Lecture `linalg2`: Triangular and tridiagonal linear systems

September 26, 2022

Summary: Description and complexity estimates for some special linear systems.

References: Some aspects taken from C. F. Van Loan's *Introduction to Scientific Computing*.

Triangular systems

Let's take a closer look at triangular systems, considered before in lecture 6. We'll focus on upper triangular systems, but our analysis and observations also pertain to lower triangular systems. Our goals are both to calculate the number of operations needed to invert an upper triangular system, and describe how such an inversion is numerically implemented. Recall that an *upper triangular system* has the form

$$U\mathbf{x} = \mathbf{b} \quad \Longleftrightarrow \quad \begin{pmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ 0 & u_{22} & \cdots & u_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & u_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}. \tag{1}$$

As described in lecture 6, we can solve such a system by **backward substitution**:

$$x_k = \left(b_k - \sum_{j=k+1}^{n} u_{kj} x_j \right) \Big/ u_{kk}, \quad \text{for } k = n, n-1, \cdots, 1. \tag{2}$$

Backward substitution works so long as $\det U = u_{11}u_{22}\cdots u_{nn} \neq 0$, that is all of the diagonal elements $u_{kk} \neq 0$. Here were are using n for the system dimension (at other times we have used N).

A *floating point operation* or *flop* is a single arithmetical operation such as a multiplication, addition, subtraction, or division. The number of flops needed to perform a given algorithm is a measure of how much work a computer (or a person!) must expend in carrying it out. If an algorithm requires many flops, then it is expensive. The more expensive an algorithm, the longer it takes to execute on a computer. Therefore, we are often quite interested in estimating the operation count (or complexity) of a given algorithm. Estimation is often necessary, as in practice we can not and need not know the exact count.

Fact: The backward substitution algorithm requires $O(n^2)$ flops. The order symbol O here means that

$$\lim_{n\to\infty} \frac{\text{exact number of operations needed to carry out algorithm}}{n^2} = K,$$

where K is a constant. For backward substitution, K happens to be 1, but this is not essential. The main point is that the algorithm grows quadratically (like n^2) with the system size n. The idea here is that an $n = 10$ system will require a $100K$ work amount, and an $n = 100$ system will require a $10,000K$ work amount. Therefore, running the algorithm for $n = 100$ will take about 100 times longer, than running it for $n = 10$.

In order to establish the $O(n^2)$ counting for backward substitution, we write out the first few steps

$$\begin{aligned}
x_n &= b_n \oslash u_{nn}, \quad 1 \text{ flop} \\
x_{n-1} &= \big[b_{n-1} \ominus u_{n-1,n} \otimes x_n\big] \oslash u_{n-1,n-1}, \quad 3 \text{ flops} \\
x_{n-2} &= \big[b_{n-2} \ominus (u_{n-2,n-1} \otimes x_{n-1}) \ominus (u_{n-2,n} \otimes x_n)\big] \oslash u_{n-2,n-2}, \quad 5 \text{ flops} \\
x_{n-3} &= \big[b_{n-3} \ominus (u_{n-3,n-2} \otimes x_{n-2}) \ominus (u_{n-3,n-1} \otimes x_{n-1}) \ominus (u_{n-3,n} \otimes x_n)\big] \oslash u_{n-3,n-3}, \quad 7 \text{ flops},
\end{aligned}$$

with the symbols $\otimes$ (multiplication), $\ominus$ (subtraction), and $\oslash$ (division) each representing a single flop. As we work our way backwards $n, n-1, \cdots, 1$, we pick up an extra two flops with each new row, and by inspection find the formula

$$\text{operations for row } k = 2(n-k) + 1. \tag{3}$$

Therefore,

$$\text{total operations} = \sum_{k=1}^{n} \big[2(n-k)+1\big] = (2n+1)\sum_{k=1}^{n} 1 - 2\sum_{k=1}^{n} k \tag{4}$$

We now use the familiar equation for the sum of sequential numbers $\sum_{k=1}^{N} k = \frac{N(N+1)}{2}$ to obtain

$$\text{total operations} = (2n+1)n - 2\frac{n(n+1)}{2} = n^2 \tag{5}$$

In this case, we take *exactly* n^2 flops to compute the inverse of this upper-triangular problem. We thus have

$$\text{total operations} = O(n^2). \tag{6}$$

Here we've calculated exactly n^2 flops, but we would write the last equation even if we had calculated, say, total operations $= 3n^2$ or $5n^2 - n + 1$.

