ANALYZING ALGORITHMS.

David F. Gleich

August 21, 2023

Analyzing algorithms for linear systems, least squares, and eigenvalues.

We have now seen a few different types of algorithms to solve the fundamental matrix computations.

Elimination with and without pivoting for linear systems, Richardson, Steepest Descent, Gauss Seidel, Jacobi, etc.

How should we pick which one to use to solve our problem? There are various ways of thinking about this question. The first one regards what is feasible with your matrix.

1 TIME AND MEMORY REQUIREMENTS

The simplest way to analyze the algorithms is in terms of how much time and memory they require. It turns out memory is usually a more pressing constraint than time. So let's start with that one!

To run the Richardson method or steepest descent method, we require:

1. a way to multiple $\boldsymbol{A}$ by a vector $\mathbf{v}$
2. memory to store two or three vectors of length n. (You saw this on the homework.)

To run the elimination matrix that gave us the LU factorization of a matrix, we require:

1. $O(n^2)$ memory for a general matrix problem because we *change* the matrix after the first step
2. $O(n)$ memory for the changes to the vector $\mathbf{b}$ [1]

[1] These can often be done in place as well.

For a general dense matrix, there is no difference between the memory requirements. However, for a matrix with any type of structure, or especially sparse structure, then it is easier to think about how to exploit that structure to make the Richardson, steepest descent method run with less memory.

Consider a matrix that is Toeplitz. For Richardson, we only need to store $O(n)$ memory to be able to do the matrix-vector products. Whereas for the LU factorization, we will need to build the matrix at $O(n^2)$ memory in order to run the algorithm.

Of course, there is the problem of how long the Richardson method takes, and whether or not it will even converge. (Remember, we only showed that we could guarantee convergence for a symmetric positive definite matrix.)

2 THE FLOP COUNT

One common way of evaluating the work of an algorithm is in terms of the number of floating point operations it does. This measure is classical, but still important, because * floating point operations used to be much more expensive than other processor operations * most systems now have special hardware dedicated to floating point computation such as vector processing units (called AVX on Intel processors) or GPUs. Hence, having a fairly exact count of the number of FLOPS allows us to understand how fast a particular operation may go on a computer.

> **Aside.** There is a lot more to high performance computations, such as cache efficient, blocking, vectorization, than just purely understanding the FLOPS, however. The use of FLOPS counts are largely just to suggest an upper-bound on performance.

Aside again. What is a FLOP and FLOPS? A floating point operation (FLOP). Or a floating point operation per second (FLOPS)? Suffice it to say that these acronyms are used inconsistently and potentially mixed. The context determines if we are counting floating point operations (FLOPs) or measuring floating point operations per second (FLOPS). This is extremely confusing when there is a prefix. What is one gigaflop? One billion floating point operations or one billion floating point operations per second. Again, context will determine if we are counting or timing.

2.1 A WARM UP: THE FLOP COUNT OF MATRIX-MULTIPLY.

Consider the following algorithm for multiplying two matrices.

```
function matmul(A::Matrix, B::Matrix)
  m,k = size(A)
  k,n = size(B)
  C = similar(A, m, n)
  fill!(C, 0)
  for j=1:n
    for i=1:m
      for r=1:k
        C[i,j] += A[i,r]*B[r,j]
      end
    end
  end
  return C
```

We can count the number of FLOPs by looking just at the inner-loop, which is an inner-product. Let's consider a simple 4-element vector

$$C[i,j] = A[i,1] * B[1,j] + A[i,2] * B[1,2] + A[i,3] * B[3,j] + A[i,4] * B[4,j].$$

There are 4 multiplications and 3 additions, for a total of 7 flops. The way our code above works, however, is to use 8 flops because we always add to the existing value:

$$C[i,j] \leftarrow C[i,j] + A[i,1] * B[1,j] + A[i,2] * B[1,2] + A[i,3] * B[3,j] + A[i,4] * B[4,j].$$

Here, we have the convention that $\leftarrow$ is the "assign' ' operation, which is commonly expressed as = inside of programming languages, but has a distinct meaning from the mathematical = operation. For instance, \verb{x = 2*x} is perfectly common computer code, but if used mathematically essentially implies that $x = 0$.

