St. Francis Xavier University
Department of Computer Science

CSCI 554: Matrix Computation
Lecture 2: Systems of Linear Equations
Winter 2022

1 Solving Systems of Equations

In our previous lecture, we reviewed a number of methods of multiplying matrices. Our first algorithm—the algorithm to multiply a matrix by a vector—allowed us to compute $Ax = b$ where we're given the matrix A and the vector x.

In most applications, however, we're not given the vector x. Instead, we're given the matrix A and the vector b, and we must find x. Algebraically speaking, we have to determine what linear combination of values from A sum to the corresponding value in b. If A is of dimension $n \times n$ and b is of size n, then we can frame the problem as the following *system of n linear equations* in n unknowns:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n. \end{aligned}$$

Here, we are given each of the terms a_{ij} and b_k, and we must solve for all of the terms x_ℓ. For our purposes, we can assume that all of these terms are real numbers. With a small amount of thought, we can see that this system of linear equations is equivalent to the matrix-vector multiplication $Ax = b$, or

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & & a_{2n} \\ \vdots & & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}.$$

If you've taken a course in linear algebra in the past, then you've doubtless heard of systems of linear equations before, and you've likely solved a fair few by hand using techniques such as back substitution, forward substitution, and Gaussian elimination. You may also recall the following result about systems of linear equations:

Proposition 1. *Let $Ax = b$ be a system of n linear equations in n unknowns. The system of linear equations has exactly one solution if and only if the matrix A is invertible; namely, the solution is $x = A^{-1}b$.*

Recall that an $n \times n$ matrix A is *invertible* (or *nonsingular*) if there exists an $n \times n$ matrix B such that $AB = BA = I_n$, where I_n is the $n \times n$ identity matrix

$$I_n = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & & 0 \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix}.$$

The identity matrix is the matrix containing 1 as the entries along its main diagonal and 0 as every other entry. It is the unique matrix with the property that $AI_n = I_nA = A$ for all $n \times n$ matrices A.

Thus, we have a necessary and sufficient condition to determine whether a system of linear equations has a unique solution! All we need to do is compute the inverse of the matrix A. There remains one problem, however: not all matrices have inverses. In the case where A does not have an inverse, the corresponding system of linear equations $Ax = b$ will have either no solution or infinitely many solutions.

So, how do we overcome this problem? The most straightforward solution is simply to not use inverse matrices in our computations. That way, we don't need to worry about receiving as input a matrix A that is noninvertible. Besides, it's often computationally expensive to compute inverses if they exist.

The alternative, then, has us solving the system of linear equations directly, using the same techniques we learned back when we were solving systems by hand in our linear algebra classes. Fortunately, it's quite straightforward to implement these methods algorithmically, and computers are quite a bit faster and more accurate than we are when it comes to computing the solution.

2 Special Systems

You may recall from your linear algebra classes that, before solving a system of linear equations, we perform row reduction on the matrix via elementary row operations so that it is in row echelon form. In doing so, the matrix takes on a special triangular form, and our computation by hand is simplified.

When it comes to solving systems of linear equations on a computer, we will again focus on these special matrix forms. We will assume that the input matrix is already of a particular special form (and, if not, then it's relatively painless to convert a general matrix to a special form). We will begin by studying the familiar triangular matrices before moving on to positive definite matrices and sparse matrices.

2.1 Triangular Systems

An $n \times n$ square matrix A is called a *triangular matrix* if all entries either above or below the main diagonal are zero. In this way, the entries either above or below the main diagonal form a "triangle" of zeroes.

If the entries above the main diagonal are zero (i.e., if $a_{ij} = 0$ when $i < j$), then we say that A is *lower triangular*. Similarly, if the entries below the main diagonal are zero (i.e., if $a_{ij} = 0$ when $i > j$), then we say that A is *upper triangular*. An example of a lower triangular matrix is presented at left, while an example of an upper triangular matrix is presented at right. Note that, in either case, an entry a_{ij} may or may not be zero; the only entries guaranteed to be zero are in the upper or lower "triangle".

$$A_L = \begin{bmatrix} a_{11} & 0 & 0 & \cdots & 0 \\ a_{21} & a_{22} & 0 & & 0 \\ a_{31} & a_{32} & a_{33} & & 0 \\ \vdots & & & \ddots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} \end{bmatrix} \qquad A_U = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ 0 & a_{22} & a_{23} & & a_{2n} \\ 0 & 0 & a_{33} & & a_{3n} \\ \vdots & & & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & a_{nn} \end{bmatrix}$$

If we restrict ourselves to considering triangular matrices, then we can make use of a necessary and sufficient condition for invertibility that doesn't involve finding the inverse.

