Algorithm efficiency comes from problem information

Christopher Rackauckas[1]

[1]Affiliation not available

April 17, 2023

the WINNOWER

COMPUTER SCIENCES

Algorithm efficiency comes from problem information

CHRISTOPHER RACKAUCKAS

CORRESPONDENCE:
me@chrisrackauckas.com

DATE RECEIVED:
August 18, 2018

DOI:
10.15200/winn.153459.98939

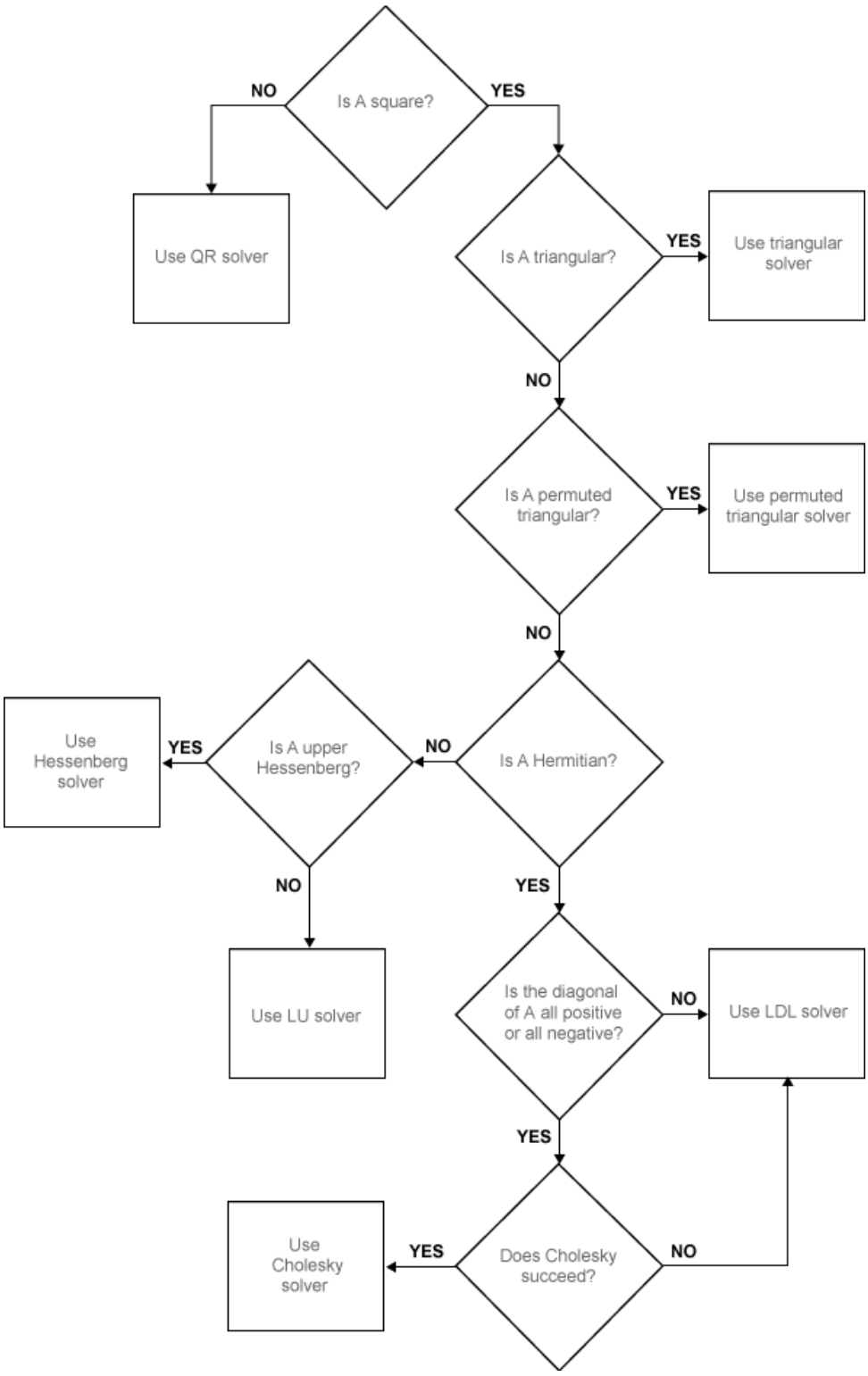
ARCHIVED:
August 18, 2018

CITATION:
Christopher Rackauckas, Algorithm efficiency comes from problem information, *The Winnower* 5:e153459.98939 , 2018 , DOI: 10.15200/winn.153459.98939

