NTNU – Trondheim
Norwegian University of
Science and Technology

TDT 24 - Iterative methods

1. Some words about linear algebra
2. The Jacobi method
3. The Gauss-Seidel method
4. Successive Over-Relaxation (SOR)

The world according to Ole

- Ole Saastad (at UiO) likes to say that

  "supercomputers only really do two things:
  linear algebra and Fourier transforms"

- This is not 100% true, but it's close enough
- The diversity comes from the enormous variety of things we can represent in those two ways

Solving Ax = b

- A is a matrix
- b is a vector full of numbers we know
- x is a vector full of unknown numbers
- That is the only problem linear algebra cares about

So, x = $A^{-1}b$, right?

- Well, yes… in principle
- There are some issues:
  - Explicitly finding A^{-1} is really slow when A is huge
  - Even with a systematic method, it’s easy to lose precision on a computer
- It gets even worse:
  - Not every A even has an A^{-1} in the first place

Why do we care about huge matrices?

- Most things in this world can be represented by complicated, curvy, continuous functions
  - Those can lead to really intricate expressions
  - Really intricate expressions can be difficult to differentiate, integrate, and otherwise analyze
- If we approximate the curves with lots and lots of short, straight lines instead, we get almost the same thing
  - Those are extremely simple to express, differentiate, integrate, *etc.*
  - We need lots and lots of them in order to stay close to curves
  - The curvier our curve is, the more lines we'll need
- When A is an NxN matrix, and b,x have N elements each, Ax = b represents the relationship between N straight lines

It’s not just about physics

- Graphs can be encoded as matrices
  - If you look at the connectivity graph of web pages, it can be interpreted as a gigantic matrix
  - If you look at the layers of an artificial neural network, they can be interpreted as a string of semi-large matrices
  - The development of stock prices over time can be interpreted… oh, you get the point.
- When you figure out how to make linear algebra out of something, you can “just” throw more computing power at it to become rich and famous

The tricky part

- In order to solve Ax = b *efficiently* and *accurately*, you need to choose a method that works
- The choice of method doesn't just depend on the size of A, but also the values of the numbers that go into it
- We haven't fully automated this classification of matrix types yet, so:
  - You have to know what *your* numbers represent
  - This enables you to reason about how to manipulate them

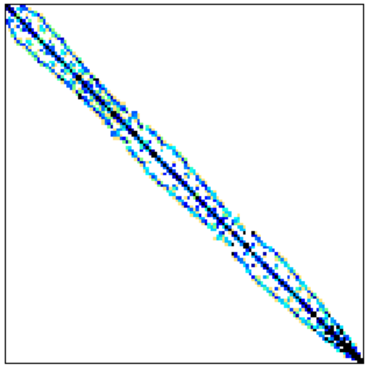
Today’s menu

- We’ll cover three (closely related) solutions today
  - Jacobi iterations
  - Gauss/Seidel
  - SOR

  and think for a moment about the parallelism, so we can trade bigger computers for faster results

An example: FIDAP/ex3

(from the SuiteSparse matrix collection)



- Regrettably, we can’t create example data by just pulling a ton of random numbers out of a hat
  - They probably won’t correspond to a set of lines that fit together
- The SuiteSparse matrix collection is an online repository of various matrices that are derived from practical problems
- We can grab one from there, and make sure it has the properties we need today

NTNU – Trondheim
Norwegian University of
Science and Technology

Tales from the code archive

download.sh

- Downloads our matrix in a simple text format and extracts it
- You can look inside ‘ex3.mtx’ and probably figure out what the meta-data represents
- I’m not going to add complexity by parsing the whole format properly, we’ll just steal the values for this one example and hardcode everything

convert_full_matrix.c

- The *.mtx file is a compressed representation
- Only coordinate pairs for i>=j are listed, symmetry implies the rest
- This program fills it out as a complete NxN array of numbers, multiplies the entries where i=j by 10 (more on that in a minute)
- Finally, it saves the matrix as binary data in ‘ex3.dat’
- Don’t do this to arbitrary *.mtx files without checking how big the matrices actually are first

We’ve got our A

- Where’s the b vector?
  - There isn’t one
  - Never mind that, we can make one
- If we just add all the values in the rows of A, that’s the same thing as multiplying it by a vector full of 1-s
  - row_sums.c in the code archive does that, and stores the result in ‘b.dat’
  - Now we have our linear system, and can write solvers that produce matching ‘x.dat’ files
  - If the solvers work correctly, we can calculate Ax afterwards and see that it actually produces b
  - ...or just use the fact that we already know the correct x, because this is a constructed example

Jacobi iterations

- One way to look at Ax = b is to say that each row in A corresponds to a (multivariate) linear equation:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 10 \\ 11 \\ 12 \end{bmatrix}$$

is the same thing as

$$\begin{bmatrix} 1x_1 + 2x_2 + 3x_3 = 10 \\ 4x_1 + 5x_2 + 6x_3 = 11 \\ 7x_1 + 8x_2 + 9x_3 = 12 \end{bmatrix}$$

Solving one equation

- Equation #2 is a bit special for x_2

$$4x_1 + 5x_2 + 6x_3 = 11$$

so we can solve it for x_2 and get

$$x_2 = \frac{11 - 4x_1 - 6x_3}{5}$$

- If we do the same thing for every row, a pattern emerges:

$$x_i = \frac{b_i - \sum_{j=0}^{i-1} A_{ij}x_j - \sum_{j=i+1}^{N-1} A_{ij}x_j}{A_{ii}}$$

