LU-and-Inverses

September 7, 2017

1 Whence cometh the L in LU?

Last time, we constructed the LU factorization by what may have seemed like a laborious procedure. Getting U was "easy", it was just Gaussian elimination. But to get L, we first wrote out the individual elimination steps as matrices, then inverted them to move them to the other side, then multiplied them together to get L.

However, it turns out that we can just "read off" L much more simply directly from the pivot-row "multipliers" that we use during elimination steps. To see this, let's first write a Julia function to perform Gaussian elimination (without row swaps!) and print out all of the steps, adapting our function from pset 1:

```
In [1]: # perform Gaussian elimination of A without row swaps, returning U,
        # while printing a message for each elimination step.
        function print_gauss(A)
            m = size(A,1) # number of rows
            U = copy!(similar(A, typeof(inv(A[1,1]))), A)
            for j = 1:m   # loop over m columns
                for i = j+1:m   # loop over rows below the pivot row j
                    # subtract a multiple of the pivot row (j)
                    # from the current row (i) to cancel U[i,j] = U[U+1D62][U+2C7C]:
                    ℓ[U+1D62][U+2C7C] = U[i,j]/U[j,j]
                    println("subtracting $ℓ[U+1D62][U+2C7C] × (row $j) from (row $i)")
                    U[i,:] = U[i,:] - U[j,:] * ℓ[U+1D62][U+2C7C]
                    U[i,j] = 0 # store exact zero to compensate for roundoff errors
                end
            end
            return U
        end
```

```
Out[1]: print_gauss (generic function with 1 method)
```

Now, let's try it on a randomly chosen 5×5 matrix A:

```
In [2]: A = [4  -2  -7  -4  -8
             9  -6  -6  -1  -5
            -2  -9   3  -5   2
             9   7  -9   5  -8
            -1   6  -3   9   6]
```

```
Out[2]: 5×5 Array{Int64,2}:
          4  -2  -7  -4  -8
          9  -6  -6  -1  -5
         -2  -9   3  -5   2
          9   7  -9   5  -8
         -1   6  -3   9   6
```

```
In [3]: print_gauss(A)

subtracting 2.25 × (row 1) from (row 2)
subtracting -0.5 × (row 1) from (row 3)
subtracting 2.25 × (row 1) from (row 4)
subtracting -0.25 × (row 1) from (row 5)
subtracting 6.666666666666667 × (row 2) from (row 3)
subtracting -7.666666666666667 × (row 2) from (row 4)
subtracting -3.6666666666666665 × (row 2) from (row 5)
subtracting -1.2442748091603053 × (row 3) from (row 4)
subtracting -0.4732824427480916 × (row 3) from (row 5)
subtracting 33.495145631067295 × (row 4) from (row 5)

Out[3]: 5×5 Array{Float64,2}:
         4.0  -2.0   -7.0    -4.0         -8.0
         0.0  -1.5    9.75    8.0         13.0
         0.0   0.0  -65.5   -60.3333    -88.6667
         0.0   0.0    0.0     0.262087   -0.659033
         0.0   0.0    0.0     0.0        31.7767
```

In comparison, here is the LU factorization of A from the built-in `lu` function, with row-swaps disabled:

```
In [4]: L, U = lu(A, Val{false})
        U

Out[4]: 5×5 Array{Float64,2}:
         4.0  -2.0   -7.0    -4.0         -8.0
         0.0  -1.5    9.75    8.0         13.0
         0.0   0.0  -65.5   -60.3333    -88.6667
         0.0   0.0    0.0     0.262087   -0.659033
         0.0   0.0    0.0     0.0        31.7767
```

Same U matrix! Now let's look at L:

```
In [5]: L

Out[5]: 5×5 Array{Float64,2}:
          1.0     0.0        0.0         0.0      0.0
          2.25    1.0        0.0         0.0      0.0
         -0.5     6.66667    1.0         0.0      0.0
          2.25   -7.66667   -1.24427     1.0      0.0
         -0.25   -3.66667   -0.473282   33.4951   1.0
```

Notice that the entries of L below the diagonal are *exactly* the multipliers that were printed out during Gaussian elimination (the factors by which the pivot row is multiplied before it is *subtracted* from a row below it).

One way to see this is to consider the matrix product LU, which should give A. Consider, for example, the third row of this: the third row of L tells us what linear combinations of the rows of U gives the third row of A. It says:

- third row of `A` = `[-2,-9,3,-5,2]` = `-0.5` × (row 1 of U) + 6.66... × (row 2 of U) + (row 3 of U)

```
In [6]: L[3,1] * U[1,:] + L[3,2] * U[2,:] + U[3,:]

Out[6]: 5-element Array{Float64,1}:
         -2.0
         -9.0
          3.0
         -5.0
          2.0
```

But this is exactly the *reverse of the elimination steps*, so of course it works. **Putting the multipliers in L is the right thing!**

See section 2.6 of the textbook for more info.

