2010 Questions of the Month

The January Question of the Month

Here is another imaginary conversation between you (as a head of a group) and your employee Roshka, in the computational engineering company BATALA (Benchmarks And Theoretical Analysis for Low-tech Applications).

Roshka: Hi, boss. First of all, is it true that you recommended to the general manager to transfer me to the BATALA branch in Australia?

You: Roshka, calm down. Yes, I did suggest this. I think you will contribute more being there.

Roshka: But it's so far away!

You: That's the whole... Ahh, yes, it's a bit far away, but the company will allow you to visit home every month or so. Now, let's get back to business. Did you solve the vibration problem I gave you and found the eigen-frequencies?

Roshka: Listen, I have great news for you. After hearing them, you will reconsider sending me to Australia... The code we have to solve eigenvalue problems is based on iterations and gives approximate eigen-frequencies. But we don't need it. I developed myself an algorithm that finds all the eigen-frequencies for any stiffness and mass matrices _exactly_ without need for iterations! It finds the exact values in a finite number of steps!

You: Roshka, this is impossible. Please use the code we have, and give me the results before packing your stuff for Australia.

Why did you claim that Roshka's algorithm is totally impossible?

Answer

Any algorithm for finding the eigenvalues of a matrix (except for very small matrices, namely 4x4 or smaller) MUST be iterative. Here is an explanation of this.

Let's start with a few known facts on the algebraic eigenvalue problem. The eigenvalues of a square NxN matrix A are the scalar values `s' which satisfy A v = s v for some vectors v (the eigenvectors). As we all know from a basic algebra course, there are non-trivial solutions of this equation if and only if det[A-sI] = 0 where 'det' means the determinant, and I is the unit matrix. "Developing" the determinant gives an Nth-degree

polynomial in s (called the characteristic polynomial of A). The eigenvalues are thus the N roots of this polynomial.

Now, if it was possible to find the exact eigenvalues of A in a finite number of steps, this would mean that it is possible to find the roots of an Nth-degree polynomial in a finite number of steps. In other words, it would be possible to write down a formula for these roots. However, it is well-known that this is impossible! In fact, to quote ET, "from Galois theory one cannot find (a formula for the) exact roots of a polynomial of degree five or more". We are all familiar with the formula for the roots of a quadratic polynomial, and the fact is that such a formula (albeit more complicated) exists for 3rd- and 4th-degree polynomials, but beyond 4th-degree no such formula exists and it will never exist, even if Roshka stands on his head.

Correct solutions were received from (in alphabetical order):

Amiel Herszage, Eli Turkel, Asher Yahalom, Zvi Zaphir.

Comments to the January Question of the Month

The January Question of the Month dealt with the question: Is it possible to find a non-iterative algorithm for finding the exact eigenvalues associated with a matrix eigenvalue problem coming from FEM? The answer was negative, and the argument given was the following. Finding eigenvalues of a matrix is equivalent to finding the roots of the characteristic polynomial, and it is well known (by Galois theory) that it is not possible to find a general closed-form formula for roots of polynomials beyond degree 4.

One comment which is worth making is that despite the equivalence mentioned above, algorithms for finding eigenvalues are rarely based on finding roots of a high-degree polynomial. There are much better ways to find eigenvalues, for example by methods based on so-called Krylov subspaces.

Two comments were made by Achi Brandt: a minor one and a major one.

AB's minor comment is that Galois theory only says that there are high-degree polynomials whose roots cannot be found by a formula (involving rational numbers, the 4 basic operations and the kth root); however, this does not contradict the fact that there are high-degree polynomials whose roots _can_ be found via a formula. So maybe the FE formulation used by Roshka led into such a matrix problem associated with a characteristic polynomial of the latter type... (DG: This is highly improbable, but I admit the "logical crack" in my argument.)

AB's major comment is very interesting and (at least for me) surprising,

since it is opposed to the view one can find in classical numerical linear algebra books like those of Strang and of Trefethen and Bau (from which I borrowed my Question in the first place). According to AB, an eigenvalue solver must be iterative simply because many basic operations involved in it are inherently iterative, and this has nothing to do with Galois theory but with the fact that in practice we always work with finite precision. AB also does not think that the distinction between round-off errors and other types of errors, like those generated by iterative algorithms when stopping them after a finite number of iterations, is meaningful. Here is a quote from his message:

"The fact that you can solve a problem with root operations does not give you any numerical advantage. In solving fourth order polynomials (via a "formula" - DG) you extract roots several time. Each time you do it, you use Newton-type iterations. For the same amount of work of extracting that one root, you could solve by Newton the original problem. And for comparable amount of work you could solve by Newton higher order polynomials (and many other equations).

Putting it differently, if Roshka could show that his solution just involves a finite number of solving a simple nonlinear equation by Newton iterations, his claim that he does it in a finite number of steps would be reasonable, in fact as accurate as saying that the equation x^2 = 2 is solved in a finite number of steps."

The February Question of the Month

A certain commercial Finite Element (FE) code (no names) gives, when one uses linear triangular elements in 2D or linear tetrahedral elements in 3D (e.g. the simplest, constant strain element in elasticity), a warning message:

"Warning: This element is too stiff and may lead to poor accuracy".

This warning appears only for the linear triangular or tetrahedral (4 face pyramid) elements - neither for quadrilateral (Meruba in Hebrew) or hexahedral ("brick") elements, nor for higher-order triangular or tetrahedral elements.

(1) What does this warning mean, exactly? What is a "stiff element" and what is the consequence of the elements being "too stiff"?

