Supplementary notes 1: A recursive treatment of backsubstitution and a first LU factorization algorithm

Kevin Long
Department of Mathematics and Statistics
Texas Tech University

August 31, 2015

The rest is just the same, isn't it?
- Mozart to Salieri, in *Amadeus*

If you understand only one thing about mathematics, it should be this: we never solve hard problems. We only transform hard problems into sequences of easy problems, then solve the easy problems.

Contents

1 Triangular matrices

Not all linear systems are hard to solve. Particularly easy is a problem with a diagonal matrix: each equation is solved with a single division for a total cost of N. Also easy are systems with triangular matrices, for example, the upper triangular system

$$\begin{aligned} U_{11}x_1 + U_{12}x_2 + U_{13}x_3 + \cdots + U_{1N}x_n &= b_1 \\ U_{22}x_2 + U_{23}x_3 + \cdots + U_{2N}x_N &= b_2 \\ &\vdots \\ U_{NN}x_N &= b_N. \end{aligned}$$

It should be clear how to solve this: start at the bottom and work your way up, solving for one variable in each step. It should also be clear that you can solve a *lower triangular* system in the same way, but working from top to bottom. Generically, L and U will be used for lower and upper triangular matrices[1].

You probably also recall the Gaussian elimination algorithm to transform a system to upper triangular form; we'll look at that soon enough. Let's prove some properties of triangular matrices.

2 Proof techniques for structured matrices

In undergraduate linear algebra you probably proved theorems such as

Theorem 2.1. *Let A be a nonsingular square matrix. Then $\left(A^T\right)^{-1} = \left(A^{-1}\right)^T$. We conventionally write both with the single symbol A^{-T}.*

Proof. From the properties of the transpose we know $(AB)^T = B^T A^T$. By definition of the inverse, we know $A^{-1}A = I$ and $AA^{-1} = I$, and also that the inverse is unique. Compute

$$\left(A^{-1}A\right)^T = I^T = I.$$

Use the product transpose property to transform to

$$A^T \left(A^{-1}\right)^T = I.$$

Therefore, $\left(A^{-1}\right)^T$ "plays the role of" an inverse of A^T in this equation. The inverse is unique, so $\left(A^{-1}\right)^T$ *is* the inverse of A^T, so we could write it as $\left(A^T\right)^{-1}$. □

This proof was very abstract, in the sense that we needed to know *nothing* about the internal layout of A. All we used were the operational properties of the transpose and inverse: how they behave when used in the operation of matrix multiplication. You'll recall many proofs of this sort from your elementary linear algebra courses.

A little thought should reveal that this sort of operational proof can't be used directly in proving properties of triangular matrices. If the strength of operational proofs is that they don't require any information about matrix layout, the weakness is that they can't reveal anything about it either. There are no hooks on which to hang any specification of structure.

[1] A word about notation: lower triangular matrices are sometimes called left triangular, because the nonzeros are both below and to the left of the diagonal. Therefore L is a logical symbol for generic lower (or left) triangular matrices, and indeed that's what people use. Either U or R would seem to be the logical choices for upper triangular / right triangular matrices. But U is also used to denote generic unitary matrices, making R the logical choice; unfortunately most books in English use L and U, trusting you to work out any ambiguity between "upper" and "unitary" from context. Authors from the German-speaking world tend to use the more logical L and R (*link und recht*). Out of years of habit and for consistency with most English-language sources I'll use L and U and leave you to sort out any confusion that results.

What to do. Two ideas will get us there: partitioning and recursion. Write an N by N block upper triangular matrix as follows

$$T = \begin{pmatrix} \tau & t^T \\ 0 & T_1 \end{pmatrix}$$

in terms of a scalar τ, a length $N-1$ row vector t^T, and an $N-1$ square matrix T_1. We'll use this sort of partitioning often, so make sure you understand how the pieces fit together to make an N by N matrix. As a convention (following Stewart) I'll usually use lower-case Greek letters for the scalar, lower-case Latin letters for the vector, and upper-case Latin letters for the matrices.