Still, computing the L in the LU factorization requires care to put all of the multipliers in the right place with the right sign. It is a pain for human beings, which is why we typically don't do it when performing Gaussian elimination by hand. However, computers are great at this kind of tedious bookkeeping, and since keeping track of L requires almost *no extra work*, computers essentially *always* figure out *both* L and U when doing Gaussian elimination.

2 Using LU factorizations

Lots of things that you might want to do with a matrix A become easier once you have the $A = LU$ factorization. Most importantly, it becomes much easier to solve systems of equations.

(Exactly *how much* easier is something we'll quantify later. Short answer: for an $n \times n$ matrix A, it takes around n^3 operations to perform Gaussian elimination to get U and L, but subsequently solving for x by takes only around n^2 operations.)

2.1 Solving Ax=b

When we do Gaussian elimination by hand, we convert $Ax = b$ to $Ux = c$ by performing the same elimination steps on b to get c as we performed on A to get U. Often, we do this by "augmenting" the matrix A with the right-hand side b. This makes it easier (*for hand calculation*) to keep track of what operations to do on b. For example:

```
In [7]: b = rand(-9:9, 5)
```

```
Out[7]: 5-element Array{Int64,1}:
         -7
          2
          4
         -4
         -7
```

```
In [8]: _, U_and_c = lu([A b], Val{false}) # eliminate augmented matrix (without row swaps)
        U = UpperTriangular(U_and_c[:, 1:end-1]) # all but last column is U
        c = U_and_c[:, end] # last column is c
```

```
Out[8]: 5-element Array{Float64,1}:
           -7.0
           17.75
         -117.833
            1.21628
          -40.1748
```

Then we can solve $Ux = c$ by backsubstitution (`U \ c`), and it should give the same answer (up to roundoff error) as `A \ b`:

```
In [9]: [U\c  A\b] # print them side by side
```

```
Out[9]: 5×2 Array{Float64,2}:
          0.505958   0.505958
         -0.928506  -0.928506
          2.16407    2.16407
          1.46166    1.46166
         -1.26428   -1.26428
```

However, the computer doesn't do this: on a computer, you **almost never augment the matrix with the right-hand-side**. Instead, you:

1. Factor $A = LU$ by Gaussian elimination (not including row swaps, discussed below!), giving $Ax = b \implies LUx = L(Ux) = b$
2. Let $c = Ux$. Solve $Lc = b$ for c by forward-substitution.
3. Solve $Ux = c$ for x by backsubstitution.

The key point to realize is that solving $Lc = b$ for c involves *exactly the same elimination steps* as if you had augmented the matrix with b during Gaussian elimination. The bookkeeping is more tedious for a human, but computers are good at bookkeeping, and there turn out to be several practical advantages for computer software to separate solving for LU and solving for c.

```
In [10]: L, U = lu(A, Val{false}) # Gaussian elimination without row swaps
         c = L \ b # solve Lc = b for c
```

```
Out[10]: 5-element Array{Float64,1}:
            -7.0
            17.75
          -117.833
             1.21628
           -40.1748
```

Same c as before!

Let's write a little program to write out the steps of forward-substitution so that we can see that they are indeed the elimination steps from before:

```
In [11]: c = similar(b, Float64)
         for i = 1:length(b)
             print("c[$i] = b[$i]")
             c[i] = b[i]
             for j = 1:i-1
                 print("- $(L[i,j]) * c[$j]")
                 c[i] = c[i] - L[i,j] * c[j]
             end
             println(" = ", c[i])
         end
         c
```

```
c[1] = b[1] = -7.0
c[2] = b[2]- 2.25 * c[1] = 17.75
c[3] = b[3]- -0.5 * c[1]- 6.666666666666666 * c[2] = -117.83333333333333
c[4] = b[4]- 2.25 * c[1]- -7.666666666666666 * c[2]- -1.2442748091603053 * c[3] = 1.2162849872773336
c[5] = b[5]- -0.25 * c[1]- -3.6666666666666665 * c[2]- -0.47328244274809156 * c[3]- 33.495145631067324
```

```
Out[11]: 5-element Array{Float64,1}:
            -7.0
            17.75
          -117.833
             1.21628
           -40.1748
```

In Julia, `A \ b` does this whole process for you implicitly.

