Judging Complex Numerical Methods

One weird trick to evaluate numerical methods for solving complex-valued systems

Abstract

Thinking a little bit about how to numerically solve complex-valued PDEs, we can devise a shorthand way of figuring out whether or not the method we'd like to use to solve it will yield nonsense results. Nonsense in this case being defined as: explodes or decays. By making sure that complex-valued PDEs are only solved by unitary operators—regardless of how we discretize space or time—we can weed out many numerical methods that obviously won't work. This includes all purely implicit or explicit methods. An appendix with an example that matches up well with a simple quantum system, the particle-in-a-box, shows that our line of thinking isn't entirely ridiculous.

The problem with math in school is that there's this obsession with finding neat and tidy analytical solutions to problems, even as the equations that describe these problems get increasingly complicated. It's gotten so bad, that there's even a million-dollar prize if you can prove solutions for Navier-Stokes' equations exist one way or the other. From my perspective this is bonkers simply because guys like me invented computers so that we didn't have to worry about the math. Do we really need fancy Greek when we can print out fancy graphs? Yes: because should they exist, analytical solutions are fun and neat and useful.

When that's not possible though, we get to turn to computers. This is lots of fun because math in computers is how we got video games to run in 4K. However, even with the most powerful hardware there are times when things just don't end up working the way we expect them to. Here's an example: when the answers to our math problems necessarily have to be complex-valued; when complex numbers aren't just a fancy way of simplifying a problem, i.e. the equations of quantum mechanics, e.g. Schrodinger's equation.

Normally what people do when faced with complex numbers and computers, is they divide the math up into two different pieces, real and imaginary. But that just means when you're dealing with differential equations, you now have a coupled system of equations to solve. At this point, what's the best way of solving it? Do you set up a pseudo-FDTD method, with things separated in space or time, and influencing one another? Is that reasonable; is that good; is that interesting? Who knows!

What if instead we assume that complex numbers are an inborn part of whatever computer we're dealing with, and just blunder ahead; first with basic ways of computerizing the math, and then with increasing complexity? Let's find out.

To keep things relatively simple, let's start out with a general differential equation that we might want to solve that we know requires complex-valued solutions[1].

$$j\frac{\partial\psi}{\partial t} = a\nabla^2\psi + \mathbf{b}\cdot\nabla\psi + c\,\psi$$

Where ψ is what we're trying to find solutions for, *a* is a constant, **b** and *c* can be functions of space and time. To make things simpler, we'll take a page out of a physics textbook and compact the right-hand side such that:

$$j\frac{\mathrm{d}\psi}{\mathrm{d}t} = \mathrm{H}\psi$$

Which probably has a pretty straightforward solution if you give yourself easy initial conditions, and then boundary conditions where everything just goes to zero at spatial infinity. Then again, all it really does is describe a locally

[1] David Bohm. *Quantum Theory*. Ch. 4.5; Pg. 84. New York, NY: Dover Publications, Inc., 1989.

rotating phase change, instantaneously, across all space. But if we have interesting initial conditions, can we solve this numerically so that we can learn something? Without separating real and imaginary components, what if we take the simplest approach and explicitly discretize things so that we get:

$$j\frac{u^{t+1}-u^{t}}{\Delta t}=\mathrm{H}u^{t}$$

Where we've replaced ψ with *u* as that seems to be the style for numerical analysis. Now since we're not looking to get an analytical solution for things, just see how everything evolves over time (and maybe converge), all we really want to do is solve repeatedly as time evolves. To do that, we iterate a bunch of times using:

$$u^{t+1}=\left(1-j\mathrm{H}\Delta t\right)u^{t}$$

Now, the dynamics of the system will ultimately be determined with whatever's contained within H—even if we assume this is a simple ODE instead of a more complex PDE. However, we can say one thing for certain: no matter what we're doing this solution is going to blow up to infinity rather quickly. Why? Since this is a complex function, we can tell by taking the magnitude of the solution after, e.g., *n* steps, with an initial condition when *t* is equal to zero:

$$|u^{n}|=\left(1+(\mathrm{H}\Delta t)^{2}\right)^{\frac{n}{2}}|u^{0}|$$

Each time we march forward in time to see how our initial function evolves, it gets just a little bit larger, but as it grows exponentially, it will blow up pretty quick. Even if we were to use a more advanced explicit numerical method like Runge-Kutta, the magnitude of the function would still increase exponentially by the same amount as we iterate. If we try to use a strictly implicit method instead, we would get something that looks like:

