**1. Use floating-point numbers if you can, and use them early.**

Of the most common issues that I see when I review slow code is that the programmer has inadvertently asked *Mathematica* to do things more carefully than needed. Unnecessary use of exact arithmetic is the most common case.

In most numerical software, there is no such thing as exact arithmetic. 1/3 is the same thing as 0.33333333333333. That difference can be pretty important when you hit nasty, numerically unstable problems, but in the majority of tasks, floating-point numbers are good enough and, importantly, much faster. In *Mathematica* any number with a decimal point and less than 16 digits of input is automatically treated as a machine float, so always use the decimal point if you want speed ahead of accuracy (e.g. enter a third as 1./3.). Here is a simple example where working with floating-point numbers is nearly 50.6 times faster than doing the computation exactly and then converting the result to a decimal afterward. And in this case it gets the same result.

N[Det[Table[1/(1 + Abs[i - j]), {i, 1, 150}, {j, 1, 150}]]] // AbsoluteTiming

{3.9469012, 9.30311*10^-21}

Det[Table[1/(1. + Abs[i - j]), {i, 1., 150.}, {j, 1., 150.}]] // AbsoluteTiming

{0.0780020, 9.30311x10^-21}

The same is true for symbolic computation. If you don’t care about the symbolic answer and are not worried about stability, then substitute numerical values as soon as you can. For example, solving this polynomial symbolically before substituting the values in causes *Mathematica* to produce a five-page-long intermediate symbolic expression.

Solve[a x^4 + b x^3 + c x + d = 0, x] /. {a → 2., b → 4., c → 7., d → 11.} // AbsoluteTiming

{0.1872048, {{x → -2.20693}, {x → -1.1843}, {x → 0.695616 - 1.27296 &#F74E;}, {x → 0.695616 + 1.27296 I}}}

But do the substitution first, and [Solve](http://reference.wolfram.com/mathematica/ref/Solve.html) will use fast numerical methods.

Solve[a x^4 + b x^3 + c x + d = 0 /. {a → 2., b → 4., c → 7., d → 11.}, x] // AbsoluteTiming

{0.0468012, {{x → -2.20693}, {x → -1.1843}, {x → 0.695616 - 1.27296 i}, {x → 0.695616 + 1.27296 i}}}

When working with lists of data, be consistent in your use of reals. It only takes one exact value to cause the whole dataset to have to be held in a more flexible but less efficient form.

data = RandomReal[1, {1000000}]; ByteCount[Append[data, 1.]]

8000176

ByteCount[Append[data, 1]]

32000072

**2. Learn about Compile…**

The [Compile](http://reference.wolfram.com/mathematica/ref/Compile.html) function takes *Mathematica* code and allows you to pre-declare the types (real, complex, etc.) and structures (value, list, matrix, etc.) of input arguments. This takes away some of the flexibility of the *Mathematica* language, but freed from having to worry about “What if the argument was symbolic?” and the like, *Mathematica* can optimize the program and create a byte code to run on its own virtual machine. Not everything can be compiled, and very simple code might not benefit, but complex low-level numerical code can get a really big speedup.

Here is an example:

arg = Range[ -50., 50, 0.25];

fn = Function[{x}, Block[{sum = 1.0, inc = 1.0}, Do[inc = inc*x/i; sum = sum + inc, {i, 10000}]; sum]];

Map[fn, arg]; // AbsoluteTiming

{21.5597528, Null}

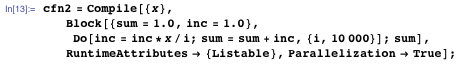
Using Compile instead of [Function](http://reference.wolfram.com/mathematica/ref/Function.html) makes the execution over 80 times faster.

cfn = Compile[{x}, Block[{sum = 1.0, inc = 1.0}, Do[inc = inc*x/i; sum = sum + inc, {i, 10000}]; sum]];

Map[cfn, arg]; // AbsoluteTiming

{0.2652068, Null}

But we can go further by giving Compile some hints about the parallelizable nature of the code, getting an even better result.



cfn2[arg]; // AbsoluteTiming

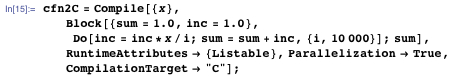
{0.1404036, Null}

On my dual-core machine I get a result 150 times faster than the original; the benefit would be even greater with more cores.