(2) While this warning message is basically "correct", in some sense it is also "incorrect". Explain this.

(3) It is true that in general, brick elements perform better than tetrahedral elements. Why, then, do we see quite often complicated FE

models using meshes of tetrahedral elements?

******
Answer
******

(1)

The standard Finite Element (FE) method, when applied to linear "symmetric" problems like elasticity, results in approximate solutions which are "too stiff". Then one can say that "the element is too stiff". To understand what this means, consider a linear elastic body fixed at part of its boundary and given to applied loads. Suppose you solve the problem using FEs and then calculate the approximate total elastic (strain) energy in the body. Then it can be proved that this approximate elastic energy is always smaller than the true elastic energy. This is one sense in which the FE method is "too stiff"; it "absorbs" too little elastic energy.

A more direct interpretation is obtained if we consider, for example, a single concentrated force P acting at a point on the body. Then it is possible to prove, as a direct consequence of the previous theorem, that the displacement under the force as obtained from the FE solution will be smaller than the true displacement under the force. The FE method is "too stiff" since it leads to deformation which is smaller than the true one. (But be careful: it is _not_ necessarily correct that with general loading, the FE displacement at each point is always smaller than the true one. With general loading, such a statement is true only in an average sense and not pointwise.)

It is known that the solution obtained by the FE method approaches the exact solution when one refines the mesh (namely it converges, or more precisely h-converges). But the property discussed above asserts that the FE solution approaches the exact solution "from below", in a certain sense. This information is important, for various reasons that we will not go into, in order to keep this answer brief.

(2)

The warning "This element is too stiff and may lead to poor accuracy" is "correct" in the sense explained above. However, why does the software blame only linear triangular and tetrahedral elements in this flaw? The property of "being too stiff" is common to _all_ the standard elements for the linear elasticity problem! Certainly, bilinear quad elements and trilinear brick elements are also "too stiff". It is true that quad and brick elements are known to be more accurate than triangular and tetra elements, but not significantly so! All these elements have the same _order_ of accuracy. So it seems a bit arbitrary to warn the user against using linear triangles and tetra elements. There is nothing wrong in using such elements. Sure, they are the most basic elements, and one usually

needs many of them to get excellent accuracy. But warning against them is like advising to someone: "Never buy the most basic car/TV/toaster in a sequence of products of a certain company. The 2nd basic car/TV/toaster is always fine, though."

(3)

Despite the fact that, in general, brick elements perform better than tetrahedral elements, we do see quite often complicated FE models using meshes of tetrahedral elements. The reason is that for very complicated 3D geometries it is easier to generate high-quality meshes based on tetras than on bricks. Indeed, most of the general "3D mesh generators" (software whose sole task is to generate a FE mesh for a given 3D geometry) generate tetrahedral meshes.

Answered correctly:
****
Orna Agmon Ben-Yehuda, Zvi Zaphir

****
The March Question of the Month
****

This time the Question does not require any "knowledge", only thinking and some basic algorithmic creativity. Suppose you have a code that involves calculating the Euclidean norm (length) of a long vector. If the dimension (length) of the vector v is N, the formula for the norm of v, denoted ||v||, is

$$\| v \| = \mathrm{SQRT}\{ \mathrm{SUM}\ (v_i)^2 \}$$

or, writing the summation in detail,

$$\| v \| = \mathrm{SQRT}\{ \mathrm{SUM\_i\_from\_1\_to\_N}\ (v_i)^2 \} .$$

Namely, the norm of v is the square-root of the sum of the squares of all the components of v. Any one of us can easily code this formula using a single loop, and thus get a routine that calculates the norm of a vector. Now, suppose that it so happens that

SUM (v_i)^2

is such a large number that it causes overflow in the computer. The square-root of this sum (which is ||v|| that you want to compute) is fine, and does not cause an overflow, but the sum itself is too huge. How would you compute ||v|| and avoid this overflow?

******
Answer
******

The keyword is, of course, scaling. The entries of the vector should be scaled so that the sum of their squares is not so large. One way to do it is by the following algorithm:

* Calculate v_max which is the maximum of all the v_i.

* Compute s = SUM_i (v_i / v_max)^2

* Then ||v|| = v_max * sqrt(s) .

Most of the answers I received suggested this algorithm. OABY suggested to use the midrange rather than the maximum, for smaller round-off errors.

Now, the algorithm above is very simple, but it has the disadvantage of needing two passes through the data: one to calculate v_max and one to do the summation in calculating s. Sometimes one desires to read the data and calculate ||v|| "on the fly", namely without having to store the data or to read it a second time. An algorithm like that has been devised by S.J. Hammarling (see, e.g., the book of Dahlquist and Bjorck, "Numerical Methods in Scientific Computing, Vol. I"). Here it is:

```
t=0, s=1
for i=1:N
  if |v_i|>0
    if |v_i|>t
      s=1+s*(t/v_i)^2
      t=|v_i|
    else
      s=s+(v_i/t)^2
    end
  end
end
||v|| = t*sqrt(s)
```

ZZ proposed (in addition to the v_max algorithm) exactly this algorithm that he had developed himself a long time ago, when he had the same difficulty as the one discussed in the Question.

This algorithm has the disadvantage that it is not amenable to vectorization of parallelization.