$$u^{t+1}=\left(1+j\mathrm{H}\Delta t\right)^{-1}u^{t}$$

But this has the opposite problem in that it very quickly decays down to zero. If instead we use an implicit-explicit method, we still have to be careful, because depending on how it's implemented the same issues can come up; either an exponential explosion or decay. However, if we're clever and use something like, say, the trapezoid rule to effectively average the simple explicit and implicit methods (i.e. the Crank-Nicolson method), we can set the problem up as:

$$j\frac{u^{t+1}-u^{t}}{\Delta t}=\frac{1}{2}\mathrm{H}(u^{t+1}+u^{t})$$

Which after re-arranging, allows us to iterate by solving:

$$u^{t+1}=\left(1+j\frac{1}{2}\mathrm{H}\Delta t\right)^{-1}\left(1-j\frac{1}{2}\mathrm{H}\Delta t\right)u^{t}$$

It isn't immediately obvious what's going on here, but we can see that it solves both of our previous problems as the exponential decay perfectly cancels out the exponential explosion. To help us figure out what's going on, we can abuse some notation by making the following approximation:

$$\mathrm{e}^{jx}=\cos(x)+j\sin(x)\approx 1+jx$$

And then use this to simplify things such that:

$$u^{t+1}=\mathrm{e}^{-j\mathrm{H}\Delta t}u^{t}$$

But we cannot use this to actually do any solving, as it's impossible to justify the small angle approximation for anything but to see if we can figure out what's going on. If we want to solve it for real, we have to convert all of the differential operators into their appropriate matrices depending on the discretization scheme we're using; e.g. central finite differences. In one dimension, we're then left with a single linear algebra problem to solve, of the form **A x** = **b**, for each time step of interest. In two dimensions, we effectively have to solve a whole series of linear algebra problems; three dimensions requires even more!

So, what is going on? It seems that for a whole class of differential equations, the dynamics require wave-like motion, where the system just rotates its phase as time goes on. In quantum mechanics they call this a unitary transformation, or operator, and this one happens to look like the Hamiltonian operator.

Now, this equation is nonsense for a whole host of reasons, because H is just hiding a whole bunch of differential operators and other various functions or constants, but if you discretize it completely it makes a little bit more sense. However, if we ignore that for a while, maybe we can use it to quickly analyze and classify *types* of numerical methods for their suitability for integrating complex-valued differential equations. For example, if we want to analyze or integrate a basic two-dimensional system that only has a Laplacian term, we can use something like the Douglas-Rachford[2] method:

$$j\frac{u^{*}-u^{t}}{\Delta t} = a\left(\delta_x^2 u^{*} + \delta_y^2 u^{t}\right)$$
$$j\frac{u^{t+1}-u^{*}}{\Delta t} = a\left(\delta_y^2 u^{t+1} - \delta_y^2 u^{t}\right)$$

Where *a* is a constant, and the deltas are a stand-in for a numerical derivative using central differences. This is a two-step method that first implicitly solves for an intermediate solution along the x-direction, and then uses that to find the solution at the next time step, solving implicitly along the y-direction. It's a modification of the standard alternating direction implicit (ADI) method, supposedly gets better results, but for a complex function it just won't work. To help us see why, we first simplify things again with a quick substitution:

$$\mathrm{H}_x = a\delta_x^2,\ \mathrm{H}_y = a\delta_y^2$$

And then we re-arrange things but add in our terrible approximation, we're left with:

$$u^{*} = \mathrm{e}^{-j(\mathrm{H}_x+\mathrm{H}_y)\Delta t}u^{t}$$
$$u^{t+1} = \mathrm{e}^{-j\mathrm{H}_y\Delta t}\left(u^{*} + j\mathrm{H}_y\Delta t\, u^{t}\right)$$

Substituting the first step into the second, we can see that:

$$u^{t+1} = \left(\mathrm{e}^{-j(\mathrm{H}_x+2\mathrm{H}_y)\Delta t} + j\mathrm{H}_y\Delta t\,\mathrm{e}^{-j\mathrm{H}_y\Delta t}\right)u^{t}$$

Now, keeping in mind that what we've done is mostly outrageous if we remember that even in one dimension, we're really solving a linear algebra problem, so everything is in terms of vectors and matrices. However, now that we're looking at a problem in two dimensions, we're really solving a whole series of linear algebra problems; effectively dealing with a matrix of matrices and a matrix of vectors so... that makes sense if you put it in your computer correctly.