Consequently, there are $2r$ FLOPs in the innermost loop. This is executed mn times, so there are $2rmn$ FLOPs in this computation of a matrix-matrix product.

2.2 THE FLOP COUNT OF LU

Let's consider something more interesting, the pivoted LU decomposition of a matrix. Our code is

```
function myreduce_all_pivot(A::Matrix)
  A = copy(A) # save a copy
  n = size(A,1)
  L = Matrix(1.0I,n,n)
  U = Matrix(1.0I,n,n)
  d = zeros(n)
  p = collect(1:n)
  for i=1:n-1
    maxval = abs(A[i,i])
    newrow = i
    for j=i+1:n
      if abs(A[j,i]) > maxval
        newrow = j
        maxval = abs(A[j,i])
      end
    end
    if maxval < eps(1.0)
```

```
18          error("the system is singular")
19        end
20
21        j = newrow
22        # swap the ith row/column
23        tmp = A[i,:]
24        A[i,:] .= A[j,:]
25        A[j,:] .= tmp
26
27        p[i],p[j] = p[j], p[i]
28        L[i,1:i-1], L[j,1:i-1] = L[j,1:i-1], L[i,1:i-1]
29
30        α = A[i,i]
31        d[i] = α
32        U[i,i+1:end] = A[i,i+1:end]/α
33        L[i+1:end,i] = A[i+1:end,i]/α
34        A[i+1:end,i+1:end] -= A[i+1:end,i]*A[i,i+1:end]'/α
35      end
36      d[n] = A[n,n]
37      return L,U,d,p
38    end
```

In the first block of code (lines 2-7) we do no FLOPs because it's just allocating memory. There are $n-1$ executions of the loop on line 8, and each execution consists of

- Line 12: $(n-i)$ floating point comparisons (these are counted because they may require something like a subtraction), whereas absolute values are not because they can be done with an extremely simple operation.
- Lines 21-28: a swap, that does not involve FLOPs
- Lines 32-33: $2(n-i)$ divisions
- Line 34: $3(n-i)^2$ multiplications, additions, and divisions (one of each for each of $(n-1)^2$ elements.

Next, we have to sum this over all i. So the total FLOP count of this implementation is

$$\sum_{i=1}^{n-1}(n-i)+2(n-i)+3(n-i)^2.$$

There are various rules to compute these sums called a "finite calculus'' that work like a crank equivalent to standard calculus. However, I find it easier just to use Wolfram Alpha, which yields

$$\sum_{i=1}^{n-1}(n-i)+2(n-i)+3(n-i)^2 = n^3-n.$$

Note, however, that we can reduce Line 34 to \$2(n-i)^2 FLOPs if we use the elements of either L or U by avoiding the division. Also, note that if we actually computed the standard LU decomposition where $U[i, i+1:end] = A[i, i+1:end]$ then we could further avoid $n-i$ divisions, yielding a more efficient count of

$$\sum_{i=1}^{n-1}(n-i)+(n-i)+2(n-i)^2 = \frac{2}{3}(n^3-n).$$

Aside. Note that these final sums must be integer values!

3 SENSITIVITY OF INPUT-OUTPUT MAPS

3.1 DERIVATION OF THE HILBERT MATRIX

One of the most interesting matrices and matrix problems arose around the turn of the century when David Hilbert wanted to look at approximating functions by polynomials. This leads to a way to quantify how difficult a linear systems is to solve on the computer, called the condition number.