Proposition 2. *Let A be an $n \times n$ triangular matrix. Then A is invertible if and only if $a_{ii} \neq 0$ for all $1 \leq i \leq n$.*

Let's now focus in particular on lower triangular and upper triangular matrices.

2.1.1 Lower Triangular Systems

Given an invertible lower triangular matrix

$$A = \begin{bmatrix} a_{11} & 0 & 0 & \cdots & 0 \\ a_{21} & a_{22} & 0 & & 0 \\ a_{31} & a_{32} & a_{33} & & 0 \\ \vdots & & & \ddots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} \end{bmatrix},$$

we can model the equation $Ax = b$ by the following system of linear equations:

$$\begin{aligned} a_{11}x_1 &= b_1 \\ a_{21}x_1 + a_{22}x_2 &= b_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 &= b_3 \\ &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + a_{n3}x_3 + \cdots + a_{nn}x_n &= b_n \end{aligned}$$

Observe that our job is made quite easy at first: from the first linear equation, $a_{11}x_1 = b_1$, we can calculate $x_1 = b_1/a_{11}$. Now that we have the value x_1, we can substitute it into the second linear equation and calculate $x_2 = (b_2 - a_{21}x_1)/a_{22}$. Proceeding in this way for all of the other linear equations, we can calculate every value x_i in order: the ith entry of x is equal to

$$x_i = \frac{b_i - \left(\sum_{j=1}^{i-1} a_{ij}x_j\right)}{a_{ii}}.$$

Remark. Our assumption that A is invertible is crucial here, as this ensures each diagonal entry is nonzero by Proposition 2. Thus, we can perform the division in each step without accidentally dividing by zero.

This method is known as *forward substitution*, as we're calculating values of x_i and carrying them forward as we substitute each value into the next linear equation. Expressing the forward substitution method as an algorithm isn't too difficult; in fact, we did most of the hard work just in writing the general formula for x_i earlier!

Altogether, we have the following algorithm to perform forward substitution.

Algorithm 1: Forward substitution

```
for 1 ≤ i ≤ n do
    x_i ← b_i
    for 1 ≤ j ≤ (i − 1) do                ▷ this loop is not executed when i = 1
        x_i ← x_i − a_ij x_j
    if a_ii = 0 then
        return error
    else
        x_i ← x_i / a_ii
return x
```

One efficiency observation we can make regards the values we need to use at each iteration of the algorithm. Strictly speaking, we don't need to allocate space for both the vector b and the vector x; we could just reuse the space allocated for b to store and return x. This is because we only use the value b_i on the ith iteration of the algorithm. Thus, once we're done with the value b_i, we can overwrite it with our calculated value x_i. Then, once our algorithm halts, we simply return b (which should more accurately now be called "x").

Now, how efficient is our forward substitution algorithm? Let's count the flops to find out. Within the inner for loop, we have two flops: one for subtraction and one for multiplication. We iterate through this inner for loop a total of $(i-1)$ times on the ith iteration of the outer for loop, and since we iterate through the outer for loop n times, the inner for loop is therefore executed a total of $(0+1+2+\cdots+n-1) = \sum_{i=1}^{n}(i-1)$ times. Thus, the total number of flops across both of the for loops is $2 \cdot \sum_{i=1}^{n}(i-1) = n(n-1) \in O(n^2)$.

Note that we focused on the nested for loops in our analysis, even though there are other flops performed outside of the for loops. Ultimately, these flops don't affect our analysis, since the number of times we execute those steps grows proportional to n instead of n^2. Thus, our algorithm still has a runtime proportional to $O(n^2)$ as n grows large.