[2] Sandip Mazumder. *Numerical Methods for Partial Differential Equations*. Waltham, MA: Elsevier Inc., 2016.

Remember, the reason we did all of this was to see if this method could be used to integrate a complex function, and I think it obviously can't be for two reasons. First, if we focus on the first term on the right-hand side of the equation:

$$\mathrm{e}^{-j(\mathrm{H}_x+2\mathrm{H}_y)\Delta t} u^t$$

It appears as though the initial function has its phase rotated at least two times faster than we would expect from the earlier one-dimensional analysis for a properly conditioned method. Truthfully, it was trying to test a basic solver using this method where I noticed a 2-3x phase rotation, inspiring me to look a little deeper into what was going on.

Second, if we rewind a bit by undoing our terrible approximation to remember which pieces should properly be, "matrix," inverses, this term can be re-written as:

$$(1+j\mathrm{H}_y\Delta t)^{-1}(1+j\mathrm{H}_x\Delta t)^{-1}(1-j\mathrm{H}_y\Delta t)\, u^t$$

Looking at this, since there are more inverse terms than not, we can see that as time marches on, the magnitude of our solution will decay exponentially as:

$$|u^n| \approx (1+(\mathrm{H}_x\Delta t)^2)^{-\frac{n}{2}}|u^0|$$

Which was also something that I discovered when playing around in MATLAB. Ironically, the fix for this issue is to go back to a standard ADI method, such that:

$$j\frac{u^*-u^t}{\Delta t/2} = a(\delta_x^2 u^* + \delta_y^2 u^t)$$

$$j\frac{u^{t+1}-u^*}{\Delta t/2} = a(\delta_x^2 u^* + \delta_y^2 u^{t+1})$$

Following through on all of the terrible substitutions to get to the end of the analysis, we can see that the phase seems to be rotated at the correct rate, and the magnitude does not change at each time step:

$$u^{t+1} = \mathrm{e}^{-j(\mathrm{H}_x+\mathrm{H}_y)\Delta t}\, u^t$$

This is the end goal of using this approach to judging numerical methods for the suitability of integrating complex-valued functions: the output of the method for a single time step needs to depend solely on the input function multiplied by a local phase rotation. Now we can actually use it to do something slightly new, like start adding terms so that we can numerically integrate fancier systems. For example, what if we want to add a potential term? For a simple 1D problem, we could guess something like:

$$j\frac{u^{t+1}-u^t}{\Delta t} = \frac{1}{2}\mathrm{H}(u^{t+1}+u^t) + \frac{1}{2}V(u^{t+1}+u^t)$$

Where H is as we've defined it poorly before (including all spatial derivatives), and *V* is some potential that is a function of space. Following through with our analysis as above, this then comes out to be:

$$u^{t+1} = \mathrm{e}^{-j(\mathrm{H}+V)\Delta t}\, u^t$$

Which is exactly what we're looking for. Just remember that we can't actually use this to solve anything unless we put it back in a form of a matrix inverse multiplied by another matrix. This is because in order for us to put this into a computer, we would have to represent everything as matrices and vectors. While H, as a mostly differential operator, will obviously get us a matrix defining how each element of *u* mixes with each other element, *V* appears as a standard function (returning a vector) so it would have to be added to H as diag(*V*). Extending this to the two-dimensional case, we do notice one nuance:

$$j\frac{u^* - u^t}{\Delta t/2} = \mathrm{H}_x u^* + \mathrm{H}_y u^t + \frac{1}{2}V(u^* + u^t)$$

$$j\frac{u^{t+1} - u^*}{\Delta t/2} = \mathrm{H}_x u^* + \mathrm{H}_y u^{t+1} + \frac{1}{2}V(u^{t+1} + u^*)$$

Why don't we have the same symmetry of, "averaging," of implicit and explicit numerical methods like we do in the one-dimensional case? That's because we've split this particular method up into two steps, for all of the same reasons we would use ADI in the first place: if the equations work out, it's easier to solve tri-diagonal systems. If we were to use the standard Crank-Nicolson method for a two-dimensional system, our analysis would still show that things are nicely balanced and we get the answer that we would want:

$$u^{t+1} = \mathrm{e}^{-j(\mathrm{H}_x + \mathrm{H}_y + V)\Delta t}\, u^t$$

Showing that nothing is going to explode, or have its phase rotated too quickly, as time is advanced. The only real complication with everything we've done so far is source terms. It's handy that in quantum mechanics there aren't really any source terms, but if you want to couple certain systems together, e.g. via magnetic fields, you will need source terms. The trick there is to make sure that you add sources to each step of your algorithm in the same proportion, equally, so that they're properly represented in both implicit-explicit integrations.