2.2 Multiple right-hand sides and AX = B

Suppose that we need to solve $Ax = b$ for **multiple right-hand sides** b_1, b_2, and so on. Once we have computed $A = LU$ by Gaussian elimination, we can *re-use* L and U to solve each new right-hand side:

1. Find $A = LU$ by Gaussian elimination
2. Solve $Ax_1 = b_1$ by `x₁ = U \ (L \ b₁)`
3. Solve $Ax_1 = b_2$ by `x₂ = U \ (L \ b₂)`
4. etcetera

Since solving triangular systems of equations (L or U) is easy, this way we only do the hard/expensive part (Gaussian elimination once).

Julia provides a shorthand for this process, so you don't have to worry about L and U and explicit forward/backsubstitution. Instead, you compute `LU = lufact(A)`, which creates an "LU factorization object" `LU` that internally stores L and U in a compressed format (along with any permutations/row swaps as discussed below), and then you can do `LU \ b` for each new right-hand side and it will do the (fast) triangular solves:

```
In [12]: LU = lufact(A)

Out[12]: Base.LinAlg.LU{Float64,Array{Float64,2}}([4.0 -2.0 ... -4.0 -8.0; 2.25 -1.5 ... 8.0 13.0; ...

In [13]: [LU\b  A\b] # print them side by side

Out[13]: 5×2 Array{Float64,2}:
           0.505958   0.505958
          -0.928506  -0.928506
           2.16407    2.16407
           1.46166    1.46166
          -1.26428   -1.26428
```

Equivalently, if we let $B = (b_1\ b_2\ \cdots)$ be the matrix whose columns are the right-hand sides, and $X = (x_1\ x_2\ \cdots)$ be the matrix whose columns are the solutions, then solving $Ax_1 = b_1$, $Ax_2 = b_2$, ... is equivalent to solving $AX = B$, because $AX = (Ax_1\ Ax_2\ \cdots)$ in the "matrix × columns" picture of matrix multiplication:

```
In [14]: b₁ = rand(-9:9, 5)
         b₂ = rand(-9:9, 5)
         x₁ = A \ b₁
         x₂ = A \ b₂
         [x₁ x₂] # print results side by side

Out[14]: 5×2 Array{Float64,2}:
          -0.87626   -8.72869
          -0.515124  -6.74427
           2.09991    1.8561
           1.6908    12.2282
          -2.61717  -11.2915

In [15]: B = [b₁ b₂]
         A \ B

Out[15]: 5×2 Array{Float64,2}:
          -0.87626   -8.72869
          -0.515124  -6.74427
           2.09991    1.8561
           1.6908    12.2282
          -2.61717  -11.2915
```

It gives the same answer! On a computer, solving for a bunch of right-hand sides at once by `A \ B` is often **more efficient** than solving them one by one (for technical reasons involving the speed of memory access). Conceptually, it is often convenient to think of many right-hand sides and solutions together, in a matrix, rather than separately.

3 Row swaps and PA = LU

Up to now, we have mostly ignored the possibility of row swaps. Row swaps may be *required* if you encounter a zero pivot (assuming there is a nonzero value below it in the same column), but this is often unlikely to occur in practice (especially for random matrices!).

However, even as in the example above where no row swaps were *required*, a computer will often do them *anyway*, in order to minimize roundoff errors. As you saw on pset 1, roundoff errors (the computer only keeps about 15–16 significant digits) can be disastrous if the pivot is merely very small. So, the computer swaps rows to **make the pivot as big as possible**, as strategy called *partial pivoting*. As a result, the `lu` function in Julia returns *three* things: L, U, and the permutation p giving the **re-ordering of the rows** of A that is needed. For example:

```
In [16]: L, U, p = lu(A)
         L

Out[16]: 5×5 Array{Float64,2}:
           1.0        0.0        0.0       0.0        0.0
           1.0        1.0        0.0       0.0        0.0
           0.444444   0.0512821  1.0       0.0        0.0
          -0.111111   0.410256   0.582822  1.0        0.0
          -0.222222  -0.794872   0.171779  0.0242696  1.0

In [17]: U

Out[17]: 5×5 Array{Float64,2}:
          9.0  -6.0  -6.0      -1.0      -5.0
          0.0  13.0  -3.0       6.0      -3.0
          0.0   0.0  -4.17949  -3.86325  -5.62393
          0.0   0.0   0.0       8.67894   9.95297
          0.0   0.0   0.0       0.0      -0.771206

In [18]: p

Out[18]: 5-element Array{Int64,1}:
          2
          4
          1
          5
          3
```

`p` says tells you in what order we should put the rows of A to match the product LU: we should re-order `A` to put row 2 first, then row 4, then row 1, then row 5, then row 3. We can do this in Julia easily by:

```
In [19]: A[p,:] # A with the rows in order p

Out[19]: 5×5 Array{Int64,2}:
           9  -6  -6  -1  -5
           9   7  -9   5  -8
           4  -2  -7  -4  -8
          -1   6  -3   9   6
          -2  -9   3  -5   2
```