This is a high level post about algorithms (especially mathematical, scientific, and data analysis algorithms) which I hope can help people who are not researchers or numerical software developers better understand how to choose and evaluate algorithms. When scientists and programmers think about efficiency of algorithms, they tend to think about high level ideas like temporary arrays, choice of language, and parallelism. Or they tend to think about low level ideas like pointer indirection, cache efficiency, and SIMD vectorization. However, while these features matter, most decent implementations of the same algorithm tend to get quite close in efficiency to each other (probably What does my algorithm know about my problem?

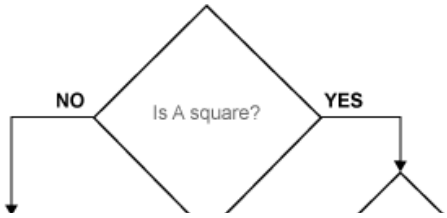
Let me highlight one of the biggest success stories in numerical analysis: solving linear systems. The problem is simple, for what values of x does $Ax = b$? Most people learn early on in their math career that you solve this via Gaussian elimination which is a very nice $\mathcal{O}(n^3)$ algorithm (okay, it's actually not, but it doesn't get much better). In fact, as far as we know right now, solving linear systems has the same asymptotic complexity as matrix multiplication which is currently $\mathcal{O}(n^{2.3728})$. This grows pretty fast, meaning the problem is quite hard, but is it all a game of who can write the most efficient general linear solver?

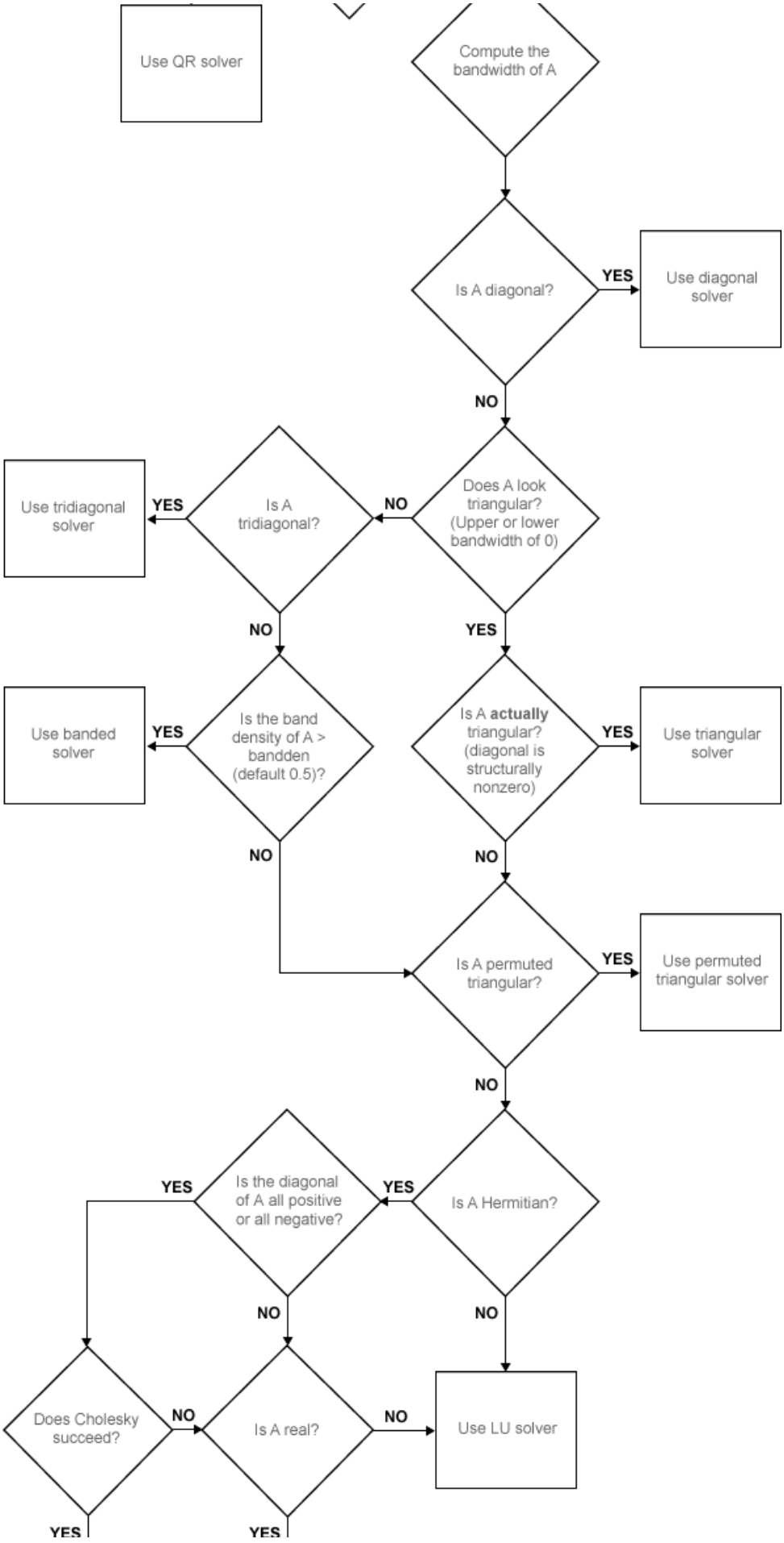
Not even close. The algorithm complexity is the worst case scenario, but in most applications you actually aren't in "the worst case". MATLAB's backslash (A\b) operator solves the linear system $Ax = b$, but what algorithm does it use? Well, Mathworks describes it with a picture:



(picture credit to Mathworks, this is linked directly to their documentation)

Do you think that's it? If so, then you're wrong. That's only when the matrix is full/dense. If it's a sparse matrix, then there's another algorithm for it:




Use QR solver
Compute the bandwidth of A
Is A diagonal?
YES
Use diagonal solver
NO
Does A look triangular? (Upper or lower bandwidth of 0)
NO
Is A tridiagonal?
YES
Use tridiagonal solver
NO
YES
Is A actually triangular? (diagonal is structurally nonzero)
YES
Use triangular solver
Is the band density of A > bandden (default 0.5)?
YES
Use banded solver
NO
NO
Is A permuted triangular?
YES
Use permuted triangular solver
NO
Is A Hermitian?
YES
Is the diagonal of A all positive or all negative?
YES
NO
NO
Is A real?
NO
Use LU solver
Does Cholesky succeed?
NO
YES
YES

(picture credit to Mathworks, this is linked directly to their documentation)

Julia's backslash operator is similar, but what's cool is you can actually inspect Julia's source code and see exactly what it's doing. But it's the same idea. This kind of algorithm is known as a polyalgorithm since it's many different algorithms together. The algorithm starts by checking what kind of type the matrix is and then performing a specialized algorithm on that type, and if no specialized algorithm exists, falls back to Gaussian elimination through a pivoted QR-factorization.

Why would you write an algorithm like this? The trivial answer is because it's faster. If you know the matrix is diagonal then the solution is a $\mathcal{O}(n)$ element-wise division by the diagonal. If you know the matrix is positive-definite then you can use a Cholesky decomposition to solve the equation which takes about half of the operations of Gaussian elimination. However, I think it's better to realize the common thread here. The diagonal method is faster because it knows about properties of diagonal matrices and uses them. The Cholesky decomposition is faster because it utilizes details about positive-definite matrices in order to get rid of possible operations.

THESE SPECIALIZED ALGORITHMS ARE FAST BECAUSE THEY USE MORE "INFORMATION" ABOUT THE PROBLEM

And that's only the start of this entire field of mathematics known as numerical linear algebra. There are iterative methods which can also be used. (Dense) Matrix factorizations are dense and thus require the same amount of memory as the full matrix itself (though some sparse matrix factorizations similarly need the memory for non-zero elements and maybe some more). Iterative solvers require only the ability to do $A*x$ and thus are much more memory efficient and can scale well on sparse matrices. This also makes them easily to parallelize across large clusters and thus are the method of choice for large solving large sparse systems that arise from things like PDEs. Even then, you can still make it more efficient by choosing a preconditioner, but good choices of preconditioners are dependent on, you guessed it, properties of your A or the equation it is derived from.