2.1.2 Upper Triangular Systems

As you might now expect following the previous section, given an invertible upper triangular matrix

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ 0 & a_{22} & a_{23} & & a_{2n} \\ 0 & 0 & a_{33} & & a_{3n} \\ \vdots & & & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & a_{nn} \end{bmatrix},$$

we can model the equation $Ax = b$ by the following system of linear equations:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \cdots + a_{1n}x_n &= b_1 \\ a_{22}x_2 + a_{23}x_3 + \cdots + a_{2n}x_n &= b_2 \\ a_{33}x_3 + \cdots + a_{3n}x_n &= b_3 \\ &\vdots \\ a_{nn}x_n &= b_n \end{aligned}$$

This time, we can calculate the value of x_n directly by computing $x_n = b_n/a_{nn}$, and we can then substitute that value of x_n into the linear equation for x_{n-1}, and so on. In this way, we're carrying values of x_i backward as we substitute each value into the previous linear equation, and this gives rise to the method of *backward substitution*.

Algorithm 2: Backward substitution

```
for n ≥ i ≥ 1 do                          ▷ the outer for loop decrements the value of i
    x_i ← b_i
    for (i + 1) ≤ j ≤ n do                ▷ this loop is not executed when i = n
        x_i ← x_i − a_ij x_j
    if a_ii = 0 then
        return error
    else
        x_i ← x_i / a_ii
return x
```

Comparing Algorithm 1 for forward substitution with Algorithm 2 for backward substitution, we see that the only differences are in the bounds of both for loops. Ultimately, we perform the same number of iterations in Algorithm 2 as we do in Algorithm 1, and so our earlier analysis can be reused to show that the backward substitution procedure has a runtime of $O(n^2)$.

2.2 Positive Definite Systems

We now move on to discussing another kind of special matrix called a *positive definite* matrix. In this and the following section, our focus will be on positive definite matrices and their associated systems, but our overall goal will be much the same as in the previous section: solving some linear system of equations.

Before we define what a positive definite matrix is, we require a bit of background knowledge. Given an $m \times n$ matrix A, we say that the *transpose* of A, denoted $A^\intercal$, is the $n \times m$ matrix obtained by "flipping" A along its main diagonal. That is, the entry a_{ij} at index (i, j) of A would be located at index (j, i) in $A^\intercal$.

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & & a_{2n} \\ \vdots & & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \qquad A^\intercal = \begin{bmatrix} a_{11} & a_{21} & \cdots & a_{m1} \\ a_{12} & a_{22} & & a_{m2} \\ \vdots & & \ddots & \vdots \\ a_{1n} & a_{2n} & \cdots & a_{mn} \end{bmatrix}$$

If some matrix A has the property that $A = A^\intercal$, then we say that A is a *symmetric* matrix.

We can take transposes of vectors in exactly the same way as we do for matrices. If we're given a column vector x of size n (i.e., of dimension $n \times 1$), then the transpose of x would be a row vector also of size n (i.e., of dimension $1 \times n$).

With all of these notions, we can define the main topic of this section: positive definiteness.

Definition 3 (Positive definiteness)**.** Let A be an $n \times n$ symmetric matrix with real number entries. If, for all nonzero size-n column vectors x, A satisfies the property

$$x^\intercal A x > 0,$$

then we say that A is a positive definite matrix.

How can we compare a vector-matrix-vector multiplication on the left-hand side to a single value on the right-hand side? Through an analysis of the dimensions of $x^\intercal$, A, and x, we can see that the product on the left-hand side will result in a 1×1 matrix; that is, a single value. Thus, we can use an inequality to compare both sides.

Example 4. The matrix

$$A = \begin{bmatrix} 2 & -1 & 0 \\ -1 & 2 & 1 \\ 0 & -1 & 2 \end{bmatrix}$$

is an example of a positive definite matrix. To verify this, consider the result you would obtain by computing the product $x^\intercal A x$ given an arbitrary vector

$$x = \begin{bmatrix} a \\ b \\ c \end{bmatrix}.$$

Omitting the technical calculations, we would end up with $x^\intercal A x = a^2 + (a - b)^2 + (b - c)^2 + c^2$. Since this is a sum of squares, and since x is a nonzero vector, this evaluates to a nonnegative value.

Positive definite matrices have some rather nice properties which lend themselves well to efficient numerical algorithms, and we'll see positive definite matrices reappear in later lectures. Going beyond matrix computations, positive definite matrices also play a large role in the study of optimization problems and approximation algorithms.

Before we continue, let's establish a few properties of positive definite matrices.

