the subroutine that adds the index values to the array elements. It's over 100 lines long, even without comments, and it's impossible to understand without the comments. And it's hard even then. Just as a taste, here's the code for the fast part of the inner loop:

```
IxLoop:
   vcvtdq2ps %ymm2,%ymm4           # Convert 1st set of Ix+Iy values to floats.

   vaddps    %ymm6,%ymm4,%ymm9     # Add 8.0 to each Ix+Iy value. (2nd set of 8)
   vaddps    %ymm6,%ymm9,%ymm10    # Add 8.0 to each Ix+Iy value. (3rd set of 8)
   vaddps    %ymm6,%ymm10,%ymm11   # Add 8.0 to each Ix+Iy value. (4th set of 8)

   vaddps    (%r11),%ymm4,%ymm0    # Add Ix+Iy values to 1st input value set.
   vaddps    32(%r11),%ymm9,%ymm1  # Add Ix+Iy values to 2nd input value set.
   vmovups   %ymm0,(%r15)          # Store the 1st set of results in Out.
   vmovups   %ymm1,32(%r15)        # Store the 2nd set of results in Out.
   vaddps    64(%r11),%ymm10,%ymm12 # Add Ix+Iy values to 3rd input value set.
   vaddps    96(%r11),%ymm11,%ymm13 # Add Ix+Iy values to 4th input value set.
   vmovups   %ymm12,64(%r15)       # Store the 3rd set of results in Out.
   vmovups   %ymm13,96(%r15)       # Store the 4th set of results in Out.

   vpaddd    %ymm8,%ymm2,%ymm2     # Add 32 to each of initial Ix+Iy increments.

   addq      $128,%r11             # Increment pointers to In and Out by 128,
   addq      $128,%r15             # ready for the next 32 elements.
   cmpq      %r8,%r11              # Have we reached the end of this line yet?
   jb        IxLoop                # If not, do another 32 elements.
```

Confession time. I'd not have written that without having first looked at the —S O3 assembler output from the compiler. It was only when I looked at that listing that I realised how clever the compiler was being.

It had worked out that adding the Ix and Iy index values to each element amounted to adding 0,1,2,3,… etc. to the elements of the first line, then 1,2,3,4… etc. to the elements of the second line and so on. And it knew it could handle 4 elements at a time in one of the 128-bit vector registers it was using with -O3. So, it loaded 0,1,2,3 into one register, did a vector add of that to the first 4 element in the input array, stored those into the output array, then added a vector it had pre-loaded with 1,1,1,1 to that first register, then added that to the next four elements of the input array, and so on. Then it reset itself for the next row.

Actually, it didn't quite do that, it set up four sets of registers like this and went through the input data 16 elements at a time, then added 16,16,16,16 to each of the four sets of registers and carried on - it unwound the loop partially to one that ran 16 elements at a time instead of just one.

The code above is close to what the compiler generates with -march=native, using the 256-bit registers to handle data 32 elements at a time. I thought this was quite impressive stuff. A couple of years ago, the compiler was doing something similar, but with a number of surprising inefficiencies on top of that basically clever structure, and by using similar but cleaner code I got my factor 2 improvement. But now, the compiler is doing much better. I don't do things quite the way it does, and I may squeeze a little more out in places (I keep the index values as floats instead of ints, which may help), but not enough to be worth the effort - and anyway, I lose on the roundabouts, as I'll come to.

Logically, the assembler programmer should always be able to at least draw with a compiler; in the worst case, they simply copy what the compiler produces and it runs at exactly the same speed. Maybe they can improve in places, in which case they eke out a win. But that isn't quite the case in practice. The compiler has more patience, and can write code much faster. It can unwind loops that a human isn't going to bother with. In this case, you might be able to do a row at a time very quickly 32 elements at a time, but what if the row length isn't a multiple of 32? Then you have a load of up to 31 values at the end that you have to deal with. My assembler code just falls back on doing them one at a time in a simple end loop. The compiler can see if it can break those up - maybe it's a multiple of 16, in which case it be done four goes of 4 elements at a time with 128-bit registers. Maybe it's a multiple of 4, in which case it can be done just 4 at a time with 128 bit registers. A compiler has the patience to generate the code for these cases, and I don't. So, my simple end loop is less efficient than the code from the very patient compiler.

In the end, the timing differences between my best assembler effort and the best optimised compiler code was in the noise of the test results. In fact, the difference between the best C/C++, Fortran, and Assembler was so small that if you care about it, you're taking all this far too seriously.

I still find it satisfying to be able to code in assembler, but then, I also drive a car with a manual gear change. I'm keeping the car, but I doubt if I'll be writing a lot more assembler. But sometimes, if you can read the code different options or coding techniques generate, you can learn something useful, and it does give you a better feel for the system you're using.

Fortran

Fortran is still heavily used for high performance computing, particularly for large scale numerical simulations. It isn't a coincidence that these are problems that use large multi-dimensional rectangular arrays, something Fortran provides excellent support for, once you've created them. If you pass an array to a Fortran subroutine, you can tell the subroutine the array dimensions and the compiler will handle all the array index calculations as if it's the most natural thing in the world - which I thought it was, until I tried to do the same thing in C.

'Once you've created them' is, or was, an important caveat. Originally (admittedly, back in the '50s), Fortran had no way of dynamically allocating memory for its arrays, which had to be declared with hard-coded sizes. In the days when everyone (almost!) was using DEC VAXes and Fortran 77, people wrote C routines to be called from Fortran to allocate arrays in memory using malloc() and then used non-standard tricks to pass these to subroutines as if they were 'normal' Fortran arrays. It was only with Fortran 90 that Fortran gained modern facilities for dynamic array allocation. And while people were waiting for all the modern facilities Fortran now has, a lot of programmers moved over to C and then to C++.

Even so, the Fortran code for the main subroutine used in this test:

```
subroutine sub (In,Nx,Ny,Out)
implicit none
integer Nx,Ny
real In(Nx,Ny),Out(Nx,Ny)
integer Ix,Iy
do Iy = 1,Ny
   do Ix = 1,Nx
      Out(Ix,Iy) = In(Ix,Iy) + Ix + Iy
   end do
end do
end
```

could have run on most Fortran 77 implementations. It's only the main test routine I use to call this that makes use of the more recent Fortran dynamic array allocation facilities. And I think this is as clear and straightforward an implementation as you'll find in any language. (The 'implicit none' statement may not be obvious and actually isn't needed here - it tells the compiler that all variables have to be explicitly declared, and tends to be included out of habit.)

In particular, note that the arrays are clearly multi-dimensional rectangular arrays, not arrays of arrays. A Fortran 2D array is held in contiguous memory, and to access them all a compiler needs to know is where the array starts in memory and its dimensions. In principle, this ought to make it easy for a Fortran compiler to generate efficient code for accessing array elements.

The arrays are created using:

```
allocate (In(Nx,Ny),Out(Nx,Ny))
```

and the subroutine call is :

```
call sub(In,Nx,Ny,Out)
```

The use of allocate() to create arrays whose dimensions are only known at run time is a Fortran 90 addition, so the main routine for this test requires at least an F90 compiler.