Now that we have theoretically established the operation count for backward substitution, let's consider how to practically implement the algorithm. On page 212 of the textbook by C. F. Van Loan (*Introduction to Scientific Computing, a Matrix–Vector Approach Using* MATLAB, second edition), one finds the following Matlab implementation (here we correct some "bugs" in Van Loan's comments and add a few of our own).

```
   function x = UTriSolve(U,b)
%           x = UTriSolve(U,b)
%  Solves the nonsingular upper triangular system Ux = b,
%  where U is n-by-n, b is n-by-1, and x is n-by-1. Note
%  that the algorithm has been vectorized with successive
%  overwriting of b.

   n = length(b);
   x = zeros(n,1);
   for j = n:-1:2
       x(j) = b(j)/U(j,j);
       b(1:j-1) = b(1:j-1) - x(j)*U(1:j-1,j);
   end
   x(1) = b(1)/U(1,1);
```

This implementation of the algorithm has been cleverly vectorized. Indeed, notice from the formulas above, that, for example, $u_{k,n}x_n$ will need to be subtracted from each b_k for all $k = n-1, n-2, \cdots 1$. This is done in one step by overwriting the first $n-1$ entries of b as follows:

$$\begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_{n-1} \end{pmatrix} \longrightarrow \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_{n-1} \end{pmatrix} - \begin{pmatrix} u_{1n}x_n \\ u_{2n}x_n \\ \vdots \\ u_{n-1,n}x_n \end{pmatrix}. \tag{7}$$

In Matlab this is achieved with the statement `b(1:n-1) = b(1:n-1) - U(1:n-1,n)*x(n)`, which is equivalent to the expression inside the `for` loop from `UTriSol` when $j = n$. Although we shall not explain exactly why, vectorization of the algorithm in this way ensures that it will run fast on most computers.

The same conclusions hold for lower triangular systems,

$$L\mathbf{x} = \mathbf{b} \quad \Longleftrightarrow \quad \begin{pmatrix} \ell_{11} & 0 & \cdots & 0 \\ \ell_{21} & \ell_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ \ell_{n1} & \ell_{n2} & \cdots & \ell_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}, \tag{8}$$

which can be solved via **forward substitution**:

$$x_k = \left(b_k - \sum_{j=1}^{k-1} \ell_{kj} x_j \right) \Bigg/ \ell_{kk}, \quad \text{for } k = 1, 2, \cdots, n, \tag{9}$$

provided all $\ell_{kk} \neq 0$. That is to say, forward substitution is an $O(n^2)$ algorithm, and Van Loan provides a vectorized implementation `LTriSol` analogous to `UTriSol`.

Tridiagonal systems

Another special type of linear system is a tridiagonal system $T\mathbf{x} = \mathbf{b}$. While a tridiagonal matrix T is neither upper nor lower triangular, it can be written as the product $T = LU$ of a bidiagonal lower triangular matrix L (all entries below the subdiagonal are zero) and a bidiagonal upper triangular matrix U (all entries above the superdiagonal are zero). The inversion algorithm for a tridiagonal system $T\mathbf{x} = \mathbf{b}$ then goes as follows.

1. Construct the "LU–factorization" $T = LU$. As shown below, this step takes $O(n)$ work.

2. Solve $L\mathbf{y} = \mathbf{b}$ for $\mathbf{y} = U\mathbf{x}$ via forward substitution. Formally $\mathbf{y} = L^{-1}\mathbf{b}$. This step takes only $O(n)$ work, rather than the expected $O(n^2)$ amount, since L is bidiagonal.

3. Solve $U\mathbf{x} = \mathbf{y}$ for $\mathbf{x}$ via backward substitution. This step takes only $O(n)$ work, rather than the expected $O(n^2)$ amount, since U is bidiagonal.

In total, the algorithm takes only $O(n) + O(n) + O(n) = 3O(n) = O(n)$ work. The next lecture will consider LU factorization for a general nonsingular matrix (essentially Gaussian elimination). In that context, the analog of the above steps will cost $O(n^3)$ work. In part, our discussion here is a warm–up for studying general linear systems and LU factorization. Before describing the above algorithm in more detail, we give a particular example arising from ODE theory.

Example 1
Consider the following two–point boundary value problem on $[a, b]$:

$$v''(x) = g(x), \qquad v(a) = \alpha, \quad v(b) = \beta,$$

where α, β are constants and $g(x)$ is a prescribed function. Our task is to solve for the function $v(x)$. To do this numerically, we introduce a uniformly spaced grid $x_k = a + (k-1)(b-a)/(n-1)$, for $k = 1, 2, \cdots, n$. Then, with $v_k = v(x_k)$, we may use

$$v''(x_k) \simeq \frac{\dfrac{v_{k+1} - v_k}{h} - \dfrac{v_k - v_{k-1}}{h}}{h} = (v_{k+1} + v_{k-1} - 2v_k)/h^2 \tag{10}$$