If (and only if) T_1 is UT, the whole matrix T_1 is UT. We could then contain T as the lower corner of a block UT matrix, and then *that* matrix must be UT, and so on. By recursive partitioning we can build UT matrices using only two by two systems at each step.

As an example of what we can do with this idea, let's prove the very useful theorem

Theorem 2.2. *Let S and T be two $n \times n$ upper triangular matrices. Then ST is upper triangular.*

Proof. All one-by-one matrices are UT, so the theorem is clearly true for $n = 1$. Assume the inductive hypothesis that for some n, given any pair of $n \times n$ UT matrices S' and T' it is true that $S'T'$ is UT. Form a pair of block matrices

$$S = \begin{pmatrix} \sigma & s^T \\ 0 & S' \end{pmatrix}$$

$$T = \begin{pmatrix} \tau & t^T \\ 0 & T' \end{pmatrix}.$$

Then compute

$$ST = \begin{pmatrix} \sigma & s^T \\ 0 & S' \end{pmatrix} \begin{pmatrix} \tau & t^T \\ & T' \end{pmatrix} = \begin{pmatrix} \sigma\tau & \sigma t^T + \tau s^T \\ 0 & S'T' \end{pmatrix}.$$

The lower right entry is UT by hypothesis, so if the theorem is true for any n, it is by induction true for all larger n. The theorem is true for $n = 1$, so it's true for all n. □

This is a classic inductive proof; we'll see lots of them when working with matrices. If you're not familiar with inductive proofs, buy a mathematics or computer science student a bottle of wine and get them to show you a few simple examples.

3 Backsubstitution viewed inductively

Let's look at $Ux = b$ through the lens of induction. Write the problem in partitioned form as

$$\begin{pmatrix} \upsilon & u^T \\ 0 & U' \end{pmatrix} \begin{pmatrix} \xi \\ x' \end{pmatrix} = \begin{pmatrix} \beta \\ b' \end{pmatrix}.$$

Now assume that we've been able to solve $U'x' = b'$ for x'. We can write this ***formally*** as $x' = U'^{-1}b'$. I emphasize the word formally because ***we do not actually form the inverse matrix*** U'^{-1}**.** The inverse notation is shorthand for the equivalent (in theory) operation "carry out backsubstitution to solve $U'x' = b'$." Then we can eliminate x' from the top row of the block equation, obtaining

$$\upsilon\xi = \beta - u^T x'$$

and finally

$$\xi = \frac{1}{\upsilon}\left(\beta - u^T x'\right).$$

This fails iff $\upsilon = 0$. Because we can solve a one-by-one UT system (barring a zero divisor), by induction we can solve any UT system in which none of the diagonal elements are zero. This formally proves that the backsubstitution procedure works.

This might seem like overkill for such a simple problem, but it's a technique we'll use frequently. Moreover, it leads to a simple method of computing the cost of backsubstitution.

4 Computing the cost of backsubstitution

We've broken down backsubstitution into three steps:

1. Solve the smaller system $U'x' = b'$ for x'
2. Form the scalar product $\gamma = u^T x'$
3. Compute $\frac{1}{v}(\beta - \gamma)$

How many operations are needed for each step? First, though, what's an operation? The unit of cost will be the *flop*, for floating-point operation. We assume all "atomic" floating point operations have roughly equivalent cost, *i.e.*, a division costs about the same as an addition, and so on. On any modern computer this can be assumed to be true for any of the usual calculator operations: addition/subtraction, multiplication/division, square roots, and so on. Some computers might have specialized composite operations, for example the ability to perform $\alpha + \beta\gamma$ more efficiently than the product $\beta\gamma$ followed by the sum $\alpha + (\beta\gamma)$. You'll sometimes see such combined operations counted as *flams*, for floating-point add/multiply. won't worry about such distinctions; for us, a flop is a flop is a flop. In a few weeks we'll talk more about floating point math; for now, simply assume that a computer can do elementary arithmetic calculations efficiently and correctly (to some precision).