The only Fortran compiler I had available was gfortran. The Intel Fortran compiler claims to be a particularly good optimiser and is available for OS X, but, as already mentioned, is not free. However, gfortran seems to do very well, and is pretty much on a par with g++ for C/C++, returning amongst the fastest speeds tested. By default, gfortran does not optimise, but it supports a number of different optimisation levels, -O1, -O2, -O3 and just -O. Of these, -O, -O1, -O2 all produced roughly identical times in this case, while -O3 introduced the use of the X86-64 128-bit vector instructions for a noticeable speed-up. There are a large number of additional flags that switch various optimisations in and out, but the most significant improvement came with the -march=native flag, which allows the compiler to use any features the current processor might have, which in this case included support for the 256-bit vector instruction set.

The various optimisation flags produced the following times (relative to the fastest combination tested):

No optimisation (the default) took about 40 times the time of the most optimised code.
-O, -O1, -O2 were all about the same for this code, about 6 times the speed of the best.
-O3 tweaked another 2 to 3 times the performance out of the code
-O3 -march=native was only a neglibible fraction worse than the equivalent C++ code.

When I ran some earlier versions of some of these tests a few years ago, the code compiled by gfortran could shave about 25% off that taken by equivalent code compiled using g++. And back then, I could beat gfortran by using assembler, which is no longer the case, so both gfortran and g++ have improved significantly, but it looks as if g++ has improved rather more than gfortran has. I don’t find this surprising, given the relative popularity of the languages.

C/C++

C/C++ is the most complex of the languages to cover here, and this is a very long section, simply because it really isn’t obvious how best to code the problem! C has had a long-standing problem with passing multi-dimensional arrays to subroutines, and C++ merely provided more ways of getting around the problem, rather than actually building a simple solution into the language itself.

This gets complicated. Let’s look at how you might write a subroutine that is passed a 2D integer array and sets each element to the sum of its index values. That’s similar to what the main test described in this document does, but is rather simpler.

C/C++ do support true rectangular multi-dimensional arrays. You can have code that declares one quite trivially:

```
      #define nx 10
      #define ny 20

   int a[ny][nx];
   for (int iy = 0; iy < ny; iy++) {
      for (int ix = 0; ix < nx; ix++) {
         a[iy][ix] = ix + iy;
      }
   }
```

This has been in C right from the start. Note though that the size of the array a - the nx,ny dimensions - have to be constants, set here using #define. That was a limitation in C right up until C99 which introduced variable length arrays, which let you do:

```
   int nx = 5;
   int ny = 4;
   int a[ny][nx];
   for (int iy = 0; iy < ny; iy++) {
      for (int ix = 0; ix < nx; ix++) {
         a[iy][ix] = ix + iy;
      }
   }
```

and this obviously makes things much more flexible. You can get your array size from anywhere (say from the command line arguments) and create an array of that size. I'm just not interested here in techniques that only work for arrays whose size are known at compile time. Astronomical data processing programs have to work with images (or other data) of varying dimensions, and I want something that will work with dynamically allocated arrays.

In both cases, these are genuine rectangular 2D arrays, and a is simply the start address of a block of nx by ny integers. In the block of code that defines that array, the compiler knows how to work out where a[iy][ix] is, because it knows the number of elements in each row. So it can calculate an offset from the start of the array to any given element. It knows a[iy][ix] is offset by (iy * nx + ix) elements from the start of a. Remember that index numbers start at 0. The compiler stores the array in row-major order, row by row, so there are iy rows worth of data, each of nx elements, before the start of the row numbered iy, and then another ix data elements before you come to element numbered ix in that row, giving (iy * nx + ix) elements from the start of the array (at a[0][0]) to element a[iy][ix]. Sorry if i'm spelling out something that's fairly obvious.

What's the problem? The compiler knows how do this "in the block of code that defines that array". And not anywhere else. In particular, not in a subroutine. So, the problem is how you pass this to the subroutine, and what the subroutine has to look like. What you can't do is this:

```
void fillArray (int a[][],int nx, int ny) {
   for (int iy = 0; iy < ny; iy++) {
      for (int ix = 0; ix < nx; ix++) {
         a[iy][ix] = ix + iy;
      }
   }
}

int main() {
   int nx = 5;
   int ny = 4;
   int a[ny][nx];
   fillArray (a,nx,ny);
   return 0;
}
```

You can't pass one of these arrays to a subroutine and have the subroutine work out how to calculate the offset into a of an element like a[iy][ix]. Because the subroutine doesn't know how long each row is. It doesn't know that nx is the length of the rows and ny is the number of rows. This code doesn't compile - the compiler complains about a[] having an 'incomplete element type".

Now, if you know that each row is going to be 5 elements long, you'll find that this will compile:

```
void fillArray (int a[][5],int nx, int ny) {
   for (int iy = 0; iy < ny; iy++) {
      for (int ix = 0; ix < nx; ix++) {
         a[iy][ix] = ix + iy;
      }
   }
}
```

and you can call it like this:

```
#define nx 5
```

```
#define ny 4

int main() {
   int a[ny][nx];
   fillArray (a,nx,ny);
   return 0;
}
```

But, frankly, I don’t think that’s very useful. Not for this sort of problem. What’s more, note that I used an old fixed size array in the main program. It does work with C99 variable length arrays as well, so long as you declare them as int a[ny][5] - you have to use a constant in the declaration for the row size if you want to pass it to a function that declares it this way. Which sort of makes sense.

I don’t know why C/C++ can’t manage to define a syntax that would let you pass the size of an array to a subroutine. Simply supporting something like:

```
void fillArray (int a[int rows][int cols],int nx, int ny) {
```

would solve all this at a stroke.

But, since that hasn’t happened, how do we pass a dynamically-sized 2D array to a subroutine?

There are two fairly straightforward answers, both slightly unpalatable, that work in ordinary C, as opposed to C++. (I’m using C++ syntax here, because little things like being able to specify the type of the loop index in the ‘for’ statement itself are convenient, but the basic ideas work without needing C++ features.) Then there are other approaches that do make use of C++ features.

The ‘C raw’ option

You can treat the array as 1D and do the indexing yourself:

```
void fillArray (int a[],int nx, int ny) {
   for (int iy = 0; iy < ny; iy++) {
      for (int ix = 0; ix < nx; ix++) {
         a[iy * nx + ix] = ix + iy;
      }
   }
}
```

In this case the call has to be:

```
   int ny = 4;
   int nx = 5;
   int a[ny][nx];
   fillArray (a[0],nx,ny);
```

Because a[0] is the address of the start of the first row of the 2D array, which is in fact the start of the nx * ny elements that make up the array data. This works because we know that in fact the compiler has placed all the rows contiguously in memory and we can treat the whole 2D array as a 1D array that starts at that address. (We could also use *a, &(a[0][0]), which all come to the same thing.)

This works. but it does seem irritatingly as if the programmer is doing the hard work for the compiler. But more than that, it's simply going to be error prone. Writing a[iy][ix] seems quite natural, and is easy to read. a[iy * nx + ix] is harder.

So, personally, I don't care for that solution. However, this is the test version I called "C : raw", and the compilers do a very good job of optimising this sort of code - they have been given a good start, after all!

Importantly, and we'll come back to this, that version of fillArray() would also work if we'd simply allocated a large enough block of memory using malloc() or something similar. So I could also do:

```
   int nx = 5;
   int ny = 4;
   int* a = (int*) malloc(nx * ny * sizeof(int));
   fillArray (a,nx,ny);
```

But really all this is doing is saying I can emulate a 2D array using a 1D array of the equivalent size if I do all the hard work myself. I really want to be able to write code that looks as if it's working with a real 2D array.