RBZ invented a one-pass algorithm, which is different from this one! It is very simple and nice, but it uses trigonometric functions which may be expensive to calculate if N is very large. It goes like this:

```
M = v_1
```

```
for i = 2:N
   a = tan^-1 (v_i / M)
   M = v_i / sin(a)
end
||v|| = M
```

The explanation is this. M denotes here the norm of the partial vector consisting of the first i entries of v that were reached so far in the loop. If we denote this norm more precisely by M_i, it is easy to see that it obeys the recursive relation

$$(M_i)^2 = (v_i)^2 + (M_{i-1})^2$$

This equation can be interpreted geometrically as the Pythagoras theorem for a right-angle triangle. From this interpretation come the relations involving tan(a) and sin(a) appearing in the algorithm above. Very original!

Answered correctly (in alphabetical order):

Orna Agmon Ben-Yehuda, Rami Ben-Zvi, Rachel Gordon, Amiel Herszage, Yoav Ofir, Jonathan Tal, Anne Weill, Asher Yahalom, Zvi Zaphir.

****
The April Question of the Month
****

This time we will ask about the concept "multiobjective optimization".

As everybody knows, the standard mathematical statement of an algebraic optimization problem consists of seeking the minimum (or maximum) of a function - called the objective function, or simply the cost - of a number of "design variables" (the unknown parameters), under some equality or inequality constraints. A central point in this statement is that we have a single objective function. For example, if we want to design an optimal part of an aircraft structure, we can define the objective function to be the weight, and then impose some constraints on the stresses and deformation of the structure, etc.

The term "multiobjective optimization" tells us that the problem involves more than one objective function. Physically this may make sense, since the designer may be interested in minimizing more than one thing: the weight, the cost, the stresses, the deformation, the inverse of the buckling load, etc. But how can we minimize simultaneously more than one objective function? This does not seem to make sense. If we have N objective functions, we can solve N different optimization problems, but this is not what "multiobjective optimization" is about. Also, if we have N objective functions (obj1, obj2, ... objN) to be minimized, we can

define

Obj = max(obj1, obj2, ..., objN)

and then solve a single optimization problem which would minimize the most "critical" cost. But this is again not what "multiobjective optimization" is about.

So what is it about?

******
Answer
******

The definition of "multiobjective optimization" (also called multi-criteria optimization) which is now widely accepted was originally proposed by Vilfredo Pareto. (Look at his picture in http://en.wikipedia.org/wiki/Vilfredo_Pareto ; doesn't he remind you of a certain Jewish visionary?)

In multi-objective optimization we have several objective functions. However, according to the Pareto construction, we do not try to optimize each of them separately, or their maximum, or some weighted linear combination of them. Instead, we try to find ALL the solutions (namely all combinations of design variables) that have the following property:

An improvement in any one of the objective functions (achieved by slightly changing the values of the design variables) leads to degradation in at least one of the other objective functions.

The set of all such solutions (or design values) is called the Pareto set, and the solutions themselves are called Pareto-optimal solutions. The goal in multiobjective optimization is to find this entire set.

After this set is found (by some computational method; there are several and we will not elaborate on this here), it must be represented in some efficient graphical way so that the user/designer can find her hands and legs in this sea of solutions. There are standard ways to represent this set graphically.

Then the user/designer looks at these solutions and makes a decision regarding the chosen design, based on various considerations. The point is that these consideration are mainly heuristic, and are not easy to quantify. This is why this approach is usually better than an approach that would try to find a single optimal solution.

Correct answers were obtained by:

Orna Agmon Ben-Yehuda, Rami Ben-Zvi, Asher Yahalom.

****
The May Question of the Month
****

This time I'll ask you a staggering question. The word "staggering" has two common meanings in the context of computational methods. What are they? An answer on _one_ meaning would be enough to be considered a correct answer.

******
Answer
******

The more common meaning of the two, which all the answers I received pointed to, is related to the concept of "a staggered grid". In such a scheme, to quote ET, "not all the variables are located at the same point, or sometimes the solution at different times are not at the same point (for example, at half points every other time step). Two examples are the Yee scheme (in computational EM) and leapfrog; the latter is staggered because u(t+ delta-t) depends only on u(t-delta*t) and on fluxes at t." An additional example is the staggered finite difference scheme commonly used for the stress-velocity formulation of elastodynamics (for example, in applications of earthquake engineering). Yet one more example is finite volume schemes for fluid flow, where (to quote both RBZ and AY) pressure (or density or enthalpy) is located at the cell center, while velocity (or momentum) components are located at the middle of the cell faces.

A second, slightly less common, meaning of staggering has to do with weak coupling of fields and equations. To illustrate this idea, suppose we have a system of differential equations involving two different types of unknown fields, say u and e. (We can think, for example, of piezo-elasticity, where u is the displacement field and e is the electric field.) Suppose the equations have the form

A[u] + a e = F1 (1)

B[e] + b u = F2 (2)

where A[u] and B[e] are differential terms, and "a e" and "b u" are algebraic (or lower-order differential) terms. Equations (1) and (2) are fully coupled, because both u and e appear in both equations. (Of course, some appropriate boundary and initial conditions are given.) One approach is to solve (1) and (2) simultaneously, namely attack the whole system as one entity. Another option, taking advantage of the "weak coupling" between u and e, is to separate the two sets of equations and solve each one separately, in an iterative manner. For example, we can make an initial guess for e, and solve (1) for u. Based on this u, we can solve (2) for e. Based on this e, we can solve (1) again for u. And so on. If

this procedure converges, it gives us the solution for u and e which satisfies (1) and (2) approximately. This procedure is sometimes called staggering or staggered coupling.