Working backwards: step 3 takes 2 flops, one subtraction, one division. Step 2 takes $n-1$ multiplies and $n-2$ adds. As for step 1, we don't know that cost. After all, the cost of a backsolve is exactly what we're trying to find. But call it $F(n-1)$, the (unknown) cost in flops of a backsolve on a UT system with $n-1$ variables. Then we can write the cost of the size n backsolve in terms of the size $n-1$ backsolve,

$$F(n) = \underbrace{2}_{\text{step 3}} + \underbrace{2n-3}_{\text{step 2}} + \underbrace{F(n-1)}_{\text{step 1}}.$$

This is only true for $n > 1$, because when $n = 1$ we can skip steps 1 and 2, and do step 3 with one flop only (because $\gamma = 0$). Therefore $F(1) = 1$. Then compute $F(2)$, $F(3)$, and so on. That works but doesn't provide much insight. More useful is to use the recurrence relation to find a closed-form expression for the cost. Simplify it to

$$F(n) = 2n - 1 + F(n-1)$$

and expand

$$F(n) = 2n - 1 + (2(n-1) - 1) + F(n-2)$$

and so on, to recognize that

$$F(n) = \sum_{k=1}^{n} (2k-1).$$

We can compute this sum exactly using induction, and approximate it in several ways.

4.1 Exact calculation of the backsubstitution cost

First, we'll compute the exact value by summing the series.

Theorem 4.1. *The finite series*

$$F(n) = \sum_{k=1}^{n} (2k-1)$$

sums to

$$F(n) = n^2.$$

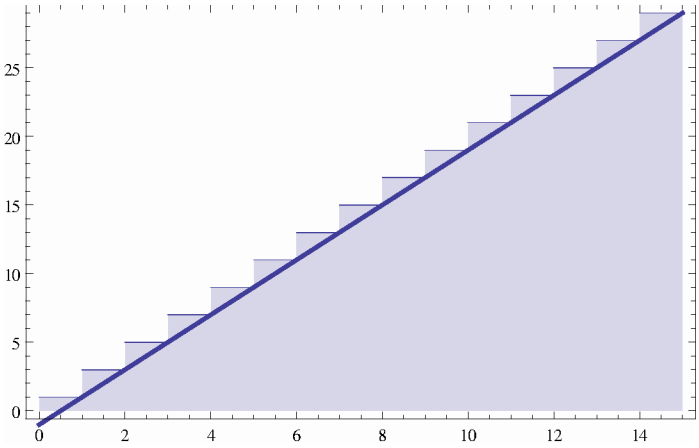

Figure 1: The series $\sum_{k=1}^{n} 2k-1$ represented as the area under a piecewise constant function, and approximately as the definite integral of a continuous function. As the number of steps becomes large the approximation becomes better.

Proof. You can check by direct calculation that the formula is correct for $n = 1$. Assume it to be true for some $n-1$. Then

$$F(n) = n^2$$

$$F(n-1) = (n-1)^2$$

and so

$$F(n) - F(n-1) = n^2 - n^2 + 2n - 1$$

and

$$F(n) = 2n - 1 + F(n-1)$$

which is the recurrence relation that defined the sum. If true for any $n-1$, it is true for n, and we know it to be true for $n = 1$. We're done. □

4.2 Approximate calculation of the backsubstitution cost

Doing finite sums such as

$$\sum_{k=1}^{n} 2k - 1$$

$$\sum_{k=1}^{n} (k+7)^2 + 2k - 1$$

exactly by induction isn't hard, but is a little messy. Luckily there are some shortcuts that are nearly always good enough for our purposes. The first thing to remember is that we're usually interested in very large n: hundreds or more. The exact sum is the area under a piecewise constant function, as illustrated in figure 1. For large n that area is reasonably well approximated by the area under the continuous curve drawn under the steps. We know how to compute areas under continuous functions as definite integrals. So, for example,

$$\sum_{k=1}^{n} (2k-1) \approx \int_0^n (2k-1)\, dk = n^2 - n.$$

This is clearly not an exact calculation (we know the sum is n^2 instead of $n^2 - n$) but for large n the approximation is a very good. When $n = 100$ the exact sum is 10000, the approximation is 9900 for a 1% relative error.