As you can see, every significant step of this tool chain is about baking more information about the problem into the solution methods. This is why I would suggest that one thinks about the amount of information an algorithm contains about a problem since that's where the true gains are. Let's take another example.

ARE NEURAL NETWORKS "EFFICIENT"?

If there's one area of computational data science that everyone is excited about right now, it's neural networks. The quickest way to explain a neural network is that it is a computationally efficient way to approximate any mapping from inputs to outputs, $f(x) = y$. The f that it creates uses a lot of matrix multiplies which are easy to GPU-parallelize, and the deep in "deep neural networks" simply is the application of more matrix multiplies to get a better approximation of f. Movie recommendations are where you take in x = (data about movies the person previously watched) and then y = (the score that you think they'd give to other movies), and you use enough data to get a good enough approximation for the movies they would like and spit out the top few. Classification problems like "does this picture contain a bird?" is just about creating a mapping between x = (a picture represented by its pixel brightness), to y = 1 or 0 (yes or no: it contains a bird). You train this on enough data and it's correct enough of the time. Amazing, right?

The next question people like to ask is, what neural network frameworks are good for these problems? TensorFlow, PyTorch, KNet.jl will all auto-GPU you things for you and in the end all give around the same performance. Of course, package developers will fight over these 2x-5x differences over the next

decade, showing that their internal setup is good for these problems, while others will show it's not good for these other problems. But what this doesn't tell you is whether you should be using a neural network in the first place.

(Yes, you can use these computational tools for things other than neural networks, but let's ignore that for now)

If you think about the point of this blog post, it should trigger something in you. I mentioned that neural networks can approximate any function $f(x) = y$... so what does the neural network "know" about the problem? Pretty much nothing. The reason why neural networks are popular is because they are a nice hammer that you can use on pretty much any problem, but that doesn't mean it's always good. In fact, because you can use a neural network on any problem it pretty much implies that it won't be great on any problem. That's the fundamental trade off between specificity and generality! My friend Lyndon White displayed this very nicely when he showed 7 different ways to solve a cla\nssification problem in Julia. Notice that the neural network method is dead last in efficiency. Neural networks aren't bad, but they just don't know anything about a problem. If the problem is linear, then a linear regression or linear SVM will DDDDOMINATE on the problem! If it's a classification problem, then algorithms which incorporate knowledge about what a classification problem means, more so than "spit out y in [0,1] via sigmoid and say yes if y is greater than some threshold", will do better. For this domain, it includes things like decision trees. For movie recommendation problems, factorization machines more succinctly capture our internal model of how a user's movie ratings should be related, and thus it's no surprise that they tend to win most of the Netflix prize type of competitions. If you want more information, SciKitLearn has a great chart for choosing methods.

Note that neural networks can be made problem-specific as well. Convolutional neural networks are successful in image processing because they add an extra constraint about the relations of the data, specifically that small "stencils" of the larger matrix (i.e. groups of nearby pixels) should be interrelated. By adding that kind of information into the structure of the neural network and its optimization algorithms, this then becomes a very efficient tool for tasks where the data has this structure.

So neural networks are not useless even though they are not always efficient. They are extremely useful since they can be applied to pretty much any problem. But the general neural network architectures (like deep feed-forward neural networks) lack knowledge about the specific problems they are approximating, and so they cannot hope to be as efficient as domain-optimized algorithms. How do you make them better? You introduce ideas like "memory" to get recurrent neural networks, and it's no surprise that the research shows that these methods are more efficient on problems which have some memory relation. But the generality of neural networks leads me to my favorite example.

ALGORITHM SPECIFICITY FOR DIFFERENTIAL EQUATIONS (OKAY, THIS GOES INTO MORE DEPTH!)