As AH points out, staggering (in this sense) is used quite a lot in fluid-structure interaction problems, where the equations in the solid and the equations in the fluid are decoupled in this fashion. For example, see the paper "Artificial added mass instabilities in sequential staggered coupling of nonlinear structures and incompressible viscous flows" by Förster, Wall and Ramm, in the journal Computer Methods in Applied Mechanics and Engineering, Volume 196, Pages 1278-1293, 2007.

Correct answers were received from (alphabetically):

Orna Agmon Ben-Yehuda, Rami Ben-Zvi, Amiel Herszage, Eli Turkel, Asher Yahalom.

****
The June Question of the Month
****

Imagine the following dialog between A and B, two very smart engineers working in a company doing computations. While A tends more toward practical work, B is more of a theoretician.

A: You won't believe it, but last week I invented my own time-stepping method!

B: Really? Very nice! Did you analyze its accuracy and stability properties?

A: No, I just had a hunch that it's a good one, so I went ahead and implemented it. I ran a test problem with it, and it was stable with reasonable time-step sizes, and very accurate.

B: Can I look at the algorithm? I'd like to try and analyze it theoretically.

A: Sure. [Gives her the algorithm.]

[Two hours later.]

B: Listen, I analyzed the algorithm and it is unconditionally unstable! This means that no matter how small you take the time-step to be, the scheme is always unstable!

A: Really? But how can that be? I am telling you that I applied it to a test problem and it was stable with the time-steps that I took.

B: Hmmm, let me think.

What can be the explanation to this? Let's exclude the possibility that A had a bug in her code, or that B had a mistake in her analysis.

******
Answer
******

First, what does in/stability of a numerical scheme mean? Generally speaking, a stable numerical scheme is a scheme that has the property that a small change in the problem's "data" (loading, sources, boundary conditions, initial conditions) implies a small change in the solution. If there exists a data set for which a small change in the data causes a large change in the solution, the scheme is said to be unstable. (Of course, "small" and "large" must be made precise mathematically, but let's skip the mathematical details.)

In time-stepping schemes, instability typically manifests itself in that the numerical solution "blows up", namely starts to grow in time without limit. This of course causes a large error and does not lead to convergence. (Stability is a necessary condition for convergence.)

Let's take for simplicity a time-stepping scheme for a linear problem. Let's assume further that there is no loading or sources and the boundary conditions are zero. In this case the only "data" are the initial values prescribed by the initial conditions. But since the scheme advances to the next time-step based on information from the current and maybe some previous time-steps, we can actually think of the numerical solution at any time-step, or a number of consecutive time-steps, as "data" for all the time-steps that come after them. Taking this view, the "data" depend not only on the initial data but also on the computational parameters and on the time-step size (and mesh density in space).

We also have to take into account that from one time-step to the next some round-off error (due to the computer operating with a finite number of digits) is introduced, which changes, more or less randomly, the "data" at any particular time step.

In the case described in the Question, the algorithm is unconditionally unstable. This means that for any time-step size, there exists a data set that causes instability.

Now, what can be the reason that B did not see the instability in the results of her test problem? Here are two possible explanations:

1. Sometimes the instability raises its ugly head only after a very long time. This is called long-time instability. For example, suppose the numerical solution behaves like

$u(t) = g(t) * \exp(0.000001\ t)$,

where g(t) is a well-behaved bounded function which is a good approximation to the exact solution. Suppose the "physical time" t has a characteristic scale of 1. Then the exponential term will cause the numerical solution to grow without limit, but this growth will be practically apparent only after a very long time (t of the order of a million). If B did not run her code for very long times, she would not see the instability.

2. Instability means that there exists a "bad data set", namely a data set for which a small change in the data causes a large change in the solution. Because of the randomness of round-off errors (and some may add - because of Murphy's law), an unstable scheme will usually exhibit the instability in practice even if the prescribed initial conditions do not belong to the "bad data set". One has no control over round-off errors, and they come in all shapes and colors, and so there is a high chance that they cause the "data" at any particular time-step to be of the "bad" kind. However, in some special cases it may happen that a "bad" data set does not introduce itself at all during the solution process, and so the instability is asleep and does not manifest itself.

Indeed, there are some theoretically-unstable schemes that are known to be "selective" in the way the instability manifests itself in them. ET has had an experience with such a scheme. In his words: "This was an ADI scheme in 3D which was used in practice for the Euler equations but is unstable. The scheme is only mildly unstable and the instability doesn't show up for moderately dense grids... We used to call this a 'CRAY instability' because it would show up only on sufficiently fine grids that could only be run on a cray supercomputer in the old days." RH gave a nice algebraic analogue of this situation (related to FE matrices and ill-conditioning) that I will omit for brevity.

These are probably the two simplest explanations to the fact that B did not see any instability in her test run, and most readers pointed to them. Additional explanations were proposed by a number of readers:

3. Nothing was said about the specific problem being solved, so there is no reason to assume that it was linear. If the problem is nonlinear, then various interesting numerical phenomena related to stability may occur. It is very hard to analyze stability in the nonlinear regime, so what is typically done is to analyze the stability for the linearized problem. Maybe B ran a nonlinear problem and A analyzed the linear problem. And maybe the nonlinearity has a stabilizing effect on the algorithm (at least for the parameters that B used in her test). ZZ gave an example for this situation in the context of nonlinear behavior of structures, where the ADAMS code was unstable for small steps but apparently stable for larger steps.

4. If the exact solution itself is "unstable", namely if it grows in time

without bound (which may occur with some special nonlinear problems, for example some nonlinear wave equations), then it may be hard to distinguish the "physical" instability from the "numerical" one. AH described such a situation.