Given a function $f(x)$ we want to create a polynomial approximation $p(x)$ that minimizes the squared error over the interval $[0, 1]$. Let $p(x) = c_1 + c_2 x + c_3 x^2 + \cdots + c_k x^{k-1}$. Then we want to pick $\mathbf{c}$ such that:

$$E(\mathbf{c}) = \int_0^1 (f(x) - p(x))^2 \, dx$$

is as small as possible. If we simply expand the objective function, then we have:

$$E(\mathbf{c}) = \int_0^1 (f(x) - p(x))^2 \, dx = \int_0^1 (f(x) - \textstyle\sum_{j=1}^k c_j x^{j-1})^2 \, dx$$

$$E(\mathbf{c}) = \sum_{j=1}^k \sum_{i=1}^k c_i c_j \int_0^1 x^{j-1} x^{i-1} \, dx - 2 \sum_{j=1}^k c_j \int_0^1 f(x) x^{j-1} \, dx + \int_0^1 f(x)^2 \, dx.$$

Hence, we have

$$E(\mathbf{c}) = \mathbf{c}^T \boldsymbol{A} \mathbf{c} - 2\mathbf{c}^T \mathbf{b}$$

where

$$A_{ij} = \int_0^1 x^{j-1} x^{i-1} \, dx = 1/(i + j - 1) \qquad b_j = \int_0^1 f(x) x^{j-1} \, dx.$$

This minimizer of this is just the solution $\boldsymbol{A}\mathbf{c} = \mathbf{b}$.

But the matrix $\boldsymbol{A}$ is surprising. It is called the Hilbert matrix (after David Hilbert) and is one of the most highly sensitive linear systems of equations.

3.2 AN INFORMAL ANALYSIS

To understand why, consider what happens if we make a tiny error in evaluating $\mathbf{b}$. The answer we want is the solution $\mathbf{x}$ in $\boldsymbol{A}\mathbf{x} = \mathbf{b}$ but we actually see $\mathbf{b}' = \mathbf{b} + \mathbf{d}$ where $\|\mathbf{d}\|$ is very small. Then, we would compute

$$\boldsymbol{A}\mathbf{y} = \mathbf{b}' \qquad \mathbf{y} = \mathbf{x} + \boldsymbol{A}^{-1}\mathbf{d}.$$

Consequently, as a rough measure of sensitivity, we'd consider $\|\boldsymbol{A}^{-1}\mathbf{d}\|$.

As you might expect, the Hilbert matrix is sufficiently structured that we can just write down the inverse after some tedious calculation. (Or we can just look it up on—say—Wikipedia.)

$$A^{-1}_{ij} = (-1)^{i+j} \binom{n+i-1}{n-j} \binom{n+j-1}{n-i} \binom{i+j-2}{i-1}^2$$

This gets large extremely quickly.[2]

Hence, we have $\mathbf{y} = \mathbf{x} + \text{big} \cdot \text{small}$, so in general, we'll expect large changes to $\mathbf{y}$ if we slightly change our right hand side.

[2] It seems like a good exercise in combinatorics to work out what the one-norm of this matrix is!

3.3 A MORE REFINED ANALYSIS

> Aside, for those with additional background on numerical analysis, we suggest repeating this section with more sophisticated and accurate quadrature methods.

Recall that this problem is to approximate a function $f(x)$ by a polynomial. That problem itself actually is *well conditioned*.[3] Small changes to $f(x)$ give small changes to the polynomial.

The real problem here is that we have chosen to represent the polynomial in a monomial basis. This is know to result in problems that occur because monomials are an ill-conditioned basis for polynomials. Small changes to the monomial coefficients cause big changes to the polynomial functions they represent. In this case, an issue that arises for this problem is that *a small perturbation to* $\mathbf{b}$ actually gives a large perturbation to $f(x)$. Hence, we *should* see large changes to $\mathbf{x}$ even for small changes in $\mathbf{b}$.

[3] The matrix method is just one method of solving the original problem, that happens to result in a problem with an ill-conditioned linear system of equations.