The "Numerical Recipes" option

An alternative approach is to forget that C/C++ does actually support (sort of) rectangular 2D arrays, and instead to use an 'array of arrays' approach. This is the scheme popularised (or at least, used) in the book 'Numerical Recipes in C". The nice thing about this is that we can write the subroutine in what feels to me at least to be a fairly natural style.

```
void fillArray (int* a[], int nx, int ny)
{
   for (int iy = 0; iy < ny; iy++) {
      for (int ix = 0; ix < nx; ix++) {
         a[iy][ix] = ix + iy;
      }
   }
}
```

This lets the subroutine code refer to the array elements using the convenient a[iy][ix] syntax, and there's no calculating the offsets by hand. That looks more like working with a proper 2D array. But this works because what we're passing as 'a' is no longer the address of the start of a 2D rectangular array. Instead, it's what the argument description says it is - 'a' is now a 1D array of pointers to int, and those pointers to int are the addresses of the start of each row of the array. So 'a' is now a 1D array of pointers to the 1D row arrays. That makes writing the subroutine simple, but how do we set up that 'array or arrays' in the calling routine?

What we need to pass to the subroutine is an array, ny long, that contains the addresses of the start of each row of the actual data. C/C++ doesn't provide an out of the box facility to do that for us, so it means a bit of hard work when we create the original array data. Here would be one way to do it, but it isn't the one I prefer. I'll explain why in a moment.

```
   int nx = 5;
   int ny = 4;
   int* a[ny];
   int adata[ny][nx];
   for (int iy = 0; iy < ny; iy++) {
      a[iy] = &(adata[iy][0]);
   }
   fillArray(a,nx,ny);
```

That's quite straightforward code. We create a 2D rectangular array 'adata', and so that we can pass it to fillArray(), we create a 1D array that we fill with the addresses of the start of each row. And this works. (Note that we could have created a separate 1D row array for each row of data and put their addresses into 'a'. We're not using the fact that the rows happen to be contiguous in memory - and fillArray() isn't allowed to assume they are.)

The reason I don't usually do this is because most compilers will allocate these variable length arrays on the stack rather than in the much larger heap used by routines such as malloc(). This isn't a requirement, but it seems to be the case for both clang and gcc. In this case, this is fine for relatively small arrays, but if you try to create a variable length array large enough for a substantial image, you may have problems. On my laptop, the code here runs fine for nx = 5, ny = 4, but generates a segmentation fault if nx = 5000, ny = 4000.

For this reason, I think this sort of code - trying to work with seriously large 2D arrays - needs to create them with one of the malloc() family of memory allocators. Apart from anything else, you can check the return value from malloc to see if you have a problem, and you can't do that with variable length arrays. So I prefer something like:

```
   int nx = 5;
   int ny = 4;
   int** a = (int**) malloc(ny * sizeof(int*));
   int* adata = (int*) malloc(nx * ny * sizeof(int));
   int* aptr = adata;
   for (int iy = 0; iy < ny; iy++) {
      a[iy] = aptr;
      aptr += nx;
   }
   fillArray(a,nx,ny);
```

and this works just as well for nx = 5000, ny = 4000 as it does for nx = 5, ny = 4.

Note that I've missed out some housekeeping here. I've not tested the return values from the malloc() calls, and I've missed out the corresponding free() calls that will be needed once the arrays aren't needed any more.

This is a bit more work than I'd like, but it does let me write a subroutine that can address elements of an array as a[iy][ix] in the way I think I ought to be able to do.

Of course, setting up those arrays is error-prone, so most people doing this sort of thing will package this up one way or another. I use a C++ class I wrote some time ago called ArrayManager, which lets me do:

```
   int nx = 5000;
   int ny = 4000;
   ArrayManager TheManager;
   int** a = (int**) TheManager.Malloc2D(sizeof(int),ny,nx);
   fillArray(a,nx,ny);
```

which I think is neat enough for my purposes. Again, I've missed out the error checking, but the ArrayManager releases all the arrays it controls once it goes out of scope (something to watch, of course), so there's no need for the explicit free() calls - although they do exist.

The Boost multi-array option

The Boost library provides a wonderful range of facilities for C++ programs, and these include a multidimensional array library. Especially if you already use Boost, this is potentially a simple way to use 2D (or higher dimensioned) arrays in a C++ program. Using Boost, the fillArray() routine I've been using becomes:

```
#include "boost/multi_array.hpp"
#include <cassert>

typedef boost::multi_array<int,2> Int2DType;
typedef Int2DType::index Index2DType;

void fillArray (Int2DType& a, int nx, int ny)
{
   for (Index2DType Iy = 0; Iy < ny; Iy++) {
      for (Index2DType Ix = 0; Ix < nx; Ix++) {
         a[Iy][Ix] = Ix + Iy;
      }
   }
}
```

and we can call it using:

```
   int nx = 5;
   int ny = 4;
   Int2DType a(boost::extents[ny][nx]);
   fillArray(a,nx,ny);
```

Note the use of typedef so I can hide the slightly awkward Boost type syntax. This code is really quite neat, and does exactly what you want. There are a few things to note, however:

o If you aren't already using Boost, it's quite a big package to have to install.
o By default, Boost checks array bounds when accessing array elements, which slows code down. To disable this, define the pre-processor symbol BOOST_DISABLE_ASSERTS, either in the code or on the command line when compiling.
o Boost is implemented, as much as possible, using very clever use of C++ templates.

The use of templates is significant. This code doesn't need to be linked against the Boost library, which is convenient - you just need the Boost include files. However, this puts a lot of the responsibility for generating efficient code onto the compiler and its optimisation capabilities, and the relative complexity of the expanded template code can make it hard for the compiler to 'see' exactly what the effect of all the code actually is, and so can constrain just how good an optimisation job the compiler can do.

This shows up in the tests, where Boost code compiled with default optimisation (none) is very slow compared to code compiled with full optimisation, (and slow compared to un-optimised code using other techniques) but with full optimisation it comes very close to the speeds of the more direct 'C: raw' and 'C: Numerical Recipes" options. The compilers (at the moment) don't seem able to see how to apply vector instructions to speed up the Boost code, but that's really a quibble that may not be relevant in the real world. Boost does remarkably well, and you just have to remember to switch in at least -O3 and to disable assertions.

The STL vectors option

I almost didn’t include this in the tests. The C++ Standard Template Library (STL) has a vector type, and you can construct a 2D array as a vector of vectors. I assumed - partly from experience with Boost, partly from memories of early C++ compilers struggling with any template code - that this might work, but would hardly be efficient. I coded it up anyway, but only towards the very end of the testing process. I was rather surprised by the results.

Using an STD vector of vectors, the code for fillArray would look like this:

```
#include <vector>

using std::vector;

void fillArray (vector<vector<int> > &a, int nx,int ny)
{
   for (int iy = 0; iy < ny; iy++) {
      for (int ix = 0; ix < nx; ix++) {
         a[iy][ix] = ix + iy;
      }
   }
}
```

Actually, you don’t even need to pass the array dimensions, since you can get those from the vectors themselves. It would be neater and much safer to do:

```
void fillArray (vector<vector<int> > &a)
{
   for (int iy = 0; iy < a.size(); iy++) {
      for (int ix = 0; ix < a[iy].size(); ix++) {
         a[iy][ix] = ix + iy;
      }
   }
}
```

which would even work if the rows of the array were different sizes.