Theorem 5. *Let A be a matrix. If A is positive definite, then it is also invertible.*

Proof. Suppose that A were not invertible. Then, by a property of invertible matrices, there would exist some nonzero size-n column vector x such that $Ax = 0$. However, this would mean that $x^\intercal A x = 0$, and so A would not be positive definite. □

Combining this result with Proposition 1, we get the following corollary:

Corollary 6. *Let A be a matrix. If A is positive definite, then the associated system of linear equations $Ax = b$ has exactly one solution.*

Proof. By Theorem 5, if A is positive definite, then it is invertible. By Proposition 1, we know that if a matrix is invertible, then its associated system of linear equations has exactly one solution. □

Thus, if we have a positive definite matrix, then we can immediately conclude that the associated system of linear equations has a unique solution! This gives us another way of establishing this nice fact without having to rely on invertibility. One question remains, though: how do we get a positive definite matrix? Well, meandering in a bit of a circle, we would be able to construct a positive definite matrix easily if we start with an invertible matrix.

Theorem 7. *Let A be an $n \times n$ invertible matrix, and let $B = A^\intercal A$. Then B is positive definite.*

Proof. Before showing that B is positive definite, we must first show that B is symmetric. We require two facts about matrix transposes: $M^{\intercal\intercal} = M$ for all matrices M, and $(MN)^\intercal = N^\intercal M^\intercal$ for all matrices M and N. With these facts, we can conclude that $B^\intercal = (A^\intercal A)^\intercal = A^\intercal A^{\intercal\intercal} = A^\intercal A = B$, and so B is symmetric.

Now, we focus on positive definiteness. For any nonzero size-n column vector x, we have that $x^\intercal Bx = x^\intercal A^\intercal Ax$. Let $y = Ax$. Then $y^\intercal = x^\intercal A^\intercal$, and $x^\intercal Bx = y^\intercal y = \sum_{i=1}^{n} y_i^2$. This sum of squares is nonnegative and nonzero, since A is invertible and x is nonzero. Thus, $x^\intercal Bx > 0$, and B is positive definite. □

Example 8. Consider the invertible matrix

$$A = \begin{bmatrix} 10 & 8 \\ 5 & 4 \end{bmatrix}.$$

Taking the transpose of A and calculating

$$B = A^\intercal A = \begin{bmatrix} 10 & 5 \\ 8 & 4 \end{bmatrix} \begin{bmatrix} 10 & 8 \\ 5 & 4 \end{bmatrix} = \begin{bmatrix} 125 & 100 \\ 100 & 80 \end{bmatrix},$$

we get that B is positive definite by Theorem 7.

So, if we have a positive definite matrix of the form specified in Theorem 7, what can be done with it? A result known as the *Cholesky decomposition theorem*, the main result of this section, tells us that every positive definite matrix has a unique decomposition into the product of a triangular matrix and its transpose.

Theorem 9 (Cholesky decomposition theorem)**.** *Let A be a positive definite matrix. Then there exists a unique decomposition $A = R^\intercal R$ such that R is an upper triangular matrix with all entries r_{ii} along the main diagonal being positive.*

Proof. Omitted. □

We call the triangular matrix R the *Cholesky factor* of A. The proof of the Cholesky decomposition theorem is somewhat lengthy and technical, so we will omit it here. Instead, we will focus on the computational aspect of the theorem and on developing an algorithm to compute the Cholesky factor R.

Why, though, do we care about having an algorithm to perform this decomposition? Well, note that R is a triangular matrix, as is the transpose of R. If we're given a system of linear equations $Ax = b$ where A is a positive definite matrix, having the Cholesky factor available allows us to rewrite the system as $R^\intercal Rx = b$. Then, taking $y = Rx$, we can solve the system $R^\intercal y = b$ using our forward substitution algorithm to obtain y, and we can then solve the system $Rx = y$ using our backward substitution algorithm to obtain x!