Be aware though that many *Mathematica* functions like [Table](http://reference.wolfram.com/mathematica/ref/Table.html), [Plot](http://reference.wolfram.com/mathematica/ref/Plot.html), [NIntegrate](http://reference.wolfram.com/mathematica/ref/NIntegrate.html" \t "_self), and so on automatically compile their arguments, so you won’t see any improvement when passing them compiled versions of your code.

**2.5. …and use Compile to generate C code.**

Furthermore, if your code is compilable, then you can also use the option[CompilationTarget](http://reference.wolfram.com/mathematica/ref/CompilationTarget.html)->“C” to generate C code, call your C compiler to compile it to a DLL, and link the DLL back into *Mathematica*, all automatically. There is more overhead in the compilation stage, but the DLL runs directly on your CPU, not on the *Mathematica* virtual machine, so the results can be even faster.



cfn2C[arg]; // AbsoluteTiming

{0.0470015, Null}

**3. Use built-in functions.**

*Mathematica* has a lot of functions. More than the average person would care to sit down and learn in one go. So it is not surprising that I often see code where someone has implemented some operation without having realized that *Mathematica* already knows how to do it. Not only is it a waste of time re-implementing work that is already done, but our guys are paid to worry about what the best algorithms are for different kinds of input and how to implement them efficiently, so most built-in functions are really fast.

If you find something close-but-not-quite-right, then check the options and optional arguments; often they generalize functions to cover many specialized uses or abstracted applications.

Here is such an example. If I have a list of a million 2×2 matrices that I want to turn into a list of a million flat lists of 4 elements, the conceptually easiest way might be to [Map](http://reference.wolfram.com/mathematica/ref/Map.html) the basic [Flatten](http://reference.wolfram.com/mathematica/ref/Flatten.html" \t "_self)operation down the list of them.

data = RandomReal[1, {1000000, 2, 2}];

Map[Flatten, data]; // AbsoluteTiming

{0.2652068, Null}

But Flatten knows how to do this whole task on its own when you specify that levels 2 and 3 of the data structure should be merged and level 1 be left alone. Specifying such details might be comparatively fiddly, but staying within Flatten to do the whole flattening job turns out to be nearly 4 times faster than re-implementing that sub-feature yourself.

Flatten[data, {{1}, {2, 3}}]; // AbsoluteTiming

{0.0780020, Null}

So remember—do a search in the Help menu before you implement anything.

**4. Use Wolfram *Workbench*.**

*Mathematica* can be quite forgiving of some kinds of programming mistakes—it will proceed happily in symbolic mode if you forget to initialize a variable at the right point and doesn’t care about recursion or unexpected data types. That’s great when you just need to get a quick answer, but it will also let you get away with less than optimal solutions without realizing it.

[*Workbench*](http://www.wolfram.com/products/workbench/) helps in several ways. First it lets you debug and organize large code projects better, and having clean, organized code should make it easier to write good code. But the key feature in this context is the profiler that lets you see which lines of code used up the time, and how many times they were called.

Take this example, a truly horrible way (computationally speaking) to implement [Fibonacci](http://reference.wolfram.com/mathematica/ref/Fibonacci.html" \t "_self)numbers. If you didn’t think about the consequences of the double recursion, you might be surprised by the 22 seconds it takes to evaluate fib[35] (about the same time it takes the built-in function to calculate all 208,987,639 digits of Fibonacci[1000000000] [see tip 3]).

fib[n_] := fib[n - 1] + fib[n - 2]; fib[1] = 1; fib[2] = 1; fib[35]; // AbsoluteTiming

{22.3709736, Null}

Running the code in the profiler reveals the reason. The main rule is invoked 9,227,464 times, and the fib[1] value is requested 18,454,929 times.