Let's briefly study this perspective. Recall that

$$b_j = \int_0^1 f(x) x^{j-1} \, dx.$$

Suppose we use a crude approximation of the integral via a set of equally spaced points $x_1 = 0, x_2 = 1/N, x_3 = 2/N, \ldots, x_N = (N-1)/N, x_{N+1} = 1$, then

$$b_j \approx \sum_{k=0}^{N} f(x_{k+1}) x^{j-1} 1/N.$$

Let $\mathbf{f}$ be a length N vector of these function values $f_i = f(x_i)$. Then

$$\mathbf{b} \approx \frac{1}{N} \underbrace{\begin{bmatrix} 1 & 1 & \cdots & 1 \\ x_1 & x_2 & \cdots & x_N \\ x_1^2 & x_2^2 & \cdots & x_N^2 \\ \vdots & \vdots & \ddots & \vdots \\ x_1^{n-1} & x_2^{n-1} & \cdots & x_N^{n-1} \end{bmatrix}}_{=\boldsymbol{M}} \underbrace{\begin{bmatrix} f(x_1) \\ f(x_2) \\ f(x_3) \\ \vdots \\ f(x_N) \end{bmatrix}}_{\mathbf{f}} \quad \text{or more compactly} \quad \mathbf{b} = \boldsymbol{M}\mathbf{f}.$$

The matrix $\boldsymbol{M}$ is non-singular as long as the points $x_1, \ldots, x_N$ are distinct, which they are for equally spaced points. So given a change to $\mathbf{b}$, then $\boldsymbol{M}^{-1}\mathbf{b}$ gives the change to the function values that $\mathbf{b}$ represents.

3.4 A FORMAL ANALYSIS: THE CONDITION NUMBER

This type of analysis has been formalized by studying the condition number of a problem. Let

$$\mathbf{m}(\mathbf{x}) : \mathbb{R}^n \to \mathbb{R}^d$$

be a map that represents the mathematical relationship between the input to a computer method $\mathbf{m}$ to its output. For instance

- *addition* then $\mathbf{m}(\mathbf{x}) = x_1 + x_2$
- *subtraction* then $\mathbf{m}(\mathbf{x}) = x_1 - x_2$
- *variance* then $\mathbf{m}(\mathbf{x}) = \sum_i (x_i - 1/N \sum_j x_j))^2$
- *linear system* then $\mathbf{m}(\mathbf{b}) = \boldsymbol{A}^{-1}\mathbf{x}$

Note that, crucuially, $\mathbf{m}(\mathbf{x})$ represents the mathematical function we are trying to compute and this has no aspect of the computer implementation.

Quiz What is the map $\mathbf{m}$ for least squares?

Once we have these functions, then we can study their relative sensitivity.

DEFINITION 1 *The relative condition number of a map* $\mathbf{m}$ *relates the relative change of the output with respect to the relative change of the input. For a specific change,* $\mathbf{d}$*, the value is*

$$\kappa(\mathbf{x}, \mathbf{d}; \mathbf{m}) = \frac{\|\mathbf{m}(\mathbf{x}+\mathbf{d}) - \mathbf{m}(\mathbf{x})\| / \|\mathbf{m}(\mathbf{x})\|}{\|\mathbf{d}\| / \|\mathbf{x}\|}.$$

For the worst case on a differentiable function as $\|\mathbf{d}\| \to 0$*, we have*[4]

$$\kappa(\mathbf{x}; \mathbf{m}) = \frac{\|(\nabla \mathbf{m})(\mathbf{x})\|}{\|\mathbf{m}(\mathbf{x})\|} \|\mathbf{x}\|.$$

[4] Note that the Jacobian is $\lim \mathbf{d} \to 0 \|\mathbf{m}(\mathbf{x}+\mathbf{d}) - \mathbf{m}(\mathbf{x})\| \|\mathbf{d}\|$.