Definition 4.1. A quantity $f(x)$ is said to be $O(x^p)$ if $\exists$ a constant $C > 0$ such that $|f(x)| \leq Cx^p$ for all $x > 0$. We'll often say that f is p-th order in x.

For example, the cost of solving $Ux = b$ for U $n \times n$ UT is $O(n^2)$.

Definition 4.2. The *order constant* of a cost estimate is the coefficient of the highest power of n.

For example, the order constant of the UT backsolve is 1, because the highest power of n in the computed cost is n^2 and its coefficient is 1.

If you understand the Riemann theory of the definite integral, you will see that it's not hard to prove that the approximation of a sum

$$\sum_{k=1}^{n} k^p \quad p \geq 0$$

by a definite integral

$$\sum_{k=1}^{n} k^p \approx \int_0^n x^p \, dx \quad p \geq 0$$

gives both the correct order n^{p+1} and the correct order constant $\frac{1}{p+1}$. We will nearly always use this approximation in place of exact calculation of finite sums; there is rarely any point to doing a cost calculation exactly. What usually matters is the dominant term.

4.3 Visual estimate of backsubstitution cost

Finally, there's a very quick graphical shortcut to estimating the cost of backsubstitution. An $n \times n$ matrix has "area" n^2 elements; a UT matrix has roughly half that area made up of zero entries so the cost will be $\sim \frac{1}{2}n^2$ times the number of operations per element. In the backsubstitition algorithm there are roughly two operations per element: multiplying U_{ij} by x_j and then subtracting from b_i. So the total cost is roughly n^2.

5 A first look at computing the LU factorization: the Sherman's March algorithm

We can use partitioned matrices to develop a recursive algorithm for computing an LU factorization.

Let A be $n \times n$ and nonsingular. In the case $n = 1$, there's an easy factorization with L unit diagonal: $L = 1$, $U = A$. For $n > 1$, write A in partitioned form,

$$A = \begin{pmatrix} A' & b \\ c^T & \delta \end{pmatrix}.$$

The top corner submatrix A' is $(n-1)$ by $(n-1)$ and we'll assume it has a known factorization $L'U'$. Next write the LU factorization of A in partitioned form (with ones on the diagonal of L), and multiply through:

$$\begin{pmatrix} L' & 0 \\ \ell^T & 1 \end{pmatrix} \begin{pmatrix} U' & u \\ 0 & v \end{pmatrix} = \begin{pmatrix} L'U' & L'u \\ \ell^T U' & \ell^T u + v \end{pmatrix}.$$

By hypothesis we already have L' and U'. By equating the product LU to A we have the equations

$$\begin{aligned} L'u &= b \\ U'^T \ell &= c \\ v + \ell^T u &= \delta. \end{aligned}$$

The first two equations are LT and can be solved by backsubstitution to obtain u and ℓ. With u and ℓ in hand the third equation can then be solved for the scalar v. We've thus constructed the factorization of an $n \times n$ matrix given the factorization of its upper left $(n-1) \times (n-1)$ submatrix. Since we already know we can start the procedure by factoring a one-by-one matrix, we have completed an inductive proof that this algorithm forms a factorization, *unless* one of the steps fails.