“b minus the off-diagonal Ax, divided by the diagonal”

Shortening it a little

- If we just remember that “sigma” for row i stands for the product of the off-diagonal A elements and x, it’s easier (and common) to write

$$x_i = \frac{b_i - \sigma_i}{A_{ii}}$$

NTNU – Trondheim
Norwegian University of Science and Technology

How this works

- If we know the x vector that solves Ax=b and put all the numbers in, all the equations we just suggested will be perfectly balanced
- If we just guess some random x and hope for the best, they probably won't
- *However:*
  - *If we only make our sigma out of the guessed values and take the result as a new guess for x_i, it might be closer to the solution than our previous guess*
  - Under a certain condition, it provably is
  - I'm not proving it, they do that in math class

NTNU – Trondheim
Norwegian University of
Science and Technology

The certain condition

- If the diagonal element A_{ii} in row i is bigger than the (absolute) sum of all its neighbors, this method improves each guess a little bit
- When this is the case in every row, we say that the A matrix is *strictly diagonally dominant*
- ex3 isn't strictly diagonally dominant the way it's provided from the web
- This is why we multiplied the diagonal by 10
  - Probably ruining the fluid mechanics problem we got the matrix from
  - Still, we got a linear system that we can solve with Jacobi iterations

The Jacobi iteration method

- Guess some random x vector, and call it x^{old} or something
- Do this to every row in order to get new x-values

Output: new x-values $\longrightarrow$

$$x_i = \frac{b_i - \sum_{j=0}^{i-1} A_{ij} x_j - \sum_{j=i+1}^{N-1} A_{ij} x_j}{A_{ii}}$$

$\longleftarrow$ *Input: old x-values*

Check if the new x values are close enough to the old that we can say they've stopped changing

  – Or, if you prefer, close enough to a solution that we can call them one. That requires you to calculate a little bit more, though

- Do it again if they are not (yet) close enough

In code

- The program ‘jacobi.c’ does what we just described
  - At least if you prepared ex3.dat and b.dat before running it
- The solution comes out in ‘x.dat’
  - We expect it to be a vector of all 1-s, since that’s how we made b
  - ‘plot_solution.gp’ draws a picture of ‘x.dat’ so we can see it’s all 1-s
  - If you look super-closely at the values, they’re a *little* bit off, but we can use them still (or increase precision in exchange for more iterations)

Gauss-Seidel iterations

- This is exactly the same thing, except for one observation:
  - When we're working somewhere in the middle of the rows, we've already produced improved guesses for a bunch of our x-s
  - Why not use those improved estimates in the sigma-part immediately, instead of postponing them for the next iteration?
  - It can't be any worse than waiting
- What happens in the code:
  - All we really need to do, is stop using separate vectors for new and old values
  - Just put the updates right into one single x-vector right away
- See implementation in 'gauss-seidel.c'

Gauss-Seidel advantages

- (Slightly) lower memory requirement, since we only use one vector where there were two
- Finds the answer almost twice as fast for this example
- What’s not to like?

Gauss-Seidel disadvantage

- It’s not as easy to parallelize
  - Half the execution time is good and all that, but why don’t we just get ~1/4 the time by running Jacobi iterations on 4 cores instead?
  - The Jacobi iteration is perfectly parallelizable, all the reading is from old values and all the writing is in new values
  - The Gauss-Seidel iteration has a dependency where you can’t update x_5 before you’re finished with $x_{0,1,2,3,4}$…
- There are algorithms for this
  - Once you have x_0, you can add it to all the other sigmas in parallel
  - This *wavefront* type of method isn’t as effective as the full-on parallelism of Jacobi
  - It also isn’t as easy to program correctly

SOR

- This is a variant of Gauss-Seidel in which the update is turned into a weighted sum of the old and new values:

$$x_i^{new} = (1-\omega)x_i^{old} + \omega\frac{b_i - \sigma_i}{A_{ii}}$$

- I didn't write code for this, but it's not hard
  - you can try it at home
- Pick omega-values between 0 and 2
  - The best one will depend on your A and your b, so a reasonably good one is usually found by testing
  - 1.5 is not a bad place to start, usually
- Advantages/disadvantages are exactly as for Gauss-Seidel

Alternative parallelization of G-S and SOR

- Lots of discretizations create matrices with regularly spaced, diagonal lines
  - We get those from approximating areas and volumes using squares and cubes
- Those contain a pattern where you can update many rows in parallel as long as they don’t require each others’ values in the sigma part of the update
  - Typically, you can do one half in parallel first and the other half in parallel afterwards
  - That is called Red/Black ordering
  - It doesn’t work trivially on ex3, but I can fix you an example if you ask

What I hope you can take away from all this

- In math classes, these methods usually appear in the form of equations that we memorize to pass an exam
- When you face an unsolved, real-life problem, it can be fantastically productive to think

  “Hmmm… can I turn this into a matrix and a pair of vectors?”
- If you can, then it’s “easy” to super-size your solution
- This mode of thinking gets easier with practice
  - It can be hard to come up with examples to practice on
  - Now you have this ex3-example to start from if you want to

For less sloppy notation:

- I just waved my hands and ran some code today
- If you prefer a more thorough and mathematical treatment of the same material, I highly recommend

  Numerical Mathematics and Computing
  by E. Ward Cheney and David Kincaid
- It’s a whole book, so it has chapters on all kinds of other neat stuff as well
  - Jacobi, Gauss-Seidel, and SOR are covered somewhere in the two chapters on linear systems
  - I haven’t checked whether they move around between editions or not.