Correct solutions were obtained from:

Orna Agmon Ben-Yehuda, Rafi Haftka, Amiel Herszage, Eli Turkel, Asher Yahalom, Zvi Zaphir.

****
Comment on the Answer to the June Question of the Month
****

In my answer I wrote: "In time-stepping schemes, instability typically manifests itself in that the numerical solution 'blows up', namely starts to grow in time without limit." Eli Turkel comments that this is not a precise description. With a stable time-stepping method, the solution remains bounded (or doesn't grow too fast) as the time-step is refined. It is possible that for long times the solution blows up, while it is stable and converges as the time step gets smaller.

****
The July Question of the Month
****

This month's question is courtesy of Micha Wolfshtein.

Sometimes we hear someone saying (or we say ourselves): "After many trials and runs I finally received a converged solution". What is convergence and what is the meaning of the sentence quoted above? (Please note that there are two questions to answer here.)

******
Answer
******

For simplicity, let's consider a numerical method which has a single computational parameter h that controls the fineness of the discretization. Examples for h are the grid spacing in a finite difference method, or the element size in a finite element method, or the time-step size in a time-stepping scheme. As h is reduced, the discretization becomes finer.

Now, in the language of numerical analysis, the method is said to be convergent if, in the limit when h goes to zero, the computed solution approaches the exact solution of the problem. (The approach should be defined in some norm, but let's not discuss this.) In other words, the method is said to be convergent if we can get close to the exact solution

as much as we like by taking a sufficiently fine discretization. In yet other words, the method is said to be convergent if we can make the error (defined as the difference between the computed solution and the exact solution) as small as we wish by making the discretization fine enough.

Is this mathematical notion of convergence exactly what is meant by the sentence "After many trials and runs I finally received a converged solution"? Of course not. There are two major reasons why the word "converged" in this sentence cannot have the meaning of convergence as defined above. The first reason is that in most practical cases the exact solution is not known (otherwise why do we want to solve the problem numerically at all?), so there is no precise way to check if the computed solution indeed approaches the exact solution. The second reason is that the mathematical definition of convergence talks about approach to the exact solution in the limit when h goes to zero. In practice, we can never reach this limit. In actual computations we always work with a positive h, and even if we feel that it is very small, it is still infinitely larger than the limit of h=0.

Concerning the last point, one might suggest that we could relax the definition of convergence when dealing with an actual computer, and say that we call the solution "converged" if the error is reduced to the level of the machine precision error. But in most cases this is extremely difficult or impossible to perform because of memory and/or CPU time constraints. (Not to mention round off errors that may kill us when we are near machine precision.) Besides, the fact remains that the exact solution, hence the error, is not known, so we cannot check how large it is.

So what is meant by "After many trials and runs I finally received a converged solution"? When we say this we mean that we have made many runs with decreasing values of h, namely with finer and finer discretizations, and finally reached the point where a further decrease of h doesn't change the results significantly. More precisely, the change in the results due to further decrease of h is smaller than the tolerance of accuracy that we are interested in.

The essential point here is that instead of looking at the difference between the computed solution and the exact solution (which we cannot do), we look at the difference between two or mote computed solutions with different h values.

Here is a demo of this. Suppose for example, that we make a first run with a certain h (say, a certain finite element mesh) and obtain results. Then we decrease h by 2 (namely we refine the mesh by a factor of 2) and obtain results again. We see that the 2nd significant digit changed in our results due to the refinement. We decrease h by 2 again and run again, and this time only the 3rd significant digit changes in the results. We decrease h by 2 once more and run again, and this time only the 4th significant digit changes in the results. At this point we stop and say:

the solution that we obtained seems to be correct up to the first 3 digits, and that's good enough for us. So our solution "converged"!

Of course, this notion of "practical convergence" is not well-defined mathematically and one should be careful in using it. Extreme examples can be found where, based on such observations, one would conclude that the results are "converged" and in fact the results would be completely wrong. However, the more common case is that the problem and solution and numerical method are all "well behaved", and in such a case using a procedure like this to get correct computational solutions is certainly legitimate as is done all the time in the industry.

Correct answers were received from:
****

Rami Ben-Zvi, Eli Turkel (and thanks Micha Wolfshtein who proposed the Question and appended it with his detailed answer).

****
Comments to Answer of July Question of the Month
****

In the answer to last month's question on the practical notion of a "converged solution", I gave a simple description of how one might obtain such a solution by refining the discretization repeatedly, until the change in numerical results becomes smaller than a certain threshold. (I appended this description with a warning that sometimes this would result in a completely wrong solution.) To this description I received two comments.

Eli Turkel comments that "convergence gets tricky when solving a direct numerical simulation (DNS) of the Navier-Stokes equations. As the mesh is refined there appear finer scales that are not refined. Hence, one cannot take ever finer grids as the physical problem changes with the finer grid."

Micha Wolfshtein comments that a good way to obtain a "converged solution" is by the Richardson extrapolation technique. A slightly edited version of MW's description is as follows. In the Richardson extrapolation technique one makes use of the Taylor series expansion, in powers of the mesh size h, of the discrete representation of the differential equation, to get an estimate for the error. Then one can plot a sequence of the computed values for varying h at a given point. If the rate of convergence (the order) is say p, then by plotting the point value versus h^p one should get a straight line which can be extrapolated to the h=0 line, and the value at the intersection is an estimate for the exact solution. If the truncation error is larger than the round off error then the line will start fluctuating before the straight line is reached. Indeed, even if one gets a straight line at some point there is always a region of smaller h where the line fluctuates. Theoretically this should be done at each

point; practically we do this for some representative points.