Algorithm 1 Sherman's March implemented in MATLAB

```
function [L, U] = sherman2(A)

[M, N] = size(A);

L = eye(N);
U = zeros(N,N);
U(1,1)=A(1,1);


for i=2:N
  a = A(1:i-1, i);
  b = A(i, 1:i-1)';
  gamma = A(i,i);
  u = L(1:i-1,1:i-1) \ a;
  ell = U(1:i-1,1:i-1)' \ b;
  upsilon = gamma - ell' * u;
  U(1:i-1,i)=u;
  L(i,1:i-1)=ell;
  U(i,i)=upsilon;
end
```

What can go wrong? The backsolve against L cannot fail because L has unit diagonal. The backsolve against U will fail if one of the diagonal entries of U' is zero. How can this happen? Notice that the last diagonal entry of U will be the number v computed in the previous step. This number is called the *pivot* for the current cycle of the algorithm. If it happens that $\gamma = \ell^T u$, then the pivot is zero and the next step of the algorithm will break down. If a zero pivot is never encountered, then the algorithm runs to completion and an LU factorization exists for A.

In his book *Matrix Algorithms,* Stewart calls this algorithm *Sherman's March*[2]because it proceeds through the matrix from northwest to southeast. I consider it the simplest way to introduce LU factorization; among other reasons, the cost estimate is quite simple. The main drawback to Sherman's March is that we can't identify a zero pivot until after the step has been done, by which time it's too late to fix by row exchange. A simple variation on Sherman's March applied to symmetric positive definite (SPD) matrices will give us an efficient and failure-proof algorithm the Cholesky factorization.

Since Sherman's March will fail upon a zero pivot it's rarely used in practice (except in its Cholesky variant, where zero pivots are impossible). Later we'll study an algorithm – Gaussian elimination – that can cope with zero pivots.

Sherman's March is easily implemented in MATLAB, as shown in algorithm 1. I've left it uncommented to encourage you to read the code carefully. This program accepts a matrix A and returns the factors L and U; that is not optimally efficient because it requires storage of all three matrices. High-performance implementations will overwrite the lower part of A with L (skipping the known ones on the diagonal) and the upper part with U.

5.1 An example of Sherman's March

Let's use Sherman's March to compute the LU factorization of

$$A = \begin{bmatrix} 2 & 3 & 1 & 2 \\ 4 & 7 & 3 & 6 \\ 6 & 11 & 9 & 11 \\ 4 & 7 & 11 & 10 \end{bmatrix}.$$

[2]For those who don't know US history: Sherman's "march to the sea" was a campaign in the US Civil War in which Union forces under General W. T. Sherman advanced Southeast from Atlanta to the Georgia coast.

We start from the upper left (northwest). The first submatrix is the one-by-one matrix $A' = 2$ which factors to $L' = 1$, $U' = 2$.

- **Level 1:**
  1. Identify $L'\backslash U' = 2$, $b = 3$, $c^T = 4$, $\delta = 7$
  2. Solve $L'u = b \implies u = 3$
  3. Solve $U'^T\ell = c \implies \ell = 2$
  4. Compute $v = \delta - \ell^T u \implies v = 1$
  5. Form L' and U' for next level
     - $L' = \begin{bmatrix} 1 & 0 \\ 2 & 1 \end{bmatrix} \quad U' = \begin{bmatrix} 2 & 3 \\ 0 & 1 \end{bmatrix}$
     - Packed: $L'\backslash U' = \begin{bmatrix} 2 & 3 \\ 2 & 1 \end{bmatrix}$