We need to be slightly careful with the choice of norm when we do these analyses in terms of the dimension of the Jacobian matrix.

— TODO – Expand on this with Gautschi's notes too.

EXAMPLE 2 *What is the condition number of the simple act of multiplying two numbers? The map* $m(x, y) = xy$*.*

3.5 THE CONDITION NUMBER OF A LINEAR SYSTEM

For the map $\mathbf{m}(\mathbf{b}) = \boldsymbol{A}^{-1}\mathbf{b}$ we have:

$$\nabla \mathbf{m} = \boldsymbol{A}^{-1} \quad \text{and} \quad \boldsymbol{A}\mathbf{x} = \mathbf{b}$$

in which case we have

$$\kappa(\mathbf{b};\mathbf{m}) = \|\boldsymbol{A}^{-1}\| \frac{\|\mathbf{b}\|}{\|\boldsymbol{A}^{-1}\mathbf{b}\|}$$

but in terms of the solution $\mathbf{x}$ we have:

$$\kappa(\mathbf{b};\mathbf{m}) = \|\boldsymbol{A}^{-1}\| \frac{\|\boldsymbol{A}\mathbf{x}\|}{\|\mathbf{x}\|}.$$

Note that for any vector $\mathbf{x}$ and an operator induced norm, we have:

$$\frac{\|\boldsymbol{A}\mathbf{x}\|}{\|\mathbf{x}\|} \leq \|\boldsymbol{A}\|$$

by the definition of an operator-induced norm. Thus

$$\kappa(\mathbf{b};\mathbf{m}) \leq \|\boldsymbol{A}\| \|\boldsymbol{A}^{-1}\|.$$

The condition number of solving a linear system arises so often that we give it a special name.

DEFINITION 3 (condition number of a matrix) *The condition number of a matrix is an upper bound on the condition number of solving a linear system of equations for a general solution* $\mathbf{x}$ *and a general right hand side* $\mathbf{b}$*. We write*

$$\kappa(\boldsymbol{A}) = \|\boldsymbol{A}\| \|\boldsymbol{A}^{-1}\|$$

which gives the condition number for any operator induced matrix norm. The choice of norm is typically the 2-norm unless otherwise specified.

3.6 THE CONDITION NUMBER OF A LEAST SQUARES SYSTEM

Consider a least squares problem with a full rank $\boldsymbol{A}$

$$\underset{\mathbf{x}}{\text{minimize}} \quad \|\boldsymbol{A}\mathbf{x} - \mathbf{b}\| \ .$$

There are two natural maps to consider:

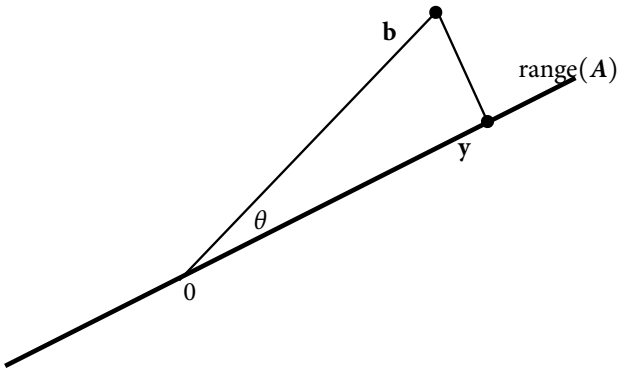
$$\text{a map from } \mathbf{b} \text{ to } \mathbf{x} \quad \mathbf{x} = (\boldsymbol{A}^T\boldsymbol{A})^{-1}\boldsymbol{A}^T\mathbf{b}$$

$$\text{a map from } \mathbf{b} \text{ to } \boldsymbol{A}\mathbf{x} \quad \mathbf{y} = \boldsymbol{A}(\boldsymbol{A}^T\boldsymbol{A})^{-1}\boldsymbol{A}^T\mathbf{b}.$$

The vector $\mathbf{y}$ is the vector of least squares predictions. Even if $\mathbf{x}$ might be sensitive, it's possible that $\mathbf{y}$ may be substantially less sensitive.