MW adds, regarding current industrial application of this technique: "I have seen industrial applications where this technique was used. It is true that for large industrial computations the economy often wins over accuracy, but the result is that practitioners often get results without error bounds, which is a bad practice to my mind."

****
The August Question of the Month
****

What name of a numerical method is associated with an animal? And what is the reason for this name? (I actually found three such numerical methods! But one answer would be sufficient. Let's see if we can set up a small zoo out of the answers.)

******
Answer
******

Here is the list of animal-methods I have collected from all the answers, in alphabetical order:

Ant Colony Optimization method: see, e.g., http://en.wikipedia.org/wiki/Ant_colony_optimization . This is a probabilistic optimization technique which can be reduced to finding good paths through graphs, just like ants look for a good path to travel.

Bee Colony algorithm: see, e.g., http://en.wikipedia.org/wiki/Bee_colony_optimization . This optimization algorithm is inspired by the behavior of a honey bee colony in nectar (Tzuf) collection.

Dogleg Optimization method, and Double Dogleg Optimization method : see, e.g., http://www.sfu.ca/sasdoc/sashtml/stat/chap46/sect25.htm . "The double dogleg optimization method combines the ideas of the quasi-Newton and trust region methods." (I didn't find the reason why it is called like that.)

Hare and Hounds Method: see, e.g., http://www.astro.up.pt/corot/welcome/meetings/m3/ESTA_Nice_Monteiro_2.pdf . This seems to be a general strategy for designing algorithms for constrained optimization. "The best models are produced by the hounds to reproduce the model constraints indicated by the hare(s)." [By the way, if you search Hare and Hounds on the internet you find some good pubs!]

LeapFrog: see, e.g., http://en.wikipedia.org/wiki/Leapfrog_integration . This is a method for solving differential equations. It is called Leapfrog

(why not frogleap?) because the discretization stencil involves backward and forward points but "jumps" over the center point.

Lion in the Desert method: see, e.g., http://bjornsmaths.blogspot.com/2005/11/how-to-catch-lion-in-sahara-desert.html and http://c2.com/cgi/wiki?BinarySearch . This is an algorithm to find the position of an item in a sorted array. The nickname is due to the riddle: How does a mathematician catch a lion in the desert? The answer is: (a) She cuts the desert into two equal halves with a lion-proof fence, (b) she picks the half which has the lion in it, and (c) she catches the lion in that half of the desert, using this algorithm recursively.

Zebra method: see, e.g., http://www.complexity.org.au/ci_louise/vol03/altas2/node3.html . Quoting ET: "This is a line Gauss-Seidel method with different lines having different colors for the order of the iterations, hence the name zebra. Note: This is pronounced differently in American and British English (with Hebrew following the British pronunciation)."

Anne Weill comments that any solver of the Poisson equation should also go into the list... (If you don't understand this, ask a French friend.)

Correct answers were received from:

Rami Ben-Zvi, Amiel Herszage, Eli Turkel, Anne Weill, Asher Yahalom.

****
The September Question of the Month
****

Once again a Roshka episode! Imagine that you are the head of a team in the computational engineering company BATALA (Benchmarks And Theoretical Analysis for Low-tech Applications). Your former employee Roshka has been sent to work at the BATALA branch in Australia. Now he suddenly knocks on your door.

You: Roshka!? What the ... Are you on vacation?

Roshka: Yes, for the holidays. And I am dying to tell you how great I have been doing in Australia.

You: Please tell me.

Roshka: I started to look into solving large deformation problems in nonlinear elasticity using finite elements. I am now an expert in using good techniques and meshes. There are only two approaches in using a mesh for such problems: either the mesh is fixed, or it follows the deformation and moves with it. If the mesh is fixed this is called an Eulerian

approach, and if the mesh follows the deformation it's called Updated Lagrangian. For some reason, they also call the Eulerian approach Total Lagrangian. Now, Updated Lagrangian is not so good because with large deformation the elements can become very distorted, and this may ruin the entire computation. On the other hand, Total Lagrangian is good because the elements always look nice and they never get distorted. Hey, why do you block your ears?

You: It's just hard for me to hear such non... such inaccurate description. Roshka, you said at least three wrong things. Do you want me to explain to you?

Roshka: Yes, but first let me tell you about the other great piece of news that I have. They decided in Australia that I am returning to your team in December! Hey, boss, do you feel well? Miri, quick, some water! The boss fainted!

Now the question is: What is wrong in Roshka's description? Even one mistake would count as a correct answer.

******
Answer
******

Here are three wrong statements made by Roshka:

1. "There are only two approaches in using a mesh for such problems: either the mesh is fixed, or it follows the deformation and moves with it." This is not true. There is a class of methods called Arbitrary Lagrangian Eulerian (ALE) in which the mesh motion can be controlled and can be set to be as desired, with or without dependence on the mechanical behavior. See, e.g., http://www.me.sc.edu/research/jzuo/Contents/ALE/ALE_1.htm .

2. "For some reason, they also call the Eulerian approach Total Lagrangian." This is not true; Total Lagrangian is a Lagrangian method, namely the analysis follows material particles, and is not the same as the Eulerian approach, in which the analysis follows the solution at fixed points in space. In both approaches the mesh used is fixed (see the next point), but they are definitely two different approaches.