- **Level 2:**
  1. Identify $L'\backslash U' = \begin{bmatrix} 2 & 3 \\ 2 & 1 \end{bmatrix}$, $b = \begin{bmatrix} 1 \\ 3 \end{bmatrix}$, $c^T = \begin{bmatrix} 6 & 11 \end{bmatrix}$, $\delta = 9$
  2. Solve $L'u = b$: $\begin{bmatrix} 1 & 0 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \end{bmatrix} \implies u = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$
  3. Solve $U'^T\ell = c$: $\begin{bmatrix} 2 & 0 \\ 3 & 1 \end{bmatrix} \begin{bmatrix} \ell_1 \\ \ell_2 \end{bmatrix} = \begin{bmatrix} 6 \\ 11 \end{bmatrix} \implies \ell = \begin{bmatrix} 3 \\ 2 \end{bmatrix}$
  4. Compute $v = \delta - \ell^T u \implies v = 4$
  5. Form L' and U' for next level
     - $L' = \begin{bmatrix} 1 & & \\ 2 & 1 & \\ 3 & 2 & 1 \end{bmatrix} \quad U' = \begin{bmatrix} 2 & 3 & 1 \\ & 1 & 1 \\ & & 4 \end{bmatrix}$
     - Packed: $L'\backslash U' = \begin{bmatrix} 2 & 3 & 1 \\ 2 & 1 & 1 \\ 3 & 2 & 4 \end{bmatrix}$

- **Level 3**
  1. Identify $L'\backslash U'$, $b = \begin{bmatrix} 2 \\ 6 \\ 11 \end{bmatrix}$, $c^T = \begin{bmatrix} 4 & 7 & 11 \end{bmatrix}$, and $\delta = 10$
  2. Solve $L'u = b$:
     - $\begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 3 & 2 & 1 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} = \begin{bmatrix} 2 \\ 6 \\ 11 \end{bmatrix} \implies u = \begin{bmatrix} 2 \\ 2 \\ 1 \end{bmatrix}$
  3. Solve $U'^T\ell = c$
     - $\begin{bmatrix} 2 & & \\ 3 & 1 & \\ 1 & 1 & 4 \end{bmatrix} \begin{bmatrix} \ell_1 \\ \ell_2 \\ \ell_3 \end{bmatrix} = \begin{bmatrix} 4 \\ 7 \\ 11 \end{bmatrix} \implies \ell = \begin{bmatrix} 2 \\ 1 \\ 2 \end{bmatrix}$
  4. Compute $v = \delta - \ell^T u$: $\implies \delta = 2$
  5. Form L and U
     - $L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 \\ 3 & 2 & 1 & 0 \\ 2 & 1 & 2 & 1 \end{bmatrix} \quad U = \begin{bmatrix} 2 & 3 & 1 & 2 \\ 0 & 1 & 1 & 2 \\ 0 & 0 & 4 & 1 \\ 0 & 0 & 0 & 2 \end{bmatrix}$

5.2 The cost of Sherman's March

The n-th level of Sherman's march requires

1. One pre-existing factorization of an $(n-1) \times (n-1)$ matrix
2. Two triangular solves for $n-1$ variables each
3. One dot product of two length $n-1$ vectors, and one subtraction

Therefore the cost of factoring at level $n$is

$$F(n) = \underbrace{F(n-1)}_{\text{cost of factoring previous level}} + \underbrace{2(n-1)^2}_{\text{two triangular solves}} + \underbrace{2\,(n-1)}_{\text{two dot products}} + \underbrace{1}_{\text{subtraction}}$$

from which we can recognize the summation

$$F(n) = \sum_{k=1}^{n} \left[2\,(k-1)^2 + 2k - 1\right].$$

The sum can be evaluated exactly, but as usual we'll approximate it by integrating the leading term

$$F(n) \approx \int_0^n 2k^2\,dk = \frac{2}{3}n^3.$$

It turns out that this cost estimate is correct for any of the various methods for computing LU factorizations, with the exception of those using full pivoting (which is rarely used).

6 Applications of the LU factorization

6.1 Using the LU factorization to solve a system of equations

The most important application of the LU factorization is to solve system of equations. Since triangular systems are easy to solve, we can solve

$$Ax = b$$

by factoring A to LU, turning our system into

$$LUx = b.$$

Writing y in place of Ux this system is just

$$Ly = b$$

which is solved easily by forward substitution. Then recover x from y by solving

$$Ux = y$$

by backsubstitution. We can describe this procedure formally with the notation

$$x = U^{-1}L^{-1}b$$

but with the understanding that we *do not actually form the inverse matrices U^{-1} and L^{-1}*. The notation $y = L^{-1}b$ is convenient shorthand for "perform forward substitution on $Ly = b$ to compute y."