To call this, all you need would be:

```
   int nx = 5;
   int ny = 4;
   vector< vector<int> > a(ny,vector<int>(nx));
   fillArray(a);
```

I was, frankly, surprised to see the test code using 2D STL vector arrays come in at almost exactly the same speed as the raw C and the Numerical Recipes versions of the code. It seems that once the compiler realises that it’s really just working along a contiguous row of data elements - and it clearly can see this in all these cases (but not quite so clearly in the Boost case) - then it can generate some really efficient code.

C Compilers and results

In my series of tests, I used the four options described in the previous sections, “C raw”, “C Numerical recipes”, “C STL vectors” and “Boost”, splitting Boost up into two sets of tests, one with asserts (essentially bounds checking) switched on, and one with it switched off. I had two

C/C++ compilers, the Clang compiler installed on OS X along with Apple's XCode development system, and the GNU gcc/g++ compiler.

Both compilers accept more or less the same optimisation flags, and default to no optimisation. They then support three specific levels of optimisation, invoked with the -O1, -O2, and -O3 flags. They also accept just -O, which appears to be equivalent to -O1 for g++ and -O2 for Clang. Note that for neither compiler does -O produce the highest level of optimisation. There are also a number of additional flags that control specific aspects of optimisation, but the most useful one I found was -march=native which allows the compiler to use any hardware facilities available on the current machine. (This is the same as for gfortran, which isn't surprising.)

The full set of results can be found in the results tables, but most interesting was the level of improvement that came from using optimisation compared to using the default compiler settings. This is particularly true for the template-heavy Boost code, where full optimisation produced code that ran over 150 times faster than unoptimised code. The code generated at high optimisation was impressive. At -O2 for Clang and -O3 for g++ the compilers started to make use of the X86-64 128-bit vector instructions in ways I'd not expected. (For a description of how this works, see the Assembler section.) With -march=native, both compilers allowed themselves to use the 256-bit vector instructions the laptop's Core I5 chip supported, for another significant speed-up, at least for the raw C, STL vectors, and Numerical Recipes codes. This produced the fastest results of any of the tests. (Just about equalled by Fortran with the same optimisations, and assembler code that did almost exactly the same thing.) The Boost code never got quite that fast, but at -O3 both compilers were doing very creditably with the Boost code.

Using the full range of vector instructions supported by the processor, the compilers managed to improve on their non-vector performance for the raw, STL vector, and Numerical Recipes codes by about a factor 7. Without using these vector instructions, they got more or less the same performance out of all three sets of code.

What does this mean for someone who just wants to use C/C++ to write 2D array-handling code as easily and efficiently as possible? It depends. I don't recommend the 'raw C' approach of emulating 2D arrays with 1D arrays and doing your own index calculations. That's error prone, hard to code, and difficult to follow - you can't even tell the code is really working on 2D arrays. I use the Numerical Recipes approach myself - the code that does the hard work in the subroutines looks perfectly clear, and although the array initialisation looks awkward at first, but you can package up the tricky parts (feel free to use the ArrayManager code if you like), and it can give the best performance. Code using Boost looks neat (apart from some awkward type syntax that can be hidden by typedefs) and the only downside seems to be needing the Boost library if you weren't already using it, and the possibility that the compilers may not be able to eke out the very last bit of performance from it - but may well, depending on the actual problem being coded. Code using STL vectors worked at pretty much the maximum possible speed as well (to my slight surprise), so that also seems to be a perfectly good option, although I don't have much experience with it. I suspect not much existing code is designed to run with a 2D vector array of this type - and a number of useful things you could do with a simple block of memory are probably not available when working with these more complex containers. But there may be other advantages - STD vectors can be extended, for example. All four options have their place (although I'd avoid the raw C option myself).

Some C miscellany - Template routines

In various background readings for this document, I've seen suggestions for what looks like a flexible way of passing a 2D array to a subroutine. Done this way, fillArray() might look like this:

```
template <size_t ny, size_t nx>
void fillArray (int (&a)[ny][nx])
```

```
{
   for (int iy = 0; iy < ny; iy++) {
      for (int ix = 0; ix < nx; ix++) {
         a[iy][ix] = ix + iy;
      }
   }
}
```

At first glance, this looks just the thing! This works for any size array, and the size even manages to get passed to the subroutine without even needing additional parameters. However, look at the main routine and how this needs to be called:

```
   const int nx = 5;
   const int ny = 4;
   int a[ny][nx];
   fillArray(a);
```

which also looks very neat. BUT: this only works for fixed arrays whose size is known at compile time. If you have a couple of arrays of different sizes, the template is used to generate two versions of fillArray(), one for each size of array. It looks neat, but it’s actually just a way of wrapping up code that works on an array of absolutely fixed size - or multiple versions for arrays of different fixed sizes. It’s clever use of templates, and there may be programs that can make good use of this, but it isn’t what I’m looking for here.

More C miscellany - Gaming the system

At one point, I experimented with this version of the subroutine used for the Numerical Recipes case:

```
void subr (float* In[], int Nx, int Ny, float* Out[])
{
   static float* FixedPtr = NULL;
   static int FixedNx = 0;
   static int FixedNy = 0;

   if (Nx != FixedNx || Ny != FixedNy || FixedPtr == NULL) {
      if (FixedPtr) {
         free (FixedPtr);
         FixedPtr = NULL;
      }
      FixedPtr = (float*) malloc(Nx * Ny * sizeof(float));
      FixedNx = Nx;
      FixedNy = Ny;
      for (int Iy = 0; Iy < Ny; Iy++) {
         for (int Ix = 0; Ix < Nx; Ix++) {
            FixedPtr[Iy * Nx + Ix] = Ix + Iy;
         }
      }
   }

   long Nelm = Nx * Ny;
   float* InPtr = &(In[0][0]);
   float* OutPtr = &(Out[0][0]);
   float* Fptr = FixedPtr;
   for (long Ielm = 0; Ielm < Nelm; Ielm++) {
      *OutPtr++ = *InPtr++ + *Fptr++;
```

```
    }
}
```

I hope everyone is suitably horrified by this. It represents a deliberate attempt to ‘game’ the benchmark, trying to speed things up by making use of the knowledge that this routine is going to be called lots of times with test arrays of the same dimensions. The first time it gets called, it allocates an array that it fills with the pre-calculated sums of the array indices. That wastes a bit of time, but on subsequent calls, all it has to do is whip rapidly through the arrays it’s passed, just adding these pre-calculated results to the input array. That should pay dividends for all the subsequent calls. It even does everything in 1D to try to speed things up.

Obviously, this is dreadful code. It even seems morally wrong. Moreover, it’s going to leave that pre-calculated results array around after the last call without ever freeing it.

And how much did it speed things up? Like to guess?

It slowed things down.

Modern processors are fast. Even modern memory is slow. You don’t save time pre-calculating results, if it means you have to read them from an additional array. It takes longer to get that data from memory into the processor than it would for the processor to calculate it on the fly. I think I’d have realised that if I’d thought more about it. Still, now I’ve done it, so you don’t have to.

Python

Python has become enormously popular over recent years, and deservedly so. It’s an interpreted language, with a comprehensive set of packages available. I use it for a variety of housekeeping tasks - the sort of thing I might once have used shell scripts for, and I particularly like using it for text handling applications. The script that runs all the tests described here is a Python program called Run.py. I’m always impressed by the speed Python can do the sort of text file handling I use it for. Python does provide built-in facilities for manipulating lists and arrays, but these will always run relatively slowly in an interpreted language. For much more efficient handling of large arrays, Python has numpy, which provides a very flexible set of facilities for manipulating arrays of different dimensions. These are implemented in C, and should be much faster than anything that relies on interpreted code.