I spend my time developing new methods for (stochastic) differential equations and developing differential equation solving software. There are hundreds and hundreds of different algorithms for finding the function $u(t)$ whose derivative satisfies $u'(t) = f(t,u(t))$ where f is the data given by the user starting to t_0 and ending at t_f (this is the definition of an ODE BTW. If this is new to you, think I'm given a model f which describes how things change and I want to compute where I will be in the future). And again, solving this problem efficiently is all about incorporating more information about the problem into the solver algorithm.

Just as a interesting note, you can solve a differential equation with a neural network, but it's not good. I was hoping that the efficiency gained by using optimized GPU-libraries would be enough to overcome the efficiency lost by not specifying on the problem, but it wasn't even close and the end result was that it was difficult for a deep neural network to solve non-stiff nonlinear problems like the Lotka-Volterra equations which standard differential equations software solve in microseconds. But what's more interesting is why it failed. The neural network simply failed to understand the temporal dependencies of the problem. If you look at the failed solutions on the blog post, you see what happens is that the

neural network doesn't know to prioritize the early time points because, well, it's obvious in a temporal model that if you are wrong now then you will be wrong in the future. But the neural network sees each time point as an unconnected matrix and from there the problem becomes much harder. We could try convolutional nets or recurrent nets, or bias the cost function itself, but this is still trying to get around "the information problem" which differential equations don't have.

With a differential equation, we are modeling reality. In many cases, we know things about that reality. We might know that the trajectory of our rocket is smooth (it has all derivatives), and so we can develop high order algorithms which require 9 derivatives of the user's f and it's not surprise that these Vern algorithms benchmark really well. But is it the best integrator? No, there's no such thing. First of all, these methods are only for "non-stiff" ODEs, that is an ODE which has a single timescale. If there are multiple timescales, then one can show that these are "unstable" and require a very small time step. So what's the best algorithm?

What I try to highlight in the post on differential equation libraries is that a large choice of methods is essential to actually being efficient. For example, many only have a few integrators, any in many cases this is simply multistep methods (because integrators like LSODE and VODE, provided in ancient Fortran code, have an option to change between stiff and non-stiff problems). But are these actually good methods? That's not a good way to look at it. Instead, look at what these multistep methods mean. The form for the BDF2 method is:

$$y_{n+2} - \frac{4}{3}y_{n+1} + \frac{1}{3}y_n = \frac{2}{3}hf(t_{n+2}, y_{n+2})$$

What are the advantages of this algorithm? Well, because it uses two past data points (assume you already know y_n and y_{n+1} and want y_{n+2}), you only have to solve an implicit function for the variable y_{n+2}. If the cost of evaluating the function is high (as is the case for example with large PDE discretizations), then you want to evaluate it less so this does well! Since this only has a single f call per update, that's doing quite well.

However, it made a trade off. In order to decrease the number of function evaluations, it requires more than one previous data point. It also requires that "nothing happened" between those two points. If you have some discontinuity for example, like modeling a ball when it bounces or modeling the amount of chemicals in a patient and you give a dose, then the previous data is invalid. Adaptive order algorithms like LSODE or CVODE go back to the more error prone backwards Euler method

$$y_{n+1} - y_n = hf(t_{n+1}, y_{n+1})$$

to compute one small step (it has to be small to make the error low since the error is $\mathcal{O}(\Delta t)$), then uses that step as the previous step in a BDF2, then goes up to BDF3 etc., where each higher method requires more previous data and has a lower error order. But if you're hitting events quite often (Pk/Pd simulations where a drug dose happens every few hours), notice that this is really really bad. Like, awful: you might as well just have had an implicit Euler scheme!

And not only that, the assumption that this method is efficient requires that f is costly. Other methods, like Rosenbrock or (E)SDIRK methods, make the trade off to take more evaluations of f to get less error, and in turn use this decreased error to take larger time step (and thus less steps overall). And so it's not surprise that these methods benchmark better on small system and even "medium sized systems", while the BDF method benchmarks as more efficient on larger systems with a more expensive function evaluation (the Rosenbrock methods are things like Rosenbrock23 and Rodas4, and the BDF method is CVODE_BDF). Again, this is nothing more than incorporating more details about the problem into the choice of solution method.