Being told what your code really does, rather than what you thought it would do, can be a real eye-opener.

**5. Remember values that you will need in the future.**

This is good programming advice in any language. The *Mathematica* construct that you will want to know is this:

f[x_] := f[x] =(*What the function does*)

It saves the result of calling f on any value, so that if it is called again on the same value,*Mathematica* will not need to work it out again. You are trading speed for memory here, so it isn’t appropriate if your function is likely to be called for a huge number of values, but rarely the same ones twice. But if the possible input set is constrained, this can really help. Here it is rescuing the program that I used to illustrate tip 3. Change the first rule to this:

fib[n_] := fib[n] = fib[n - 1] + fib[n - 2];

And it becomes immeasurably fast, since fib[35] now only requires the main rule to be evaluated 33 times. Looking up previous results prevents the need to repeatedly recurse down tofib[1].

**6. Parallelize.**

An increasing number of *Mathematica* operations will automatically parallelize over local cores (most linear algebra, image processing, and statistics), and, as we have seen, so doesCompile if manually requested. But for other operations, or if you want to parallelize over remote hardware, you can use the built-in parallel programming constructs.

There is a collection of tools for this, but for very independent tasks, you can get quite a long way with just [ParallelTable](http://reference.wolfram.com/mathematica/ref/ParallelTable.html" \t "_self), [ParallelMap](http://reference.wolfram.com/mathematica/ref/ParallelMap.html" \t "_self), and [ParallelTry](http://reference.wolfram.com/mathematica/ref/ParallelTry.html" \t "_self). Each of these automatically takes care of communication, worker management, and collection of results. There is some overhead for sending the task and retrieving the result, so there is a trade-off of time gained versus time lost. Your *Mathematica* comes with four compute kernels, and you can scale up with[grid*Mathematica*](http://www.wolfram.com/gridmathematica/) if you have access to additional CPU power. Here, ParallelTable gives me double the performance, since it is running on my dual-core machine. More CPUs would give a better speedup.

Table[PrimeQ[x], {x, 10^1000, 10^1000 + 5000}]; // AbsoluteTiming

{8.8298264, Null}

ParallelTable[PrimeQ[x], {x, 10^1000, 10^1000 + 5000}]; // AbsoluteTiming

{4.9921280, Null}

Anything that *Mathematica* can do, it can also do in parallel. For example, you could send a set of parallel tasks to remote hardware, each of which compiles and runs in C or on a GPU.

**6.5. Think about *CUDALink* and *OpenCLLink*.**

If you have GPU hardware, there are some really fast things you can do with massive parallelization. Unless one of the built-in CUDA-optimized functions happens to do what you want, you will need to do a little work, but the *[CUDALink](http://reference.wolfram.com/mathematica/CUDALink/guide/CUDALink.html" \t "_self)* and *[OpenCLLink](http://reference.wolfram.com/mathematica/OpenCLLink/guide/OpenCLLink.html" \t "_self)* tools automate a lot of the messy details for you.

**7. Use Sow and Reap to accumulate large amounts of data (not AppendTo).**

Because of the flexibility of *Mathematica* data structures, [AppendTo](http://reference.wolfram.com/mathematica/ref/AppendTo.html" \t "_self) can’t assume that you will be appending a number, because you might equally append a document or a sound or an image. As a result, AppendTo must create a fresh copy of all of the data, restructured to accommodate the appended information. This makes it progressively slower as the data accumulates. (And the construct data=Append[data,value] is the same as AppendTo.)