3. "Now, Updated Lagrangian is not so good because with large deformation the elements can become very distorted, and this may ruin the entire computation. On the other hand, Total Lagrangian is good because the elements always look nice and they never get distorted." The difference between Updated Lagrangian (UL) and Total Lagrangian (TL) is only in the domain of integration. In UL the integration is done directly on the deformed configuration at each stage of the deformation. In TL the integrals are first transformed into the undeformed (reference)

configuration, and the integration is done there, which means that the mesh looks fixed. However, in essence there is no difference between TL and UL; they are equivalent, and (up to round-off errors) yield exactly the same numerical results. Therefore saying that TL is better than UL is incorrect. It's true that in UL one can "see" how the mesh becomes distorted, while in TL one does not see that, but the calculation deteriorates just the same, since the Jacobian of the transformation becomes pathological. In fact, in a sense, UL is "better" than TL because one can actually see better where the trouble is!

Correct answers were obtained from:
****

Orna Agmon Ben-Yehuda, Amiel Herszage.

****
The October Question of the Month
****

The following situation is typical. We construct a model and run it with a certain grid/mesh, and get results that turn out not to be accurate enough. Then we realize that our grid was too coarse, and we "correct" the situation by using a finer grid.

Now, there is a known method in which a coarse grid is used to "correct" (in some sense) the fine-grid solution. What is this method, and how does this "correction" work?

******
Answer
******

What I had in mind when asking this question was multigrid methods. The main inventor of multigrid was Prof. Achi Brandt from the Weizmann Institute, which we are very proud to have as a member in our community. A very rough description of the basic idea of multigrid is as follows.

Typical iterative algebraic solvers of the linear system Ax=b, like the Gauss-Seidel method, have the property that the highly-oscillatory parts of the error are damped rapidly, whereas the smooth parts of the error decay slowly. So when using such an iterative method, it is the smooth (i.e., slowly varying) modes in the error that are difficult to control, and may require a huge number of iterations to decay. Now, a crucial point that is the basis for multigrid is that what is considered a "smooth" mode or an "oscillatory" mode depends on the grid/mesh density. A function may be claimed to be slowly varying (smooth) if it is "seen" through a fine grid, but it may be regarded as oscillatory when "seen" through a coarse grid! Multigrid methods exploit this fact by making use of a coarse grid, or actually a hierarchy of coarse grids, that cause (in a way that we

shall not go into here) the fast decay of those modes that are slowly-varying in the fine grid. The operation that is associated with this is called "coarse-grid correction" which connects to our Question.

I strongly recommend the book "A Multigrid Tutorial" of W.L. Briggs to anyone who wants to learn about multigrid for the first time. I think it is a wonderful book.

Some readers (RG, RH, ET) mentioned also other methods which make use of a coarse grid to "correct" a numerical solution. One such method is Richardson's Extrapolation. I will not describe it here since a short time ago it was described very nicely by Micha Wolfshtein in his comment on a previous Question of the Month. Yet another class of methods that "corrects" the solution via a coarse grid is called Deferred Corrections; see, e.g., the paper by Rangan in http://www.cims.nyu.edu/~rangan/sdcdae_BIT.pdf .

One reader (ZZ) pointed out that there were pathological cases, usually associated with some kind of singularity, in which the numerical solution behaves _worse_ as the mesh is refined beyond a certain level. In simple words, the solution does not converge as the mesh is refined; yet the solution obtained with a "reasonably" coarse mesh is useful to engineering practice. There are a number of well-known examples of this scenario, like concentrated loads in certain configurations, shear bands when analyzed with standard FE methods, re-entrant corners, etc.

Correct answers were obtained from:

Orna Agmon Ben-Yehuda, Rafi Haftka, Amiel Herszage, Roland Glowinski, Stephane Seror, Eli Turkel, Asher Yahalom, Zvi Zaphir

****
The November Question of the Month
****

In methods like Finite Differences and Finite/Spectral Elements, use is made of some grid/mesh consisting of cells or elements. In the "high-order methods" among them, each cell/element includes many grid-points or nodes inside it and/or on its boundary. Typically these nodes are _not_ equally spaced in the cell. Why not?

******
Answer
******

This has to do with a numerical effect called the Runge phenomenon. Suppose you have a function defined in some finite domain, and you would like to interpolate it using a high degree polynomial with uniformly-spaced interpolation points. Suppose you do this, and you

calculate the interpolation error as you increase the polynomial degree p of the interpolating functions. (A method using a high p is called a high-order method.) What would happen is that the interpolated function will _not_ converge to the exact function as p goes to infinity. Of course the two functions will match at the interpolation points, by construction, but in between the interpolation points the interpolated function will exhibit wild oscillations, which will go larger and wilder as p increases.

In Finite Difference (FD) and Finite/Spectral Element (FE, SE) methods, the situation is somewhat similar. It's true that these methods do not seek the interpolation of a _known_ function. They seek the solution to some differential equations satisfying some extra conditions. Nevertheless, the grid points, or nodes, may be thought of as "interpolation points", albeit to an unknown function. In FE and SE methods this notion is more precise, since the shape functions used are indeed interpolation functions, but even FD methods behave in a similar manner in this respect.

One might think the following: "Ok, the interpolating functions go wild _between_ the interpolation points or nodes. So I will not _look_ between the nodes! I will just look at the solution _at_ the nodes, U'va Letzion Go'el." But this will fail, since we have a stability effect here. Even the slightest perturbation in the data will cause a big change in the solution. So even if we just look at the nodes themselves, as p increases we will get a nonsense solution due to "noise" (e.g., round-off errors) in the data.