This is why having a wide variety of methods which incorporate different types of problem information is really what matters for efficiency. IMEX methods are a great example of this because instead of the user specifying a single ODE $u' = f(t,u)$, the user specifies two functions $u' = f_1(t,u) + f_2(t,u)$ where one of the functions is "stiff" (and thus should be solved implicitly by things like Rosenbrock or

BDF methods) while the other is "non-stiff" (and thus should be solved by explicit methods like the Verner method). By specifying this split form, libraries like ARKODE or DifferentialEquations.jl can utilize algorithms which incorporate this information and thus will be more efficient than any of the previously mentioned methods on appropriately split problems. Another example are symplectic integrators which incorporate the mathematics of the underlying symplectic manifold into the solver itself, for problems which have a symplectic manifold. What kinds of problems lie on a symplectic manifold? Well, those arising from second order differential equations and physical Hamiltonians like N-body problems of astrodynamics and molecular dynamics. These methods, by utilizing a fundamental property of the problem specification, can noticeably reduce the amount of drift in the energy and angular momentum of the approximated solution and make the resulting simulations closer to reality. The changes an algorithm choice can make can be orders of magnitude. In the paper I showed earlier on methods for stochastic differential equations, incorporating the idea of time correlation and interpolation of Brownian motion (the B\nrownian bridge) to build an adaptive method resulted in an average of 100,000x less time steps to compute solutions to the SDE, so much that I couldn't even benchmark how long it would take for algorithms which didn't make use of this because they could not finish! (In my next paper, I am able to estimate that it would take almost 6 years to solve this problem vs the 22 seconds we're at now). Choice of language, cache efficiency, etc. pale in comparison to proper algorithm choice.

THE JULIA PROGRAMMING LANGUAGE

I hope I am getting across the point that the way to evaluate algorithms is by the information they use about the problem. This way of thinking about algorithms extends very far. I show in another blog post that the Julia programming language achieves C/Fortran levels of efficiency because it allo\nws the LLVM compiler to have total type information about your program, as opposed to more dynamic languages like R/MATLAB/Python. Julia is specifically designed so that compilers can at compile-time compute as much type information as possible to essentia\nlly build the C code you would have written. When you understand the information that is being optimized on, these performance differences aren't magical anymore: it's just using more information, more specificity, and getting more performance out because of that.

In fact, I have to put a little blurb out here for the Julia programming language as well. Its multiple dispatch and abstract typing system is a very nice way to organize algorithms by their information. Let's take how Julia's linear algebra works. For a standard dense matrix, the backslash operator internally calls factorize(::AbstractMatrix) (and A_ldiv_B!, which is an inplace way of writing A\b) which I showed above is a polyalgorithm. But if you call factorize on some special matrix type, like a Diagonal or Tridiagonal matrix, it uses a specialization of factorize or A_ldiv_B! to perform the most efficient algorithm for that type of matrix. Then in user codes and packages, other people can define a matrix type MyMatrix $J*x$ can be computed directly from a function call (this is quite common in PDEs), and you don't need to re-write your own. Instead you can make my generic Rosenbrock23 algorithm compile a code which does that... even though it was never written to know what kind of method factorize should be. This kind of "opening up of the black box" and being able to optimize pieces of it via information injection for your specific problem is a powerful tool for optimizing numerical algorithms to large scale problems. Given this post I hope it's easy to understand why I think this will (and already has started to) lead to a new generation of efficient implementations. (Okay okay, you can do generic programming with C++ templates, but you can't make me do it! Look at this "gentle introduction" and you'll be in my boat asking for a high-level language designed to have that information in mind, and I'll point you to Julia)

MORAL OF THE STORY

Getting your perfect caches lined up is small potatoes. BLAS and LAPACK are well-known super fast C linear algebra libraries. They are extremely fast not just because they are bit twiddling with assembly (though they are), but they are super fast because they incorporate information about floating point precision and use that to their advantage. Neural networks, ODE solvers, etc. are all just tools for

problems. The more specific they are to the problem you're trying to solve, the more efficient they can be. So next time you're looking to optimize some numerical codes, don't ask "what can I do to make the computation more clean/efficient", ask "what information about the problem is the solution method not using?" That's where the big potatoes come from.