Example 6.1. Solve $Ax = b$ with

$$A = \begin{bmatrix} 2 & 3 & 1 & 2 \\ 4 & 7 & 3 & 6 \\ 6 & 11 & 9 & 11 \\ 4 & 7 & 11 & 10 \end{bmatrix}$$

and $b = \begin{bmatrix} 2 & 0 & 2 & 0 \end{bmatrix}^T$. From section 5.1 we have

$$L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 \\ 3 & 2 & 1 & 0 \\ 2 & 1 & 2 & 1 \end{bmatrix} \quad U = \begin{bmatrix} 2 & 3 & 1 & 2 \\ 0 & 1 & 1 & 2 \\ 0 & 0 & 4 & 1 \\ 0 & 0 & 0 & 2 \end{bmatrix}.$$

First solve $Ly = b$ by forward substitution:

$$y_1 = b_1 = 2$$

$$y_2 = b_2 - 2y_1 = -4$$

$$y_3 = b_3 - 2y_2 - 3y_1 = 4$$

$$y_4 = b_4 - 2y_3 - y_2 - 2y_1 = -8$$

Then solve $Ux = y$ by backsubstitution:

$$x_4 = \frac{1}{2} y_4 = -4$$

$$x_3 = \frac{1}{4}(y_3 - x_4) = 2$$

$$x_2 = y_2 - 2x_4 - x_3 = 2$$

$$x_1 = \frac{1}{2}(y_1 - 2x_4 - x_3 - 3x_2) = 1.$$

The solution is $x = \begin{bmatrix} 1 & 2 & 2 & -4 \end{bmatrix}^T$.

The cost of the entire procedure for an $m \times m$ matrix is

$$\text{cost to solve } m \times m \text{ system} = \underbrace{\frac{2}{3}m^3}_{\text{LU factorization}} + \underbrace{m^2}_{\text{solving } Ly = b} + \underbrace{m^2}_{\text{solving } Ux = y}$$

$$\frac{2}{3}m^3 + 2m^2.$$

When $m \gg 1$ the factorization cost far outweighs the backsubstitution cost.

Example 6.2. Suppose you want to solve a single 30 by 30 system. The cost of factorization is $\frac{2}{3}m^3 = 18000$ flops. The cost of the two backsubstitutions is $2m^2 = 1800$ flops, only 10% of the factorization cost. Now increase that to a 300 by 300 system. Now the factorization cost is 18 million flops, while the backsubstitution cost is 180 thousand flops: 1% of the factorization cost.

6.2 Solving multiple systems

Often in applications we'll need to solve many systems sequentially using the same matrix

$$Ax_1 = b_1$$

$$Ax_2 = b_2$$

and so on. With the LU factorization this can be done efficiently because the factorization needs to be done only once. Suppose there are N different right hand side vectors $\{b_i\}_{i=1}^N$ against which you must solve. The total cost is the factorization cost plus the cost of two triangular solves per system, or

$$\text{cost to solve } N \text{ systems with same matrix} = \underbrace{\frac{2}{3}m^3}_{\text{LU factorization}} + \underbrace{2Nm^2}_{\text{triangular solves}}.$$

6.3 Should I form A^{-1}?

In elementary linear algebra you were perhaps told (as I once was) that when you have multiple systems using the same matrix, it's cost effective to form the inverse and then use it to find $x_1 = A^{-1}b_1$, $x_2 = A^{-1}b_2$, and so on. This is incorrect! You can always do better by computing an LU factorization and reusing it as shown above.