Let's develop a method to compute R. Consider the form of the Cholesky decomposition of some matrix A,

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & & a_{2n} \\ \vdots & & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} = \begin{bmatrix} r_{11} & 0 & \cdots & 0 \\ r_{12} & r_{22} & & 0 \\ \vdots & & \ddots & \vdots \\ r_{1n} & r_{2n} & \cdots & r_{nn} \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & \cdots & r_{1n} \\ 0 & r_{22} & & r_{2n} \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \cdots & r_{nn} \end{bmatrix} = R^\intercal R.$$

Assuming we have the first $(i-1)$ rows of R, we can compute the ith row of R as follows. (Note that, if we're computing the first row of R, this method still works; we just don't have any previous rows of R.) Only the first i entries in the ith row of $R^\intercal$ are nonzero, so we know that

$$a_{ij} = r_{1i}r_{1j} + r_{2i}r_{2j} + \cdots + r_{(i-1)i}r_{(i-1)j} + r_{ii}r_{ij}.$$

We can use the entries from the first $(i-1)$ rows of R to fill in every value of this formula except for r_{ii} and r_{ij} in the last term. If we set $i = j$, then

$$a_{ii} = r_{1i}^2 + r_{2i}^2 + \cdots + r_{(i-1)i}^2 + r_{ii}^2,$$

and we can calculate

$$r_{ii} = +\sqrt{a_{ii} - \sum_{k=1}^{i-1} r_{ki}^2}.$$

Plugging this value r_{ii} into our first equation for a_{ij}, we can finally calculate

$$r_{ij} = \left(a_{ij} - \sum_{k=1}^{i-1} r_{ki}r_{kj} \right) / r_{ii}$$

for all $(i+1) \le j \le n$. When $j < i$, we know that $r_{ij} = 0$, so we don't need to calculate those values.

This procedure to compute R is known as *Cholesky's method*, and not only does it prove that such a matrix R exists (by construction), but it also establishes that this R is unique: the entry r_{11} is uniquely determined by the fact that $A = R^\intercal R$ with R being upper triangular, the remaining entries r_{1j} are uniquely determined by a similar observation, and the remaining rows of R are uniquely determined by the previously-calculated rows.

Translating Cholesky's method into pseudocode, we obtain our algorithm.

Algorithm 3: Cholesky's method

```
for 1 ≤ i ≤ n do
    r_ii ← a_ii
    for 1 ≤ k ≤ (i − 1) do                    ▷ this loop is not executed when i = 1
        r_ii ← r_ii − r_ki^2
    if r_ii ≤ 0 then
        return error
    else
        r_ii ← √r_ii
        for (i + 1) ≤ j ≤ n do                ▷ this loop is not executed when i = n
            r_ij ← a_ij
            for 1 ≤ k ≤ (i − 1) do            ▷ this loop is not executed when i = 1
                r_ij ← r_ij − r_ki r_kj
            r_ij ← r_ij / r_ii
return R
```

Now, how does Cholesky's method perform? Let us count the flops. Observe that in each of the for loops involving k, we perform two flops. In the first for loop indexed by k, we perform $\sum_{i=1}^{n} \sum_{k=1}^{i-1} 2 = n(n-1)$

operations, which is $O(n^2)$. In the second for loop indexed by k, we perform a total of $\sum_{i=1}^{n}\sum_{j=(i+1)}^{n}\sum_{k=1}^{i-1} 2$ operations. This is a more complex summation, but we can simplify it to $n^3 - 2\frac{n^3}{3} + O(n^2)$, where the Big-O term hides lower-order terms. Ultimately, this sum is $O(n^3)$. For other blocks of the algorithm, such as those lines that perform divisions, square roots, or general checks, the amount of work done is less than $O(n^3)$. Thus, the overall algorithm has a runtime of $O(n^3)$.

2.3 Sparse and Banded Positive Definite Systems

As is often the case in theory versus application, not all matrices are ideally-suited for methods like the ones we've learned in this lecture. That's not to say that the methods won't work; if we're given a positive definite matrix, for instance, then Cholesky's method is guaranteed to give us a decomposition of that matrix. Rather, "idealness" in this sense pertains to the entries of the matrix.

Each of the examples we've seen involve matrices where all pertinent entries are nonzero. These matrices maximize the use of their allocated space: if the matrix is of dimension $m \times n$ and it contains mn entries, then it's using as much space as its taking up in memory. With certain special matrices, like triangular matrices, a significant portion of the matrix consists of zeroes. Since matrices are inherently rectangular structures (that is, we can't program an actually-triangular matrix into memory), these zero entries only serve as padding and therefore take up memory. Fortunately, there are many ways to compress the representation of a matrix to negate the impact of this padding; for example, we could use a run-length encoding to turn a row of fifty zero entries into four characters: "50 0".