The combination of the flexibility of Python and the speed of numpy has made for something of a winning combination. The downside of numpy is that while it is very fast for ‘whole array at a time’ operations, and even for ‘slice of an array at a time’ operations, if your program doesn’t naturally work with these large blocks of data at a time, it may be hard to get the full benefits that numpy offers.

This test is not one that is going to let numpy shine. What happens to each element depends on the position in the array of that element - the number added to it is the sum of its index values. This is not a natural fit to numpy, or to any array handling package. I’ve seen quite ingenious ways of coding something that would be straightforward to do in code that simply accessed array elements in the usual array[ix][iy] fashion, but which instead shift arrays and add them or subtract them to get the same effect. Because this uses the functions that numpy does fast, the result is fast, but the code can be obscure (some will have comments explaining clearly why it’s doing things this way and how it works; a lot doesn’t).

For Python, I coded three tests that used numpy, one using it to access single elements individually, and then two (much faster) tests that used slightly tricky fiddling with arrays to get the required effect in two different ways. I also coded one test that didn’t use numpy at all.

Python without numpy

This version of the code doesn’t use numpy at all. I coded a test that uses the basic Python list object to form 2D arrays, creating a list whose members are all lists of floating point values. The code (in csubraw.py) looks quite neat:

```
def subr(ina,nx,ny,out):
   for iy in range(ny):
      for ix in range(nx):
         out[iy][ix] = ina[iy][ix] + ix + iy
```

and the 2D ’arrays’ are created in the main routine using a ‘list comprehension’, whose syntax I find a little confusing, but it’s also quite neat:

```
ina = [[0.0] * nx for i in range(ny)]
out = [[0.0] * nx for i in range(ny)]
```

Python with numpy accessing individual elements

Here, the subroutine (in csubnp.py) looks almost the saame as the example above, and it’s pretty clear what it does. The difference is that now ‘ina’ is a numpy array, and you can access its elements using ina[iy,ix] instead of ina[iy][ix] - the give-away that this is a ‘real’ 2D array and not just an array of 1D arrays.

```
def subr(ina,nx,ny,out):
   for iy in range(ny):
      for ix in range(nx):
         out[iy,ix] = ina[iy,ix] + ix + iy
```

The two arrays are created in the main routine - and initialised to zero - using:

```
ina = numpy.zeros((ny,nx))
out = numpy.zeros((ny,nx))
```

The code is really nice and simple, but when you time it you realise you’re not seeing numpy at its best here. This really isn’t how you should be using numpy, at least not for any sort of heavy lifting. Numpy wants to work with arrays all in one go, or at the very least, with slices through arrays.

Python, using numpy vector operations

A second test using numpy (csubnpv.py) is passed the same numpy 2D arrays, but handles them quite differently:

```
def subr(ina,nx,ny,out):
   incr = numpy.arange(nx)
   ones = numpy.array(nx)
   ones.fill(1)
   for iy in range(ny):
      out[iy,0:nx] = ina[iy,0:nx] + incr
      incr = incr + ones
```

Now, this code is much, much, faster than the single element numpy test. This is numpy doing what numpy does well. The problem, such as it is, is that you have to think quite a bit about what it's doing. Take a moment to work it out, perhaps?

If you think about the problem, it consists of adding 0,1,2,3,…nx-1 to the elements of the first row of the input array. Then adding the numbers 1,2,3,4…nx to the elements of the second row. The numbers added to any row are each one more than the numbers added to the previous row. So, create that set of 0..nx-1 values in a numpy vector, and add it to the first row of the input array - which in numpy, is the slice ina[0,0:nx]. And store that in the corresponding slice of the output array. Then, add one to each element of that vector. How do you do that in numpy? By having a vector full of ones and adding them together.

This is the sort of thing you do when numpy doesn't provide exactly the facility you want - you work out a way to do it using what it does provide. What you don't do, at least if performance matters, and it doesn't always - remember that! - is do things element by element. Not if you can help it.

(This is actually a slightly simplified version of the code actually used. The code shown here is fast in the case where nx is much larger than ny, but is much slower if the array has a large number of short rows, instead of a small number of long rows. The code used actually has two branches, one for each case, working orthogonally in the two cases.)

Python using numpy arrays

It is possible to code the subroutine in python in one single line of code using numpy array operations:

```
def subr(ina,nx,ny,out):
   out[:] = ina + numpy.arange(nx)[numpy.newaxis,:] + \
                                numpy.arange(ny)[:,numpy.newaxis]
```

Numpy's arange(nx) creates a vector with the numbers 0,1,2..nx-1, and the newaxis code adds an axis to it, making it a 2D array 1 by nx. Similarly the arange(ny) creates a vector with 0,1,2..ny-1, and the newaxis code makes it a 2D array ny by 1 - note the order of the axes. When they're added, the arrays are conceptually expanded to match, each becoming ny by nx, with the values in the vectors repeated. So there is the original 2D input array, a 2D array with all the X index values, and a 2D array with all the Y index values. Add the three together - something numpy is very good at - and the job's done. (Thanks to Karl Glazebrook, who showed me this code.)

Just putting this in one line doesn't make it faster of course. This one line is doing a lot of work, and is probably creating temporary arrays in memory, all of which adds up to quite a lot of overhead. In fact, whether this code or the numpy vectors code was fastest seemed to depend on which version of Python they were run under.

Python results

The same code was run using both Python 2 and Python 3. The results can be found in the tables, but you would probably not want to use either the basic Python lists code or the numpy individual element code for anything where performance really mattered, although both were very straightforward and clear to code, which may be as important as performance in many cases. The lists code was much faster than the numpy single element code. It's clear that if performance matters, you should try to use numpy the way it was intended, to work on vectors and arrays and not on single elements at a time. However, if performance is critical, then even working on arrays and vectors, numpy is still slower than C/C++ (or Fortran, come to that). Python has you covered even there, however, because it's relatively straightforward to link compiled C/C++ code into Python programs, and that may be the best approach for code where performance matters but you don't want to lose the flexibility of Python.

Rust

I wanted to try Rust ever since I read it was the 'most loved' programming language in the Stack Overflow survey in 2019, making it the fourth year in a row its name came out of that particular envelope. (Python came second. 'Most loved' is not the same as 'most used' language, by the way - that's Javascript.) Rust is intended to be focussed on safety while maintaining high performance, so it seems a good language to look into. The Rustc compiler is a front end for LLVM, which is the back end used by a number of compilers, including Clang.

Rust only supports vectors of vectors, rather than simple rectangular arrays, and what seems the obvious code for the test subroutine is:

```
pub fn csub (input_array: &Vec<Vec<f32>>,nx: usize,ny: usize,
                                      output_array: &mut Vec<Vec<f32>>)
{
   for iy in 0..ny {
      for ix in 0..nx {
         output_array[iy][ix] = input_array[iy][ix] + (ix + iy) as f32;
      }
   }
}
```

'f32' is a single precision floating point quantity, and the Vec<Vec<f32>> syntax is fairly straightforward. The arrays are passed by reference, so they aren't copied, and only the output array is able to be modified, ie is mutable - that's the 'mut' in the declaration.