as an approximation to the second derivative. This approximation yields

$$v_{k+1} + v_{k-1} - 2v_k = h^2 g_k \tag{11}$$

as an approximation to the ODE itself, an equation which makes sense for $k = 2, \cdots, n-1$. Now, we know $v_1 = \alpha$ and $v_n = \beta$, so the linear system corresponding to our ODE boundary value problem is

$$\begin{pmatrix} 1 & 0 & & & & & \\ 1 & -2 & 1 & & & & \\ & 1 & -2 & 1 & & & \\ & & 1 & -2 & 1 & & \\ & & & \ddots & \ddots & \ddots & \\ & & & & 1 & -2 & 1 \\ & & & & & 1 & -2 & 1 \\ & & & & & & 0 & 1 \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \\ \vdots \\ v_{n-2} \\ v_{n-1} \\ v_n \end{pmatrix} = \begin{pmatrix} \alpha \\ h^2 g_2 \\ h^2 g_3 \\ h^2 g_4 \\ \vdots \\ h^2 g_{n-2} \\ h^2 g_{n-1} \\ \beta \end{pmatrix}. \tag{12}$$

Actually, the variables v_1 and v_n are easily eliminated to produce a smaller $(n-2)$–by–$(n-2)$ system for the remaining variables. Nevertheless, notice that the matrix on the lefthand is tridiagonal (as it would be also for the reduced system), that is entries not located on the diagonal, superdiagonal, or subdiagonal are zero.

We express the general tridiagonal system $T\mathbf{x} = \mathbf{b}$ as

$$\begin{pmatrix} d_1 & f_1 & & & \\ e_2 & d_2 & f_2 & & \\ & e_3 & d_3 & f_3 & \\ & & \ddots & \ddots & \ddots \\ & & e_{n-1} & d_{n-1} & f_{n-1} \\ & & & e_n & d_n \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n-1} \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_{n-1} \\ b_n \end{pmatrix}. \tag{13}$$

The LU factorization of T takes the following special form:

$$\begin{pmatrix} d_1 & f_1 & & & \\ e_2 & d_2 & f_2 & & \\ & e_3 & d_3 & f_3 & \\ & & \ddots & \ddots & \ddots \\ & & e_{n-1} & d_{n-1} & f_{n-1} \\ & & & e_n & d_n \end{pmatrix} =$$

$$\begin{pmatrix} 1 & 0 & & & \\ \ell_2 & 1 & 0 & & \\ & \ell_3 & 1 & 0 & \\ & & \ddots & \ddots & \ddots \\ & & \ell_{n-1} & 1 & 0 \\ & & & \ell_n & 1 \end{pmatrix} \begin{pmatrix} u_1 & f_1 & & & \\ 0 & u_2 & f_2 & & \\ & 0 & u_3 & f_3 & \\ & & \ddots & \ddots & \ddots \\ & & 0 & u_{n-1} & f_{n-1} \\ & & & 0 & u_n \end{pmatrix}. \tag{14}$$

Notice that here both L and U are bidiagonal. To show that such a factorization is possible, we simply perform the LU matrix multiplication and compare sides, finding

$$\begin{pmatrix} d_1 & f_1 & & & & \\ e_2 & d_2 & f_2 & & & \\ & e_3 & d_3 & f_3 & & \\ & & \ddots & \ddots & \ddots & \\ & & & e_{n-1} & d_{n-1} & f_{n-1} \\ & & & & e_n & d_n \end{pmatrix} =$$

$$\begin{pmatrix} u_1 & f_1 & & & & \\ \ell_2 u_1 & \ell_2 f_1 + u_2 & f_2 & & & \\ & \ell_3 u_2 & \ell_3 f_2 + u_3 & f_3 & & \\ & & \ddots & \ddots & \ddots & \\ & & & \ell_{n-1}u_{n-2} & \ell_{n-1}f_{n-2} + u_{n-1} & f_{n-1} \\ & & & & \ell_n u_{n-1} & \ell_n f_{n-1} + u_n \end{pmatrix}. \tag{15}$$

Therefore, we have the equations

$$\begin{aligned} d_1 &= u_1 \\ e_2 &= \ell_2 u_1 \\ d_2 &= \ell_2 f_1 + u_2 \\ &\vdots \\ e_k &= \ell_k u_{k-1} \\ d_k &= \ell_k f_{k-1} + u_k \\ &\vdots \\ e_n &= \ell_n u_{n-1} \\ d_n &= \ell_n f_{n-1} + u_n \end{aligned}$$