There is a nice geometric discussion of how to analyze conditioning for this problem in Trefethen and Bau.[5].

[5] Trefethen and Bau, Lecture 18, Theorem 18.1



A least squares problem involves the interaction of $\mathbf{b}$ with range$(\boldsymbol{A})$. The angle between the two is θ. If θ is zero, then $\mathbf{b}$ is in the range of $\boldsymbol{A}$ and the residual of the least squares problem is o$.

We can easily compute $\cos\theta$ as the ratio $\|\mathbf{y}\|/\|\mathbf{b}\|$. This quantity will play a role in the conditioning. If $\theta = 0$, then $\cos\theta = 1$, whereas if $\theta = \pm\pi/2$, then $\cos\theta = 0$.

3.6.1 The Pseudoinverse.

It's easiest to analyze this problem by establishing a single quantity for $(A^T A)^{-1} A^T$. This is the pseudoinverse.

$$A^+ = (A^T A)^{-1} A^T.$$

However, we will write this in terms of the SVD of A. Let $A = U\Sigma V^T$. Then

$$(A^T A)^{-1} = (V\Sigma^T \Sigma V^T)^{-1} = V(\Sigma^T \Sigma)^{-1} V^T$$

$$A^+ = (A^T A)^{-1} A^T = V(\Sigma^T \Sigma)^{-1} V^T V \Sigma^T U^T = V(\Sigma^T \Sigma)^{-1} \Sigma^T U^T.$$

The term $(\Sigma^T \Sigma)^{-1} \Sigma^T$ simplifies greatly. First, note that all matrices involved are diagonal. Since we are looking at full rank least squares problems, then $m \geq n$ and all $\sigma_i \neq 0$. Thus:

$$(\Sigma^T \Sigma)^{-1} \text{ is } n \times n, \text{ diagonal } = \begin{cases} \frac{1}{\sigma_i^2} & \text{on diagonal} \\ 0 & \text{otherwise.} \end{cases}$$

Hence, the entire matrix is:

$$(\Sigma^T \Sigma)^{-1} \Sigma^T = \ n \times m, \text{ diagonal } = \begin{cases} \frac{1}{\sigma_i} & \text{on diagonal} \\ 0 & \text{otherwise.} \end{cases}$$

This defines the pseudo-inverse of a diagonal matrix, which we often write:

$$\Sigma^+ = (\Sigma^T \Sigma)^{-1} \Sigma^T.$$

To recap: the pseudoinverse of A is

$$A^+ = V \Sigma^+ U^T.$$

3.6.2 The condition number of least squares in terms of the pseudoinverse.

Let $\mathbf{x} = A^+ \mathbf{b}$. Then the condition number of the least squares vector $\mathbf{x}$ in terms of $\mathbf{b}$ is:

$$\kappa(\mathbf{b}) = \frac{\|A^+\| \|\mathbf{b}\|}{\|\mathbf{x}\|} = \|A^+\| \frac{\|\mathbf{b}\|}{\|\mathbf{y}\|} \frac{\|\mathbf{y}\|}{\|\mathbf{x}\|} = \frac{\|A^+\|}{\cos\theta} \frac{\|\mathbf{y}\|}{\|\mathbf{x}\|}.$$

