A tridiagonal linear system has three nonzero “stripes”.

Tridiagonal Solution

Efficiently store and solve a tridiagonal system of linear equations.

1 A tridiagonal linear system

A linear system $Ax = b$ is called tridiagonal if, in the $i$-th equation, only the coefficients $a_{i,i-1}$, $a_{i,i}$ and $a_{i,i+1}$ are nonzero. The first and last equations will actually only have two nonzero coefficients. The name comes from the fact that, if we display the matrix, the nonzero entries fall along three diagonals:

\[
\begin{bmatrix}
    a_{1,1} & a_{1,2} & & \\
    a_{2,1} & a_{2,2} & a_{2,3} & \\
    a_{3,2} & a_{3,3} & a_{3,4} & \\
    & \cdots & a_{n-1,n-2} & a_{n-1,n-1} & a_{n-1,n} \\
    & & \cdots & a_{n,n-1} & a_{n,n}
\end{bmatrix}
\]

This special form of a linear system is of interest to us for two reasons:

- the nonzero matrix entries can be stored using $3n$ space rather than $n^2$;
- the linear system can be solved using $O(n)$ operations rather than $O(n^3)$;

Thus, if we recognize when we are dealing with a tridiagonal system, we can greatly reduce the necessary storage and computational effort needed to obtain a solution.

A tridiagonal linear system is one of the simplest examples of a sparse matrix. We will be able to use this approach to solve the linear system associated with a discretized version of the one-dimensional heat equation.
2 How to solve a tridiagonal system

Consider the following tridiagonal linear system:
\[
\begin{align*}
3x_1 + 4x_2 &= 11 \\
6x_1 + 10x_2 + x_3 &= 29 \\
10x_2 + 9x_3 &= 47
\end{align*}
\]

Of course, Gauss elimination will work on this problem, but we should notice that some of our work has been done for us already. In column 1, there is already a zero in the (3,1) position. For a larger tridiagonal system, we would see a much larger savings, since all but one of the subdiagonal entries are already zero.

To solve this system, we begin by noticing that if we subtract 2 times equation #1 from equation #2, we eliminate \(x_1\) from equation #2:
\[
\begin{align*}
3x_1 + 4x_2 &= 11 \\
2x_2 + x_3 &= 7 \\
10x_2 + 9x_3 &= 47
\end{align*}
\]

Now subtract 5 times equation #2 from equation #3, to eliminate \(x_2\) from equation #3:
\[
\begin{align*}
3x_1 + 4x_2 &= 11 \\
2x_2 + x_3 &= 7 \\
4x_3 &= 12
\end{align*}
\]

The original tridiagonal equations now have a format known as upper triangular. Upper triangular systems are easy to solve. Now we do the back solve step.

\[
\begin{align*}
x_3 &= 12/4 \\
x_2 &= (7 - 1x_3)/2 \\
x_1 &= (11 - 4x_2)/3
\end{align*}
\]

Notice that the forward elimination required just \(n - 1\) simple operations, and the backward substitution required \(n\) slightly more complicated operations.

Except for special cases where we encounter a zero pivot, any tridiagonal linear system can be solved this way. We sweep down the equations, eliminating variable \(i\) from equation \(i + 1\). Then we sweep upwards, solving for variable \(n\), then \(n - 1\), ..., until we reach variable 1, and the system has been solved.

3 Tridiagonal solution algorithm

Here is an algorithm to solve \(Ax = b\) when \(A\) is tridiagonal:

Algorithm 1 Tridiagonal linear system solver

\[
\begin{algorithm}
\begin{algorithmic}
\State \textbf{for} \ 1 \leq j \leq n-1 \ \textbf{do}
\State \textbf{if} \ A_{j,j} = 0.0 \ \textbf{then}
\State \quad \text{terminate, the algorithm has failed.}
\State \textbf{end if}
\State \quad \ s \gets A_{j+1,j} / A_{j,j}.
\State \quad \ A_{j+1,j+1} \gets A_{j+1,j+1} - s A_{j,j+1}.
\State \quad \ b_{j+1} \gets b_{j+1} - s b_j.
\State \textbf{end for}
\State \quad x_n \gets b_n / A_{n,n}
\State \textbf{for} \ n-1 \geq j \geq 1 \ \textbf{do}
\State \quad x_j \gets (b_j - A(j,j+1) x_{j+1}) / A_{j,j}
\State \textbf{end for}
\end{algorithmic}
\end{algorithm}
\]

2
Look at this algorithm, and consider how we are justified in saying that the solution of a tridiagonal system requires $O(n)$ arithmetic operations.

The reason for this reduction in work is not just that a tridiagonal matrix has a special structure, but that our algorithm realizes this structure, and takes advantage of it by only doing the work that it knows has to be done.