Recall that to compute A^{-1} you solve the system

$$AX = I$$

for $A^{-1} = X$. For a $m \times m$ matrix there are m columns in I, so computing A^{-1} requires the solution of m systems. From the previous section, that cost is

$$\text{cost to form } A^{-1} = \frac{8}{3}m^3.$$

Once A^{-1} is formed, you can solve $Ax_k = b_k$ by multiplication: $x_k = A^{-1}b_k$. The cost of a matrix-vector multiplication is about $2m^2$ (roughly one multiply and one add per element; the exact cost is $2m^2 - m$). So the cost to use the inverse to solve N systems is the cost of forming the inverse plus the multiplication cost, or

$$\text{cost to solve } N \text{ systems using } A^{-1} = \underbrace{\frac{8}{3}m^3}_{\text{forming } A^{-1}} + \underbrace{2Nm^2}_{\text{multiplications by } A^{-1}}.$$

This is *always* greater than the cost of solving the same problem using LU factorization, by an additional $2m^3$ flops. That is precisely the cost of the additional triangular solves you had to do to solve $LUX = I$ for X. When efficiency is a concern, you should not form a matrix inverse.

6.4 Computing a determinant

One of the first things we all learned in elementary linear algebra was how to compute determinants by expansion in minors. You've been asked to compute the cost of that algorithm in a homework problem so I won't reveal the cost here; suffice to say that expansion in minors is expensive enough to be effectively useless in practical calculations. It is also very susceptible to roundoff error.

So how *can* we compute a determinant? Use an LU factorization! Suppose that A has an unpermuted LU factorization

$$A = LU$$

where $L$is LTUD and U is UT. Recall the theorem about products of determinants: $\det(AB) = \det(A)\det(B)$. So if $A = LU$ then

$$\det(A) = \det(L)\det(U).$$

Now the determinant of a triangular matrix is easy to compute: it's the product of the diagonal entries. The factor L has unit diagonal, so $\det(L) = 1$. Therefore, we can compute $\det(A)$ as

$$\begin{aligned}\det(A) &= \det(U) \\ &= \prod_{i=1}^{m} U_{ii}.\end{aligned}$$

The cost of an LU factorization is far less than the cost of expansion of $\det(A)$ in minors. Should you ever need to compute a determinant in practice, this is how to do it.

Example 6.3. Using the LU factorization computed in section 5.1, we find the determinant of the matrix

$$A = \begin{bmatrix} 2 & 3 & 1 & 2 \\ 4 & 7 & 3 & 6 \\ 6 & 11 & 9 & 11 \\ 4 & 7 & 11 & 10 \end{bmatrix}$$

is $2 \times 1 \times 2 \times 4 = 16$.

7 Summary

This little paper has covered much ground. We've seen:

- How to use partition and induction to prove theorems about structured matrices, with a few examples for triangular matrices.
  - A recursive analysis of backsubstitution for solving $Ux = b$
- Computational cost estimation
  - How to use recursion to compute the cost in flops of an algorithm
  - How to estimate cost to leading order by replacing sums by integrals
  - How to estimate cost by a geometric construction
- Sherman's March: a simple algorithm for LU factorization
  - Identifying (after the fact) when it succeeds or fails
  - Estimation of its cost: $\frac{2}{3}m^3$ for an $m \times m$ matrix
- Application of the LU factorization
  - Solving a single system by a factorization and two triangular solves
  - Reuse of a factorization to solve multiple systems
  - How to compute an inverse, and why you shouldn't do so
  - How to use an LU factorization to compute a determinant

It should be emphasized that although we've so far learned to compute LU factorizations only with Sherman's March, the applications of the LU factorization do not depend on how the factorization was computed. That's an important feature of the factorization approach to numerical linear algebra: you can think about the procedure $x = U^{-1}L^{-1}b$ for solving $Ax = b$ at the "big picture" level: get a factorization then do two triangular solves, without worrying about the details of how those steps are done any more than you worry about how your computer does division or square roots.