Van Loan's implementation to solve these equations is the following.

```
   function [l,u] = TriDiLU(d,e,f)
%           [l,u] = TriDiLU(d,e,f)
%  Tridiagonal LU without pivoting, where d,e,f are n-vectors. Assumes
%  T = diag(e(2:n),-1) + diag(d) + diag(f(1:n-1),1) has an LU
%  factorization. Output l and u are n-vectors with the property that
%  if L = eye + diag(l(2:n),-1) and U = diag(u) + diag(f(1:n-1),-1),
%  then T = LU. Note l(1) and f(n) are wasted.
n = length(d); l = zeros(n,1); u = zeros(n,1);
u(1) = d(1);
for i = 2:n
   l(i) = e(i)/u(i-1);
   u(i) = d(i) - l(i)*f(i-1);
end
```

Notice the 3 flops in the `for` loop (one division, one subtraction, and one multiplication). Therefore, as written the algorithm takes $3(n-1) = O(n)$ flops to produce the vectors `l` and `u`. Notice also that one entry in each of the vectors `l`, `e`, `f` is wasted (this allows for indexing which is an exact rendering of our theoretical description). Furthermore, notice that the algorithm will **fail** if in the step `l(i) = e(i)/u(i-1)` the `u(i-1)` is zero, in which case the algorithm can't be carried through. We talk about how to fix this later, but note now that **failure** is possible (apologies to Gene Kranz).

Once we have produced the LU factorization, that is have all the numbers ℓ_k and u_k (and of course the f_k which we have for free), then we can get on with the remaining steps **1** and **2** to solve the system. These steps are achieved with the routines `LBiDiSol` and `UBiDiSol`, given on pages 217 and 218 of Van Loan's textbook. Here they are (stripped of their comments headers).

```
   function y = LBiDiSol(l,b)
%  function y = LBiDiSol(l,b)
%  Note we use y rather than Van Loan's x. We are solving Ly = b.
%  L is *unit* lower bidiagonal.
n = length(b); y = zeros(n,1);
y(1)= b(1);
for i = 2:n
   y(i) = b(i) - l(i)*y(i-1);
end
```

This algorithm is easily seen to take $O(n)$ work (there are 2 flops in the `for` loop).

```
   function x = UBiDiSol(u,f,y)
%  function x = UBiDiSol(u,f,y)
%  Note we use y for Van Loan's b. We are solving Ux = y.
%  U is upper bidiagonal.
n = length(y); x = zeros(n,1);
x(n)= y(n)/u(n);
for i = n-1:-1:1
   x(i) = (y(i) - f(i)*x(i+1))/u(i);
end
```

Again this algorithm is $O(n)$.

Here is an example of how these Matlab functions may be used to solve a real problem. We return to the ODE boundary value described in the example above. We set $g(x) = -\sin(x)$, $a = 0$, $b = \pi$, $\alpha = 0$, $\beta = 0$. Then the exact solution is $v(x) = \sin x$ (check it!). In Matlab we then run the following script.

```
%
% x array and spacing h.
%
n = 40; x = linspace(0,pi,n)'; h = pi/(n-1);
%
% righthand side in Eq. (13).
%
b = -h^2*sin(x); b(1) = 0; b(n) = 0;
%
%  Tridiagonal matrix (only form diagonal, super and sub diagonals).
%
d = -2*ones(n,1); e = ones(n,1); f = ones(n,1);
d(1) = 1; f(1) = 0; d(n) = 1; e(n) = 0;
%
% Solve the system.
%
[l, u] = TriDiLU(d,e,f); y = LBiDiSol(l,b); v = UBiDiSol(u,f,y);
%
%  Plot the error.
%
explot(x,v-sin(x)); axis tight;
xlabel('x'); ylabel('v_k - sin(x_k)'); title('Numerical error')
```

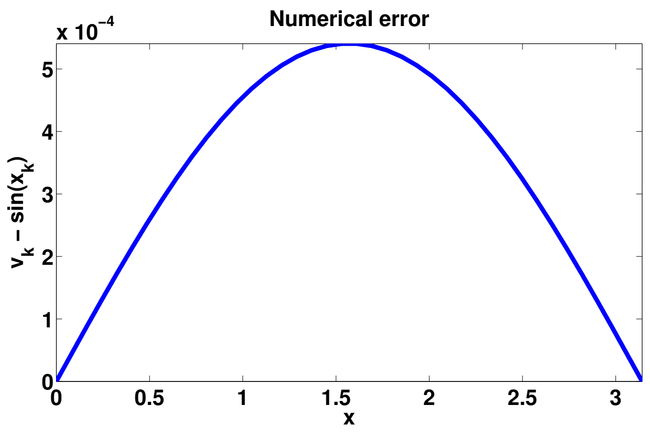

Figure 1: Error between the numerical solution v_k and exact solution $\sin(x_k)$.

The output from the script is shown in Fig. 1.