Rust has a very specific style, including liking its variable names to be in 'snake_case', which isn't my preferred style, and the rustc compiler is the first I can think of that generates warnings for using variable names it doesn't approve of. In the main routine, the arrays are created easily enough:

```
   let mut in_array = vec![vec![0.0f32; nx]; ny];
   let mut out_array = vec![vec![0.0f32; nx]; ny];
```

where 'vec!' is a macro that simplifies the declaration a little, and the arrays are passed to the subroutine using:

```
   crssub::csub (&in_array,nx,ny,&mut out_array);
```

The crssub:: comes from the fact that I defined the csub() routine in a separate file called crssub.rs. The compiler pulls this in automatically during compilation, and appears to inline the subroutine call. I didn't go into other ways of linking the program. Rust appears to perform bounds checking by default and I didn't see a way to switch that off, which goes with its emphasis on safety. Rust does have an 'unsafe' qualifier for code blocks, but this seemed to make no difference. In principle, bounds checking for csub() can be done quickly at the start, looking at the range of index values that will be used to access the vectors and comparing these with their dimensions. I didn't look deeply enough into the generated code to see just what it was doing with bounds checking.

Rustc accepts the usual -O flag, and also has a range of other flags including “-C target-cpu=native” and “-C opt-level=3”, which seem equivalent to -O3 and -march=native for other compilers. By default it does not optimise, and there is a difference of around two orders of magnitude between the speed of optimised and unoptimised code. I would expect that as Rust settles down (it went through a number of early changes to the language definition) and the compilers mature, the performance will improve even more.

R

R is mainly associated with statistics, but it can be used as a general purpose programming language. It supports the concept of a rectangular array, which it quite reasonably calls a ‘matrix’. A straightforward version of the subroutine used in these tests, using individual array element access with the straightforward array[ix,iy] syntax, looks like this:

```
csub <- function(ina,nx,ny) {
   out <- matrix(0.0,nx,ny)
   for (iy in 1:ny) {
      for (ix in 1:nx) {
         out[ix,iy] <- ina[ix,iy] + ix + iy
      }
   }
   return (out)
}
```

R is a fairly strict ‘functional’ language, which manifests itself here in the restriction that data is immutable, meaning you aren’t supposed to pass anything to a subroutine and try to modify it in that subroutine. If you try to - say by passing an existing array to a subroutine hoping the subroutine will modify it - you find you’re modifying a local version of that data which isn’t reflected in the copy of the data in the calling routine. So, this is one language where the subroutine isn’t passed an ‘output’ array. Instead, the subroutine creates an array to receive the result, which it then returns as the function value. Other than that, this code is pretty straightforward. One

Creating the input array in the main routine is simple:

```
   ina <- matrix(0.0,nx,ny)
```

and there’s no need to create an output array to pass to the routine. We just call the routine and the returned value is the output array:

```
   out <- csub(ina,nx,ny)
```

Which is all fine, once you get used to how it works. However, R is an interpreted language, and this code isn’t particularly fast.

This is where the ‘outlier’ code - the really slow code - came from. Looking for ways to speed this up, I wondered about the overheads of creating that output array and returning it every time, and (having a C background), wondered if there was any way to pass an array to the subroutine by reference, so that the subroutine would actually be working on the copy held by the calling routine and not a newly created local array. I imagined this would seem like heresy to any real ‘R’ programmer, but it turns out that R does have such a facility. It’s called a reference class object.

The code to create such a thing looks a bit like creating an instance of a C++ class:

```
ref_matrix <- setRefClass(
   "ref_matrix",fields = list(data  = "matrix"),
   methods = list(
      setData = function(ix,iy,value) {
         data[ix,iy] <<- value
      }
   )
)
out <- ref_matrix(data = matrix(0.0,nx,ny))
```

This ‘ref_matrix’ class includes a method called setData() that can be used to set an element of the data array. It turned out that using it made no difference, but I left it in the code. Now, there is only one copy of the ‘out’ array, and if you pass it to a subroutine, all that’s passed is a single reference to that one copy. The call looks like a call in so many other languages:

```
   csub(ina,nx,ny,out)
```

and the subroutine code looks pretty straightforward too.

```
csub <- function(ina,nx,ny,out) {
   for (iy in 1:ny) {
      for (ix in 1:nx) {
         out$data[ix,iy] <- ina[ix,iy] + ix + iy
      }
   }
}
```

I didn’t expect this to speed things up a lot, but I thought it would help. What I’d not expected was how much it slowed things down! By over three orders of magnitude.

What seems to happen is that each time you make an assignment to any part of such a reference class object, some sort of validity checking takes place, involving something called ‘active binding’. Unfortunately, this involves making a copy of the object. This happens for each assignment - that is, for every assignment to an element of the array, the whole array gets copied. That is a sure-fire way of slowing a program down. Presumably this can be fixed, but it seems to be fairly intricately entwined with the R internals, and it doesn’t seem to bother many people, probably because most people aren’t using reference classes in this way.

This really is an outlier, and not really important, but it does show that you can inadvertently introduce inefficiencies when trying to speed up code. Mostly, changes like this need to be tested! It’s no more than a cautionary tale.

An optimisation that actually worked was to use R’s outer() function to create an array of all the summed Ix,Iy values in a temporary array in one single, relatively efficient operation, and then to just add the two arrays together to produce the required output array:

```
csub <- function(ina,nx,ny) {
   return (ina + outer(1:nx,1:ny,'+'))
}
```

Now that’s neat. This version of csub() works on standard R matrices - none of the reference class stuff - and it shows what can be done with a language that has efficient, flexible, built-in functions than can be brought into play for your application. Outer() fills an array with elements created by running a specified operation (‘+’ here) on the ranges of values given by the other arguments. So that one outer() call creates a 2D array of the required index sums, it gets added to the input array, and returned as the function value. It isn’t mind-blowingly fast, possibly because what outer() is doing here is quite complicated, but it's around four times as fast as the version that works on individual array elements. Bear in mind that this sort of heavy duty large array manipulation isn't really what R was intended for.

Java

By now, Java is a well-established language. It looks quite similar to C, although the way Java programs are compiled and executed in rather different. In 2019 GitHub listed Java as the 2nd most popular programming language (behind Javascript). Java is compiled to a machine-independent byte code, which is then run by a Java virtual machine. This has the interesting effect that the Java compiler, javac, has no optimisation options, since any optimisation is handled by the byte-code interpreter in the virtual machine, usually using just-in-time compilation to machine code.

Java gives the impression of supporting 2D rectangular arrays, although strictly they are arrays of 1D arrays. But they can be passed to a subroutine, and that subroutine can work out the indexing so that individual elements can be accessed using array[iy][ix] syntax. Here's the test subroutine in Java:

```
public static void subr(float in[][],int nx,int ny,float out[][]) {
   int ix,iy;
   for (iy = 0; iy < ny; iy++) {
      for (ix = 0; ix < nx; ix++) {
         out[iy][ix] = in[iy][ix] + iy + ix;
      }
   }
}
```

(There, C - that wasn't too hard, was it? Java can do it!) The arrays are created simply enough using:

```
float in[][] = new float[ny][nx];
float out[][] = new float[ny][nx];
```

and the subroutine call looks like:

```
subr(in,nx,ny,out);
```

In fact, it really isn't necessary to pass nx and ny to the subroutine, since what Java is passing as the input and output arrays are objects that incorporate the data arrays but also have methods that will return the array size, so for example, instead of passing nx, it would be possible - and safer - to use in[iy].length.

And there really isn't that much to say about Java. This code runs, and runs pleasingly quickly. It's up to the speeds I get with C++ at optimisation levels that don't start using vector instructions, even if it doesn't manage to squeeze out the performance C++ can get from those vector instructions. And it's a straightforward language in which to write this sort of code.