This second term $\frac{\|\mathbf{y}\|}{\|\mathbf{x}\|} = \frac{\|A\mathbf{x}\|}{\|\mathbf{x}\|}$ measures how large $\mathbf{y}$ is compared with $\mathbf{x}$. Clearly, this is less than $\|A\|$. So we have

$$\kappa(\mathbf{b}) \leq \|A^+\| \|A\| \frac{1}{\cos\theta}.$$

However, if we let $\frac{1}{\eta} = \frac{\|A\mathbf{x}\|}{\|A\| \|\mathbf{x}\|}$, then $\frac{\|\mathbf{y}\|}{\|\mathbf{x}\|} = \|A\| \frac{1}{\eta}$. Consequently, we have

$$\kappa(\mathbf{b}) = \|A^+\| \|A\| \frac{1}{\eta \cos\theta}.$$

Because the term $\|A^+\| \|A\|$ arises just as frequently as $\|A^{-1}\| \|A\|$, we extend the definition of $\kappa(A)$ with a rectangular matrix A.

3.6.3 The condition number of the least squares prediction.

Let $\mathbf{y} = AA^+ \mathbf{b}$. Then

$$\mathbf{y} = U\Sigma V^T V \Sigma^+ U^T \mathbf{b} = U\Sigma\Sigma^+ U^T \mathbf{b}.$$

The matrix AA^+ is an orthogonal projection onto range(A). What this means for us is that $\|AA^+\| = 1$ for the operator induced 2-norm. Consequently

$$\kappa(\mathbf{b}) = \|AA^+\| \frac{\|\mathbf{b}\|}{\|\mathbf{y}\|} = \frac{1}{\cos\theta}.$$

3.7 THE CONDITION NUMBER OF AN EIGENVALUE

3.8 RELATIONSHIP WITH FRESCHET DERIVATIVE

For the full-rank least squares system, the mathematical map from

4 BACKWARDS STABILITY

There are two aspects to numerical accuracy.[6] The first aspect is the conditioning of a problem taht we addressed above. The second aspect is the stability of the algorithm to compute it. This lecture is about algorithmic stability, not about the conditioning of a problem. Take the most trivial function $f(x) = x$. The following algorithm:

[6] Most of these notes and ideas are taken from Trefethen and Bau, Numerical Linear Algebra, Lecture 14 and Higham, Accuracy and Stability of Numerical Algorithms, Chapter 1.

```
for i=1 to 60
  x = √x

for i=1 to 60
  x = x^2
```

computes the this identity function for $x \geq 0$. Yet, if you run this algorithm on a computer, you will compute the function:

$$\tilde{f}(x) = \begin{cases} 0 & 0 \leq x < 1 \\ 1 & x \geq 1. \end{cases}$$

What this shows is that this *rather silly* algorithm is not a good idea.

Let's make this idea precise and develop a definition that would allow us to make a more precise statement. Let $\tilde{y} \approx f(x)$ and let $y = f(x)$. Ideally, we'd like the

$$\text{absolute error} = |\tilde{y} - y|$$

to be small. But if y is large, this isn't reasonble. So a better goal would be ensure the

$$\text{relative error} = |\tilde{y} - y|/|y|$$

is small. The problem with this type of error analysis is that we immediately encounter the conditioning of the underyling problem. That is, if the problem is ill-conditioned, we will not be able to show that the relative error is always small. This is independent of any algorithm that we have. What has proven to be useful instead is the idea of *backwards error analysis*. That is, we ask the question:

$$\text{is } \tilde{y} \text{ the } \textit{exact} \text{ computation of some } x + \delta?$$

Formally, can we show that an algorithm will have a small δ such that

$$\tilde{y} = f(x + \delta)?$$

If so, then we use this property to establish a relative error bound via the condition number argument:

$$|\tilde{y} - y|/|y| \text{ is something like } \kappa(y)|\delta|.$$

Let's put this slightly more formally now.

DEFINITION 4 *An algorithm is* backwards stable *for computing $y = f(x)$ if it computes $\tilde{y} = f(x + \delta)$ with $|\delta|/|x| \leq Cu$ where C is a constant and u is the machine precision.*[7]

[7] Throughout this note, u is the machine precision.