In conclusion, the general algorithm can be stated as:

1) Pick a numerical method,
2) Pick a discretization scheme,
3) Re-arrange and collect terms for the current time step and the next time step,
4) Flip any terms of the form (1+*j* x) into $e^{j\,x}$,
5) Treat them as you would normally; solve for the next time step,
6) If the result is a unitary transformation representing a local phase rotation: you're good to go!
7) If it blows up to infinity or decays exponentially, try again.

This way, when you encounter a fancy or more efficient numerical method in the wild, and you want to see if it can be used to integrate complex-valued PDEs, you can quickly determine whether or not to waste your time figuring out how to get it into the computer.

Appendix: What Is the Matrix?

This is fun to think about: if we're looking for a unitary transformation representing a local phase rotation, what are we actually doing at each time step of the numerical method? The easiest way is with a simple example, the quantum particle-in-a-box[3]. Using atomic units, the equation that defines the particle-in-a-box is just Schrodinger's equation with a potential (in hartree atomic units):

$$j\frac{\partial\psi}{\partial t} = -\frac{1}{2}\frac{\partial^2\psi}{\partial x^2} + V\psi$$

[3] https://en.wikipedia.org/wiki/Particle_in_a_box

Where *V* is a potential that's zero within some region, but infinity everywhere else. From theoretical analysis, we know that we can solve this system relatively completely, with energy levels given by:

$$E_n = \frac{n^2\pi^2}{2L^2}$$

Where *L* is the total width of the region where the potential is zero, and *n* is the energy level, starting at one. We also know that the wave function for the stationary states that rotate at these energy levels can be determined by:

$$\psi_n(x) = \cos(\frac{n\pi}{L}x)$$

Where we've assumed that the zero-potential region is centered at *x* = 0. If we discretize this using central finite differences and Crank-Nicolson, we can turn it into something like this:

$$j\frac{u^{t+1}-u^t}{\Delta t} = -\frac{1}{4}\delta_x^2(u^{t+1}+u^t)+\frac{1}{2}V(u^{t+1}+u^t)$$

Where u^t is a vector representing our initial function that we want to see march along in time, δ_x^2 is a matrix that represents the second derivative (via central differences) of a vector that's multiplied on the right, and V is a matrix where the diagonal values represent a vector of values defined by our potential function. In this case, in the region of interest *V* is zero, but we have to make sure that our function goes to zero at the boundary of our region. If we then re-arrange everything, we can solve for each time step using:

$$u^{t+1} = \left(\mathbf{I} - j\frac{1}{4}\delta_x^2\Delta t + j\frac{1}{2}V\Delta t\right)^{-1}\left(\mathbf{I} + j\frac{1}{4}\delta_x^2\Delta t - j\frac{1}{2}V\Delta t\right)u^t$$

In quantum mechanics, we would say that we're finding the wave function at the next time step by multiplying the initial function u^t by the Hamiltonian operator, which is a unitary operator that advances the wavefunction in time. Since in this case, this operator is independent in time, we can investigate it even more by treating this as an eigenvalue problem with:

$$\hat{H}\mathbf{v} = \lambda\mathbf{v}$$

Where the eigenvalues are given by:

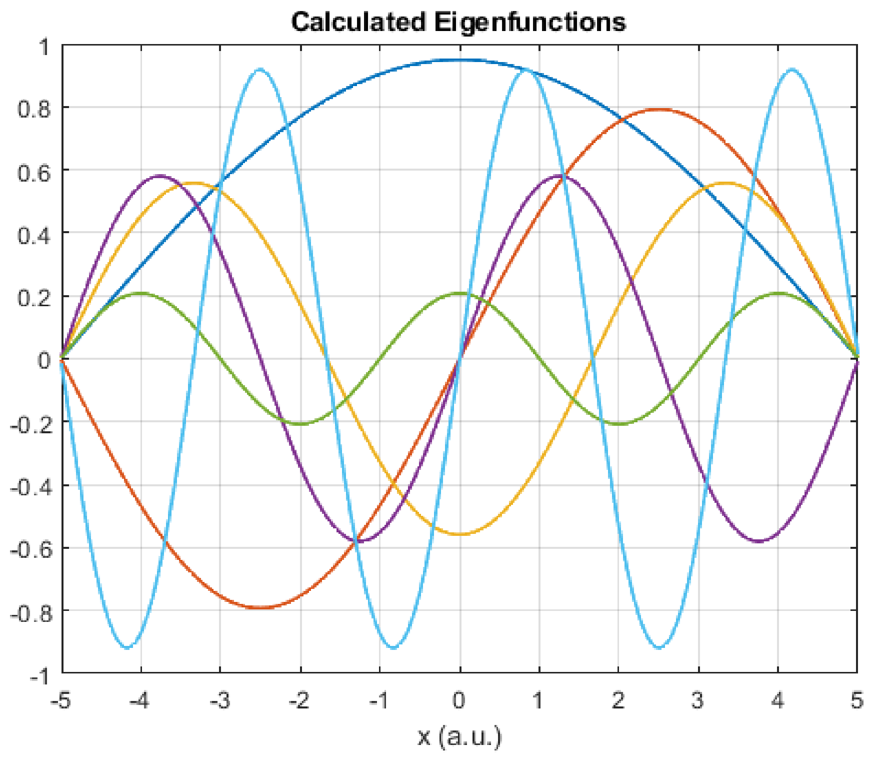
$$\lambda_n = \mathrm{e}^{-jE_n\Delta t}$$

And the eigenvectors, **v**, are going to be the eigenfunctions, or stationary states, of the given potential. Plugging all of this into MATLAB—using sparse matrices and the `eigs` function—we get the following results for the energy levels for the particle-in-a-box system, when *L* = 10, and *n* = 1 to 6:

Energy Levels (*E_n*) Calculated from Eigenvalues

Calculated	**Theoretical**	**Error (%)**
0.0492	0.0493	0.3900
0.1966	0.1974	0.3910
0.4424	0.4441	0.3946
0.7864	0.7896	0.4039
1.2285	1.2337	0.4231
1.7684	1.7765	0.4576

Which is actually pretty good! If we then plot the calculated eigenvectors of this Hamiltonian operator, we get something that looks like:



Which also matches up pretty well with the theoretical result—though they're not properly normalized. What this tells us is that the operator that we've discovered by trying to integrate a one-dimensional PDE is effectively decomposing our initial wave function into its composite (orthogonal) eigenfunctions, and then rotating each one of those by its energy (i.e. eigenvalue). Or rather, to take a cue from fiber optics, figuring out what fraction of our initial function is represented by each eigenfunction (read: via overlap integrals), and then rotating them properly.

What *this* tells us, is that we're probably doing something right. That even though we've discretized things and put them into the computer, we're actually doing proper quantum work, akin to matrix mechanics, although purely in the time-domain (as opposed to momentum or k-space or whatever). If you have potentials that change in time, or non-trivial source terms, then this analysis quickly falls apart, but you can still probably say that you're doing quantum stuff. Even better, you can use these same methods to gain further insight into non-trivial systems without analytic solutions. That's pretty cool!

The full MATLAB code is given below.

MATLAB Code:

```
% setup the workspace:
n = 1024; xmin = -5; xmax = 5; dt = 0.05;
x = linspace(xmin, xmax, n).';
dx = x(2) - x(1);

% define potential (infinite well):
V = zeros(size(x));

% calculate Hamiltonian matrix:
a = -1i*dt/4/dx^2;
H = [a*ones(n, 1), 1-2*a+1i*dt*V, a*ones(n, 1)];
H = spdiags(H, -1:1, n, n);
H = H \ conj(H);

% find the first few eigenvalues / vectors:
[v, d] = eigs(H);

% Put together the results and pretty them up:
[ev_calc, idx] = sort(diag(abs(atan2(imag(d), real(d))/dt)));
ev_theory = (1:6).^2.' * pi^2/2/(xmax - xmin)^2;
error = abs(ev_theory - ev_calc)./ev_theory * 100;

results = table(ev_theory, ev_calc);
results.Properties.VariableNames = {'Theory', 'Calculated'};

efuncs = real(v(:, idx));
efuncs = efuncs / max(max(efuncs))* 0.95;

plot(x, efuncs, 'LineWidth', 1.5);
grid on; xlabel('x (a.u.)');
title('Calculated Eigenvectors');
```