There is one interesting quirk of Java that showed up in these tests. In this line in the subroutine:

```
         out[iy][ix] = in[iy][ix] + iy + ix;
```

if I change this to:

```
         out[iy][ix] = in[iy][ix] + ix + iy;
```

the execution time doubles. The only difference is that I'm adding ix + iy instead of iy + ix. And you thought addition was commutative! I only spotted this by accident, through refactoring the code to rationalise the way I used 'x' and 'y' between the various languages. This involved reversing the use of 'x' and 'y' in the Java code, but I didn't bother to change the order of ix + iy since I assumed it made no difference. And I spent a while wondering why the code had slowed down by a factor two. Finally, I swopped the ix and iy in this line, and the speed came back. I have no idea what is going on and I don't read Java byte code. This is some sort of cautionary tale, but I'm not really sure what the message is here.

An interesting point. The fact that Java optimises by default, and clearly has a quite mature optimisation system, means that it provided the fastest execution speed for any language tested when run with default command line settings.

Javascript

Javascript is quite distinct from Java, despite the name, and a distinctly similar syntax. It is an interpreted language with dynamic typing, originally intended for use inside web browsers, but capable of being run separately from the command line. The Javascript version of the test subroutine is this:

```
function subr(inp,nx,ny,out) {
   for (var iy = 0; iy < ny; iy++) {
      for (var ix = 0; ix < nx; ix++) {
         out[iy][ix] = inp[iy][ix] + ix + iy;
      }
   }
}
```

which clearly expects in and out to be 2D arrays. Actually, like Java, these are really arrays of arrays, but in Javascript these have to be set up explicitly, with each 'row' array created separately:

```
var inp = new Array(ny);
var out = new Array(ny);
for (var iy = 0; iy < ny; iy++) {
   inp[iy] = new Array(nx);
   out[iy] = new Array(nx);
}
```

and calling the subroutine is straightforward:

```
subr(inp,nx,ny,out);
```

Although Javascript code is usually run within a browser, the node.js program includes a Javascript interpreter and can be run from the command line, so it was possible to test the Javascript implementation of the test code just like all the other implementations. Node.js doesn't have much in the way of command line flags to control optimisation. it optimises by default, and the —no-opt flag can be used to suppress optimisation. Turning off optimisation slowed down the program by a factor of about 20. The optimised performance was quite impressive for an interpreted language - the interpreter has aspects of a just-in-time compiler, and will take the time to optimise code that is being heavily used.

In addition to the automatic testing, a quick HTML script was put together to allow the Javascript code to be run in a browser. This was not tested exhaustively, but it seemed that in Safari the code ran noticeably faster than with Node.js from the command line, although Chrome ran it at about the same speed as Node.js. Other browsers were not tested. (All this testing takes time, and I was reluctant to put effort into tests that were harder to automate and needed to be run from outside the Run.py overall test script.)

Swift

Swift is the relatively new language from Apple. Originally (this is 1984), Macintoshes were programmed in a version of Pascal, but with the move to OS X came Objective-C, an object-oriented version of C. C was a subset of Objective-C, and interestingly, the object-oriented extensions in Objective-C were completely orthogonal to those in C++ - they had no overlap at all, allowing a mix called Objective-C++, which was a superset of C, objective-C and C++. However, Apple decided they needed something more modern (and less prone to the unsafe pointer code that makes C both powerful and dangerous).

Swift supports arrays of arrays, but not rectangular multi-dimensional arrays, and the original Swift implementations were very slow when it came to multi-dimensional array handling, needing coding trickery (like explicit offset calculations or purpose built 2D classes) to get any sort of efficiency. However, with version 4 of Swift in 2017, the 2D array code generation improved enormously, and these tricks became less efficient than the ‘obvious’, straightforward, way of writing the code.

```
func Subr (Input: Array<Array<Float>>, Nx: Int, Ny : Int,
                         Output: inout Array<Array<Float>>) {
   for Iy in 0...Ny-1 {
      for Ix in 0...Nx-1 {
         Output[Iy][Ix] = Input[Iy][Ix] + Float(Ix + Iy)
      }
   }
}
```

The Swift compiler, swiftc, supports three optimisation options:
-Onone (the default) provides no optimisation.
-O generates optimises code, but includes bounds checking for arrays.
-Ounucecked generates optimised code with no bounds checking.

Swift is the case where simply selecting the right optimisation flags when compiling gave a speed up of nearly 500 times (actually just over 470). Swift is a relatively new language with a relatively new compiler. If the developers want to put more effort into its array handling, they might well do even better. As it is, this is rivalling the C++ compilers for code that doesn’t use X86-64 vector instructions.

Swift has a reasonably straightforward and familiar syntax (at least for C programmers) and can generate fairly efficient code. It is certainly Apple’s preferred language for all their systems. However, at the moment, support for running it on non-Apple systems seems to be sketchy, to say the least.

Tcl

Tcl (Tool Command Language, pronounced ’tickle’) is a general-purpose, interpreted language often used in conjunction with Tk to write GUIs. The semantics are a little unusual, with a program being a set of commands made up of words, with quotes and curly braces etc. being used to delimit words. Tcl doesn’t use ‘=‘ for assignments, and there are substitution rules that apply to how words are evaluated - a variable name with a leading '$' is replaced by the value of the variable. In Tcl, the test subroutine looks like this:

```
proc subr { inary nx ny outary } {
   upvar $inary in
   upvar $outary out
   for {set iy 0} {$iy < $ny} {incr iy} {
      for {set ix 0} {$ix < $nx} {incr ix} {
         set inelem $in($iy,$ix)
         set out($iy,$ix) [expr $inelem + $ix + $iy]
      }
   }
}
```

I've never found this the easiest syntax to work with, but it is very flexible and can be used in interesting ways. The set command sets the value of a variable, and in the line starting "set out($iy,$ix)" sets the (iy,ix) element of an array variable to the result of the expression that follows. If the array in question doesn't exist, one is created and extended as necessary. Really, an array is a container with keys for each item in it, and in this case the keys are formed from the values of the two variables ix and iy. The 'upvar' is a Tcl complication that essentially means that 'in' and 'out' refer to variables (arrays in this case) in the calling routine.

In the main program, the input and output arrays are created simply by setting values to their elements - there isn't a declaration that gives the array dimensions. Just:

```
for {set iy 0} {$iy < $ny} {incr iy} {
   for {set ix 0} {$ix < $nx} {incr ix} {
      set in($iy,$ix) [expr $nx - $ix + $ny - $iy]
      set out($iy,$ix) 0.0
   }
}
```

which sets the elements of out to zero and those of in to some test values based on the index values. Then this is passed to the subroutine with:

```
subr in $nx $ny out
```

As you might imagine from this brief description, this isn't going to be blazingly fast when it comes to array access, and it isn't. Tcl is at its best in a different type of application - heavy duty number crunching is not its forte, and this shows in the test results.

Perl and Perl/PDL

Perl has been a very popular scripting language ever since it was originally developed in the late 1980s. It was designed as a general purpose scripting language, particularly for text processing, but it has been used for a large variety of purposes. Recently it has been overshadowed by Python, but it is still to be found all over - Perl scripts are used to drive all sorts of processes. It was, of course, not really designed for heavy duty number crunching, so this is another language where these tests don't play to its strengths.