For a matrix problem $f(\mathbf{x})$, we apply this as:

$$\tilde{\mathbf{y}} = f(\mathbf{x} + \mathbf{d}) \text{ where } \|\mathbf{d}\|/\|\mathbf{x}\| \leq C_n u.$$

In this case, C_n may depend on the dimension n of the matrix.

One question immediately presents itself: Are floating point operations backwards stable? We defined

$$\text{fl}(x + y) = (x + y)(1 + \delta) \text{ for } |\delta| \leq u.$$

But we can just move the $(1+\delta)$ inside and we have:

$$\text{fl}(x+y) = x(1+\delta) + y(1+\delta).$$

Thus, we are computing the exact addition for a problem whose input is perturbed by $(1+\delta)$.

Now let's see how this analysis plays out in other cases as well.

4.1 ONE-PASS ALGORITHMS FOR VARIANCE

See online codes for this.

4.2 PAGERANK ALGORITHMS

Much of my experience with these issues arises when trying to compute accurate solutions of the PageRank linear system of equations. Let $\boldsymbol{A}$ be the adjacency matrix of a directed graph. An entry of $A_{ij} = 1$ when there is a link from node i to node j and an entry has a value of 0 otherwise. For PageRank, we mathematically model a Markov process that behaves as follows. At node i

- with probability α, we follow an out-link to another node. (If there are no out-links, then we jump to a page chosen uniformly at random.)
- with probability $(1-\alpha)$, we jump to a page chosen uniformly at random.

Let X_t be the identity of the node at step t. The PageRank vector of the graph is defined as the amount of time that this process spends in each node as it runs forever:

$$x_i = \lim_{T\to\infty} \frac{1}{T} \sum_{k=0}^{T} \begin{cases} 1 & X_t = i \\ 0 & \text{otherwise} \end{cases}.$$

There are a variety of ways to get the following condition, but in the interest of time, let's just note that we can get there by simply assuming that x_i exists like we did with the hitting time analysis.[8] If there is such a vector $\mathbf{x}$, we can convert this into a linear system of equations by noting that the value of x_i must depend on the other values x_j as in the next equation. Let d_j be the number of nodes that node j links to (so this is the out-degree of node j).

$$x_i = \alpha \sum_{j \text{ links to } i} \begin{cases} x_j/d_j & d_j > 0 \\ x_j/n & d_j = 0 \end{cases} + (1-\alpha) \underbrace{\sum_{j=1}^{n} x_j/n}_{\text{random jumps}}$$

[8] Formally, this would require showing that the above limit and sum have a unique limit. Both are actually the case, but involve a study in stochastic processes that is outside the scope of this material. We will show a related derivation to justify existence.

Let $\boldsymbol{P}$ be the matrix where

$$P_{i,j} = \begin{cases} 1/d_j & \text{node } j \text{ links to node } i, d_j > 0 \\ 0 & \text{otherwise.} \end{cases}$$

If $\boldsymbol{A}$ is the adjacency matrix described above then

$$\boldsymbol{P} = \boldsymbol{A}^T \boldsymbol{D}^{-1} \text{ where } D^{-1}_{ij} = \begin{cases} 1/d_i & i = j, d_j > 0 \\ 0 & \text{otherwise.} \end{cases}$$

The matrix $\boldsymbol{D}^{-1}$ is really the inverse of the diagonal matrix of node degrees, where we simply skip over any node with degree o.[9] Let $\mathbf{c}$ be the indicator vector where $c_i = 1$ for all nodes with $d_j = 0$ and $c_i = 0$ for all other nodes. Then

[9] This is often called the pseudoinverse.

$$\mathbf{x} = \alpha \boldsymbol{P}\mathbf{x} + \alpha/n \mathbf{e}\mathbf{c}^T\mathbf{x} + (1-\alpha)\mathbf{e}/n.$$

In may cases, we assume the vector $\mathbf{c} = 0$. In which case

Given the matrix $\boldsymbol{A}$, the following algorithm