4 MATLAB tridiagonal solver

```
function x = tridiag_solve ( A, b )
n = size ( A, 1 );
for j = 1 : n - 1
  s = A(j+1,j) / A(j,j);
  A(j+1,j+1) = A(j+1,j+1) - s * A(j,j+1);
  b(j+1) = b(j+1) - s * b(j);
end
x = zeros ( n, 1 );
x(n) = b(n) / A(n,n)
for j = n - 1 : -1 : 1
  x(j) = ( b(j) - A(j,j+1) * x(j+1) ) / A(j,j);
end
return
end
```

Listing 1: tridiag_solve.m

5 A compact tridiagonal storage scheme

Since a tridiagonal matrix only has $3n - 2$ nonzero entries, it may be worth abandoning the full $n \times n$ storage scheme we are used to. Instead, we can create 3 vectors, $a, b, c$, each of length $n$, and then, for each row $i$, storing $a_i = A_{i,i-1}$, $b_i = A_{i,i}$, $c_i = A_{i,i+1}$. We will leave $a_1$ and $c_n$ to zero, since they don’t correspond to legal entries of $A$.

When we write our solution algorithm, we just have to remember to replace each mention of an entry of $A$ by the corresponding element of $a$, $b$ or $c$. Also, the right hand side will get renamed to $f$!

$$
\begin{align*}
3x_1 &+ 4x_2 &= 11 & a &0 &3 &4 \\
6x_1 &+ 10x_2 &+ 1x_3 &= 29 &\rightarrow &6 &10 &1 \\
10x_2 &+ 9x_3 &= 47 & &10 &9 &0
\end{align*}
$$

If we are very careful, we can figure out how to replace references to the full matrix $A$ by references to the subdiagonal, diagonal, and superdiagonal vectors $a, b, c$. This will allow us to create a new function `tridiag_sparse_solve()` which carries out Gauss elimination on the sparse version of our linear system.
6 MATLAB sparse tridiagonal solver

We can use the following table of replacements to create a sparse storage version of our solver:

\[
\begin{align*}
A_{j,j-1} &\rightarrow a_j \\
A_{j,j} &\rightarrow b_j \\
A_{j,j+1} &\rightarrow c_j \\
A_{j+1,j} &\rightarrow a_{j+1} \\
A_{j+1,j+1} &\rightarrow b_{j+1} \\
A_{j+1,j+2} &\rightarrow c_{j+1}
\end{align*}
\]

```matlab
function x = tridiag_sparse_solve ( a, b, c, f )

n = length ( a ) ;

for j = 1 : n - 1
    s = a(j+1) / b(j) ;
    b(j+1) = b(j+1) - s * c(j) ;
    f(j+1) = f(j+1) - s * f(j) ;
end

x = zeros ( n, 1 ) ;

x(n) = f(n) / b(n) ;

for j = n - 1 : -1 : 1
    x(j) = ( f(j) - c(j) * x(j+1) ) / b(j) ;
end

return
end
```

Listing 2: tridiag_sparse_solve.m

This tridiagonal solver now takes advantage of both reduced space requirements, \(O(3n)\) and reduced work requirements \(O(n)\).

7 Exercise: a tridiagonal matrix associated with splines

If we have a grid of \(n\) points \(x\), then we can define an \(n-1\) vector \(g\) such that \(g_i = x_{i+1} - x_i\). These spacings can be used to define an \(n \times n\) tridiagonal matrix associated with interpolation by cubic splines. The functions \(A=spline\_matrix(g)\) and \([a,b,c]=spline\_sparse\_matrix(g)\) return dense and sparse versions of this matrix.

In order to manufacture a problem, we choose a simple solution vector \(x\) and compute \(f = A \ast x\). However, for the sparse matrix version, we can call \(f=tridiag\_sparse\_mv(a,b,c,x)\) to compute this product.

Finally, to solve the linear system, we can call \(x=tridiag\_solve(A,f)\) or \(x=tridiag\_sparse\_solve(a,b,c,f)\).

1. Decide on a size \(n\)
2. Set the vector \(g\) to \(n - 1\) random positive values.
3. Compute the spline matrix \(A\) or spline sparse matrix \([a,b,c]\).
4. Decide on a solution vector \(x1\).
5. Compute the right hand side \(f\) by multiplying the matrix times \(x1\).
6. Call tridiagonal solver or tridiagonal sparse solver to recover \(x2\).
7. Verify \(x1\) and \(x2\) are equal ... actually, that \(||x1 - x2||\) is small.
8 Challenge: Computing a tridiagonal inverse

Starting with one of the tridiagonal solver codes, make a new copy called \( X = \text{tridiag-inverse}(A) \) or \( X = \text{tridiag_sparse_inverse}(a, b, c) \). Instead of inputting a right hand side vector \( f \), set up a dense \( n \times n \) matrix \( F \), which is initialized to the identity matrix. Now solve for a dense \( n \times n \) matrix \( X \), which will be the inverse of the tridiagonal matrix.

Wherever the original code referenced the vectors \( f(j) \) or \( x(j) \), you will need to replace this by a reference to an entire row, \( F(j,:) \) or \( X(j,:) \). This should be enough to get your code to work.

Use your code to compute the inverse of the spline matrix \( A \) from the previous example.

It would be nice if the inverse of a tridiagonal matrix was also a tridiagonal matrix. It isn’t!

Verify that your output matrix \( X \) is the correct inverse of \( A \) by computing \( A \times X \). If you worked with the sparse form, compute the product by \( AX = \text{tridiag_sparse_mv}(a, b, c, X) \).