Perl is often described as having an inelegant syntax, and its method of passing arguments to subroutines is more like the way command line arguments are passed to programs than the way most languages would do this. Perl puts a '$' in front of a variable name to indicate the value of the variable, and the test subroutine in Perl looks like this:

```
sub csub {
   my @Input = @{$_[0]};
   my $Nx = $_[1];
   my $Ny = $_[2];
   my @Output = @{$_[3]};
   for (my $iy=0; $iy < $Ny; $iy++) {
      for (my $ix=0; $ix < $Nx; $ix++) {
         $Output[$iy][$ix] = $Input[$iy][$ix] + $ix + $iy;
      }
   }
}
```

‘my’ indicates a local variable, and the $_[n] syntax is a way of getting the values of the subroutine parameters. Other than that, the code is relatively straightforward. Perl provides arrays of arrays. An array has to be declared with a leading ‘@‘, and will be expanded as necessary depending on the values of the array indices that get used. So, the calling program creates these arrays using:

```
my @InArray;
my @OutArray;
$InArray[$ny][$nx] = 0.0;
$OutArray[$ny][$nx] = 0.0;
for (my $iy=0; $iy < $ny; $iy++) {
   for (my $ix=0; $ix < $nx; $ix++) {
      $InArray[$iy][$ix] = $nx - $iy + $ny - $ix;
      $OutArray[$iy][$ix] = 0.0;
   }
}
```

The lines that set the extreme elements of the arrays are included first to try to give an indication of the array sizes, but aren’t necessary - the arrays will be expanded as required. The call to the subroutine has to pass the arrays by reference, which is where the backslash characters come in.

```
   csub (\@InArray,$nx,$ny,\@OutArray);
```

The speed this code runs at is fairly typical for interpreted languages. Quite usable, but not if speed is really crucial.

PDL (Perl Data Language) is a set of extensions to Perl designed to provide a number of functions similar to those of IDL, and is specifically geared to astronomy. It provides interfaces to a number of old favourites, such as PGPLOT graphics and FITS I/O. It also provides a comprehensive set of array handling operations, rather similar to what numpy provides for Python.

Rather like with numpy and Python, it is possible to use PDL to access individual array elements. In this case, the subroutine, this time working on PDL arrays rather than ordinary Perl arrays, looks like this:

```
sub csub {
   my ($Input,$Nx,$Ny,$Output) = @_;
   for (my $iy=0; $iy < $Ny; $iy++) {
      for (my $ix=0; $ix < $Nx; $ix++) {
         $Output->slice($ix,$iy) .= $Input->slice($ix,$iy) + $ix + $iy;
      }
   }
}
```

PDL uses ‘.=‘ for assignments, because it isn’t able to overload the usual ‘=‘ operator. Other than that, this code is straightforward. The first line using the ‘@_’ syntax is a shorthand for the code in the ordinary Perl case and sets four local variables to values of the passed arguments. The use of the PDL ‘slice’ facility to access an individual element is a little odd - it makes sense, as a single element is indeed just a 1-element long slice through the array in both dimensions - but suggests that a very flexible feature is being used in what may turn out to be an inefficient way.

In the main routine, the arrays are created (initially set to all zeros) using:

```
$InArray = zeroes($nx,$ny);
$OutArray = zeroes($nx,$ny);
```

and the subroutine call looks quite conventional:

```
   csub ($InArray,$nx,$ny,$OutArray);
```

The suggestion above that this may turn out to be inefficient turns out to be spot on. This is emphatically not the way PDL should be used, producing the longest execution times of any test other than the ‘outlier’ code in R.

To see what PDL can do, the subroutine needs to be reworked to make proper use of the slice facilities, as is done in the ‘PDL vector’ version of the test. Here the main calling program is unchanged, but the subroutine now looks like:

```
sub csub {
   my ($Input,$Nx,$Ny,$Output) = @_;
   $incr = sequence($Nx);
   $ones = ones($Nx);
   for ($iy = 0; $iy <$ Ny; $iy++) {
      $Output->slice(":,$iy") .= $Input->slice(":,$iy") + $incr;
      $incr += $ones;
   }
}
```

This is much faster. It also provides some indication of the flexibility of the slice facilities provided by PDL. Using string arguments allows potentially very flexible ways of specifying slices, at the expense of a slight parsing overhead. The scheme here is exactly the same as that described under “Python using numpy vector operations” and the algorithm is explained there - and in the code in csubpdlvec.pl. (As for the Python case, the code shown here is slightly simpler than the code actually used, which treats the cases nx > ny and ny >= nx separately.)

But for full speed, you can code the whole calculation in one line, using PDL’s sequence() to expand two 1D vectors created using PDL’s sequence() and then threading over the ‘+’ operator to take an 2D outer sum array where each element contains the sum of the X and Y index values. Which is what you want to add to the input array, all in one hit, to get the required output array. (Thanks to Karl Glazebrook for explaining this trick to me.)

```
sub csub {
   my ($Input,$Nx,$Ny,$Output) = @_;
   $Output .= $Input + sequence($Nx) + sequence($Ny)->dummy(0,1);
}
```

The detailed results can be found in the results tables, and show the same sort of progression as that for Python and numpy. You can get a large improvement by using vector and array operations in the way they were intended, but should avoid using systems designed for array access to set individual array elements on a one by one basis. Used the way they were intended, you may not get the speed you could get with purpose-built, compiled code (C/C++ or Fortran) but in most cases the convenience of the interpreted language may compensate for that.

Julia

Julia is a modern (launched in 2012) general purpose language that is intended to be well-suited to scientific computation. It sounds as if the tests performed here are playing to at least some of its strengths.

The subroutine code in Julia looks as straightforward as one might expect:

```
function subr!(in,nx,ny,out)
   for iy = 1:ny
      for ix = 1:nx
         out[ix,iy] = in[ix,iy] + ix + iy
      end
   end
   return
end
```

The '!' at the end of the routine name is a convention used for routines that modify some or all of their arguments. Julia has dynamic typing, so most variables do not need to be declared, and it supports rectangular arrays whose elements are addressed with an array[ix,iy] syntax. This all felt very familiar to a Fortran programmer from way back, and multi-dimensional arrays are stored in column-major order, just as Fortran does. Note that the first index varies fastest in the loop code.

In the main calling routine, the arrays are created using:

```
const in = zeros(Float32,nx,ny)
const out = zeros(Float32,nx,ny)
```

Note that the 'const' does not imply that the contents of the arrays are immutable; it just means in and out - the variable names - aren't going to be reused for something quite different at some later stage. The arrays are passed to the subroutine using:

```
   subr!(in,nx,ny,out)
```

which is all pretty simple.

Julia optimises by default. (It uses a just-in-time compiler, but compiles directly to machine code using LLVM.) It supports the command line flags -O0 (no optimisation), -O1, -O2 and -O3. Although it has command line flags that are connected with the processor type being used, none of these seemed to have any effect - other than that some of the options I tried were rejected. The default appears to be either -O2 and -O3, both of which gave about the same results in these tests. Interestingly, this made Julia second only to Java's execution speed when tested with default command line flags.

I found I liked Julia, which suggests if it is aiming at scientific programmers it may have hit the target. It isn't as fast as Fortran or C/C++ (or even Java) in these tests, but it's pretty close, and can probably be expected to improve with time.

Some useful links:

https://stackoverflow.com/questions/27139437/most-efficient-way-to-access-multi-dimensional-arrays-in-swift
https://en.wikipedia.org/wiki/Comparison_of_programming_languages_(array)
https://stackoverflow.com/questions/8767166/passing-a-2d-array-to-a-c-function
https://insights.stackoverflow.com/survey/2019