Where we sometimes run into problems, however, is in situations where our method takes an input matrix and produces an output matrix that contains more nonzero entries than the original matrix. This padding is known as *fill-in*, and it quickly becomes a headache when the memory of our computer is constrained in some way.

Example 10. Consider the matrix

$$A = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 2 & 0 & 0 \\ 1 & 0 & 3 & 0 \\ 1 & 0 & 0 & 7 \end{bmatrix}.$$

The matrix A contains three zero entries above the diagonal. However, computing the Cholesky factor of A, we get the matrix

$$R = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & -1 & -1 \\ 0 & 0 & 1 & -2 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

which has its upper-triangular portion completely filled-in. If we were to reuse the matrix A to store R, we would overwrite the three zero entries with nonzero entries, and we would therefore use more memory to store R than we needed to store A. On the other hand, if we stored R separately from A in memory, we would need to allocate space for an entirely new matrix.

A matrix that contains many zero entries is called a *sparse matrix*, because the nonzero data within the matrix is sparsely distributed. Thus, one major consideration we must make when developing methods like the ones we studied in this lecture is how the methods perform on sparse input matrices. Are the output matrices well-behaved, or are we likely to obtain output matrices with a large amount of fill-in?

It is possible to reorder the rows or columns of a sparse matrix to obtain a matrix that minimizes the amount of fill-in after running a given method. However, in a matrix with n rows, say, we would need to consider a total of $n!$ possible row reorderings; the problem therefore quickly becomes intractable even for matrices of moderate size. Fortunately, there exist some advanced algorithms that perform this reordering in a more practical manner, though we won't get into the details of such algorithms here.

One specific type of a sparse matrix that structures its nonzero data nicely is known as a *banded matrix*. In a banded matrix, the nonzero data is found in a *band* surrounding the diagonal of the matrix, and all

entries outside of this band are zeroes. Typically, this band is taken to be quite narrow (so that a matrix with zeroes in the top-right and bottom-left corners is not considered a "banded" matrix).

Formally speaking, if we have an $n \times n$ matrix A, and if there exists a value $s \ll n$ such that the entry a_{ij} is zero whenever $|i-j| > s$, then we say that A is a banded matrix with a *band width* of $2s+1$ or, equivalently, with a *semiband width* of s.

Example 11. The 6×6 matrix

$$B = \begin{bmatrix} 4 & 5 & 0 & 0 & 0 & 0 \\ 1 & 9 & 2 & 0 & 0 & 0 \\ 0 & 3 & 1 & 4 & 0 & 0 \\ 0 & 0 & 2 & 7 & 3 & 0 \\ 0 & 0 & 0 & 6 & 4 & 8 \\ 0 & 0 & 0 & 0 & 2 & 2 \end{bmatrix}$$

is a banded matrix with a band width of 3 and a semiband width of 1. Observe that we can store all of the nonzero entries of B in a smaller 6×3 matrix

$$B' = \begin{bmatrix} 0 & 4 & 5 \\ 1 & 9 & 2 \\ 3 & 1 & 4 \\ 2 & 7 & 3 \\ 6 & 4 & 8 \\ 2 & 2 & 0 \end{bmatrix}.$$

The reason why we highlight banded matrices as a special case of sparse matrices is because, unlike a general sparse matrix, a banded matrix lends itself well to methods that are capable of ignoring the zero entries outside of the band. Thus, we can save on memory by only storing the band of the matrix.

For positive definite matrices, we incur even more savings. Since all positive definite matrices are symmetric, we don't need to store the whole band: we only need to store the semiband! Fortunately, methods designed for positive definite matrices—namely, Cholesky's method—are able to ignore zero entries outside of the band and avoid adding fill-in.

Theorem 12. *Let A be a banded positive definite matrix with a semiband width of s. Then the Cholesky factor R of A also has semiband width s.*

Proof. Omitted. □

Moreover, it's possible to show that if A has a semiband width of s, then Cholesky's method will require about ns^2 flops in total to compute the Cholesky factor. If $s \ll n$, then we incur a huge savings in terms of runtime compared to the original runtime of $O(n^3)$.