So uniformly spacing the nodes is no good when p is large. There are some known non-uniform distributions of the nodes which are stable. Perhaps the most famous one is that of the Chebishev points, which is used in many spectral and SE methods. With such distribution of points, nothing bad happens when you increase p, and even between the nodes the calculated solution approaches the exact one.

Correct answers were obtained from:

Matteo Parsani (communicated by Eli Turkel).

I wonder why I did not receive more than one response to this QotM. Maybe it was too easy, or too difficult, or maybe people were too occupied with the fire on the Carmel (which has been the case with me). I hope for a stronger response to the following QotM.

****
The December Question of the Month
****

This Question is about inverse problems. It is not so easy to define

inverse problems in a simple way, but I think that once you are given some examples, you know how to recognize an inverse problem when you see one. Generally, in an inverse problem the "data" and the "solution" are switched with respect to a "regular" problem (also called forward problem). In an inverse problem, part of what we are used to regard as the "solution" turns into the "data", and part of what we are used to regard as "data" turn into the "solution".

Here is one example. In a "regular" heat-conduction problem we are given the geometry, the material properties and the thermal loading and we seek the temperature distribution. In an inverse problem we may be given the geometry and the loading and a partial knowledge of the temperature field (e.g., the temperature measured in a small part of the domain), and we may want to find the material properties.

Joseph B. Keller wrote in 1976 a famous paper on inverse problems (entitled simply "Inverse Problems"), and in the Introduction, he gave an example to an inverse problem:

What is the question to which the answer is "Chicken Sukiyaki"?

He gave the answer to this question at the end of the paper. The answer is:

What is the name of the sole surviving Kamikaze pilot?

(I guess this is not so politically correct. But in 1976 it was considered ok.)

Now we come to our question. The question is: Why are inverse problems much more difficult to solve numerically than "regular" problems?

There is no intention to go into too much detail here, or to plunge into specific examples. The question is general, and the answer should also be quite general.

******
Answer
******

The answer to this question has a theoretical aspect and a practical aspect. The theoretical aspect has to do with the fact that inverse problems are usually ill-posed. To understand what this means, let's first define what a well-posed problem is. According to the definition of the French mathematician Hadamard, a problem is called well-posed if it satisfies all of the following criteria:

1. Existence: The problem has a solution.

2. Uniqueness: The problem has no more than one solution.

3. Stability: A small change in the "data" gives rise to a small change in the solution.

Now, in inverse problems it is typically the case that one or more of these criteria is not satisfied. To discuss this, let's consider the heat conduction problem given as an example in the body of the Question: we are given the geometry and the thermal loading and some measurements of the temperature, say on part of the boundary, and we wish to find the material properties.

Let's start from Existence (criterion 1): we are not guaranteed that for the given data (the measured temperatures) a solution to this problem exists at all! You may wonder: "How can this be possible? After all, the mathematical model is supposed to represent the reality, and assuming that it indeed represents the reality well, how can it be that there is no solution when in reality we did measure those temperatures that are given as data? Surely the reality has a 'solution'..." The main answer is that the data that we measure are never 'exact' because of measurement noise which always exists. This measurement noise can change the problem from having a solution to not having a solution.

Second, there is Uniqueness (criterion 2): we are not guaranteed that for the given data the solution (if it exists) is unique. There may be two (or more) distributions of material properties that would give the same measurements. Whether the solution is unique or not depends strongly on the amount of available information (measured data). It is easy to imagine that if all we have is, say, the temperature measured at one single point on the boundary, then we can find many distributions of material properties that would result in this measurement.

Third, there is Stability (criterion 3). Many inverse problems are notorious for being unstable, namely that a small change in the data gives rise to a large change in the solution. (Note that this is _not_ numerical stability. We are talking here about the stability of the original problem, before any discretization.)

In summary, inverse problems are usually ill-posed, from one or more of the reasons given above, and this is what makes them difficult to handle by any numerical method. In contrast, the more regular problems that most of us have experience with are well-posed.

So how do we approach inverse problems when they are ill-posed? There are a number of things that we can do, and this brings us to the practical reason of why ill-posed problems are so difficult. First, in many cases an inverse problem is reformulated as an "optimization" problem. Rather than trying to find head-on the material properties that would yield the measured temperatures, we look at the functional way in which the material properties affect the temperatures in the measurement region, and among all possible distributions of material properties we seek the distribution

that minimizes the difference between the resulting temperatures and the measured data. Algorithms for optimization problems are iterative, and in each iteration we have to solve a "forward problem". This makes the solution of inverse problems typically heavy computationally.

The optimization problem, if properly formulated, always has a solution, so at least Existence is guaranteed. If there is enough data and if we are lucky, there is unique solution which is the global minimizer, but even then there are typically many local minima, and our goal is to find the global minimum among them. This calls for use of "soft" optimization methods that are not "gradient-based" (since the latter can find only a local minimum); for example Genetic Algorithms and Neural Networks.

Something else that is usually done, in order to overcome the instability of the inverse problem, is to slightly change the original problem in order to make it stable, or at least much more stable than the original one. This is called "regularization", and there are a number of known regularization techniques. A good regularization makes the problem much easier to solve while still yielding a solution (material properties distribution) which is close to the true one.

On top of all this, as a number of readers commented, there is the additional difficulty that in inverse problems we are often dealing with complicated, large, nonlinear physical systems, so that even the forward problems themselves are difficult to solve.

Correct answers were received from:

Orna Agmon Ben-Yehuda, Rami Ben-Zvi, Ilan Degani, Rafi Haftka, Eli Turkel, Asher Yahalom.