Instead use [Sow](http://reference.wolfram.com/mathematica/ref/Sow.html) and [Reap](http://reference.wolfram.com/mathematica/ref/Reap.html). Sow throws out the values that you want to accumulate, and Reapcollects them and builds a data object once at the end. The following are equivalent:

data = {}; Do[AppendTo[data, RandomReal[x]], {x, 0, 40000}]; // AbsoluteTiming

{5.8813508, Null}

data = Reap[Do[Sow[RandomReal[x]], {x, 0, 40000}]][[2]]; // AbsoluteTiming

{0.1092028, Null}

**8. Use Block or With rather than Module.**

[Block](http://reference.wolfram.com/mathematica/ref/Block.html), [With](http://reference.wolfram.com/mathematica/ref/With.html), and [Module](http://reference.wolfram.com/mathematica/ref/Module.html) are all localization constructs with slightly different properties. In my experience, Block and Module are interchangeable in at least 95% of code that I write, butBlock is usually faster, and in some cases With (effectively Block with the variables in a read-only state) is faster still.

Do[Module[{x = 2.}, 1/x], {1000000}]; // AbsoluteTiming

{4.1497064, Null}

Do[Block[{x = 2.}, 1/x], {1000000}]; // AbsoluteTiming

{1.4664376, Null}

**9. Go easy on pattern matching.**

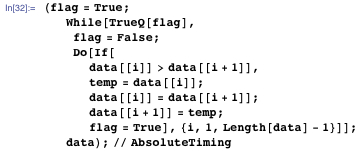
Pattern matching is great. It can make complicated tasks easy to program. But it isn’t always fast, especially the fuzzier patterns like BlankNullSequence (usually written as “\_\_\_”), which can search long and hard through your data for patterns that you, as a programmer, might already know will never be there. If execution speed matters, use tighter patterns, or none at all.

As an example, here is a rather neat way to implement a [bubble sort](http://en.wikipedia.org/wiki/Bubble_sort) in a single line of code using patterns:

data = RandomReal[1, {200}]; data //. {a___, b_, c_, d___} /; b > c → {a, c, b, d}; // AbsoluteTiming

{2.5272648, Null}

Conceptually neat, but slow compared to this procedural approach that I was taught when I first learned programming:



{0.1716044, Null}

Of course in this case you should use the built-in function (see tip 3), which will use better sorting algorithms than bubble sort.

Sort[RandomReal[1, {200}]]; // AbsoluteTiming

{0., Null}

**10. Try doing things differently.**

One of *Mathematica*‘s great strengths is that it can tackle the same problem in different ways. It allows you to program the way you think, as opposed to reconceptualizing the problem for the style of the programming language. However, conceptual simplicity is not always the same as computational efficiency. Sometimes the easy-to-understand idea does more work than is necessary.

But another issue is that because special optimizations and smart algorithms are applied automatically in *Mathematica*, it is often hard to predict when something clever is going to happen. For example, here are two ways of calculating factorial, but the second is over 10 times faster.

temp = 1; Do[temp = temp i, {i, 2^16}]; // AbsoluteTiming

{0.8892228, Null}

Apply[Times, Range[2^16]]; // AbsoluteTiming

{0.0624016, Null}

Why? You might guess that the [Do](http://reference.wolfram.com/mathematica/ref/Do.html) loop is slow, or all those [assignments](http://reference.wolfram.com/mathematica/ref/Set.html) to *temp* take time, or that there is something else “wrong” with the first implementation, but the real reason is probably quite unexpected. [Times](http://reference.wolfram.com/mathematica/ref/Times.html) knows a clever binary splitting trick that can be used when you have a large number of integer arguments. It is faster to recursively split the arguments into two smaller products, (1\*2\*…\*32767)\*(32768\*…\*65536), rather than working through the arguments from first to last. It still has to do the same number of multiplications, but fewer of them involve very big integers, and so, on average, are quicker to do. There are lots of such pieces of hidden magic in*Mathematica*, and more get added with each release.

Of course the best way here is to use the built-in function (tip 3 again):

AbsoluteTiming[65536!;]

{0.0156004, Null}

*Mathematica* is capable of superb computational performance, and also superb robustness and accuracy, but not always both at the same time. I hope that these tips will help you to balance the sometimes conflicting needs for rapid programming, rapid execution, and accurate results.