This should match LU:

```
In [20]: L*U
```

```
Out[20]: 5×5 Array{Float64,2}:
           9.0  -6.0  -6.0  -1.0  -5.0
           9.0   7.0  -9.0   5.0  -8.0
           4.0  -2.0  -7.0  -4.0  -8.0
          -1.0   6.0  -3.0   9.0   6.0
          -2.0  -9.0   3.0  -5.0   2.0
```

The computer only stores a list of numbers for `p` because that is the most efficient way to store and work with the permutation. However, for algebraic manipulations it is often convenient to think of this as a **permutation matrix** P multiplying A. Since P re-orders the *rows* of A, it must multiply A on the *left*. Constructing P is easy: it just has a single 1 in each row indicating what row of A should go there.

```
In [21]: # construct a permutation matrix P from the permutation vector p
         function permutation_matrix(p)
             P = zeros(Int, length(p),length(p))
             for i = 1:length(p)
                 P[i,p[i]] = 1
             end
             return P
         end
```

```
Out[21]: permutation_matrix (generic function with 1 method)
```

```
In [22]: P = permutation_matrix(p)
```

```
Out[22]: 5×5 Array{Int64,2}:
          0  1  0  0  0
          0  0  0  1  0
          1  0  0  0  0
          0  0  0  0  1
          0  0  1  0  0
```

```
In [23]: P * A == A[p, :]
```

```
Out[23]: true
```

Thus, LU factorization with row swaps corresponds to a factorization

$$PA = LU$$

Now, to solve $Ax = b$, a more complete process is:

1. Factor $PA = LU$
2. Multiply P by both sides to give $PAx = LUx = Pb$
3. Let $c = Ux$ and solve $Lc = Pb$ for c by forward-substitution
4. Solve $Ux = c$ for x by backsubstitution.

Of course, Julia does all of this for you automatically with `A \ b` or `lufact(A) \ b`, but we can do it manually:

```
In [24]: c = L \ b[p] # solve Lc = Pb = b[p]
         x = U \ c # solve Ux = c
```

```
Out[24]: 5-element Array{Float64,1}:
           0.505958
          -0.928506
           2.16407
           1.46166
          -1.26428
```

```
In [25]: A \ b
```

```
Out[25]: 5-element Array{Float64,1}:
           0.505958
          -0.928506
           2.16407
           1.46166
          -1.26428
```

Hooray, this is the same answer as above!

One final point often confuses people here, if you think carefully about the above process. By writing $PA = LU$, it seems like you *first* decide on the row-reordering of A, and *then* compute the LU factorization of PA. But how do you know the proper row-reordering *before* you do elimination? In fact, this is an illusion: the computer figures out the row-reordering as it goes along (partial pivoting as described above), but it then cleverly works backwards to figure out what reordering it *should* have done in the beginning!

4 Singular matrices

If we encounter a zero pivot (or even just a small pivot, on a computer) during Gaussian elimination, we normally swap rows to bring a nonzero pivot up from a subsequent row. However, what if there are *no* nonzero values below the pivot in that column? This is called a singular matrix: we can still proceed with Gaussian elimination, but **we can't get rid of the zero pivot**.

If you have $Ax = b$ where A is singular, then there will typically (for most right-hand sides b) be **no solutions**, but there will occasionally (for very special b) be **infinitely many solutions**. (For 2×2 matrices, solving $Ax = b$ corresponds to finding the intersection of two lines, and a singular case corresponds to two parallel lines — either there are no intersections, or they intersect everywhere.)

For example, consider the following 4×4 matrix $A = LU$:

$$\underbrace{\begin{pmatrix} 2 & -1 & 0 & 3 \\ 4 & -1 & 1 & 8 \\ 6 & 1 & 4 & 15 \\ 2 & -1 & 0 & 0 \end{pmatrix}}_{A} = \underbrace{\begin{pmatrix} 1 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 \\ 3 & 4 & 1 & 0 \\ 1 & 0 & 2 & 1 \end{pmatrix}}_{L} \underbrace{\begin{pmatrix} 2 & -1 & 0 & 3 \\ 0 & 1 & 1 & 2 \\ 0 & 0 & 0 & -2 \\ 0 & 0 & 0 & 1 \end{pmatrix}}_{U}$$

The third pivot in U is zero! Now, suppose we want to solve $Ax = b$. We first solve $Lc = b$ to apply the elimination steps to b. This is no problem since L has 1's along the diagonal. Suppose we get $c = (c_1, c_2, c_3, c_4)$. Then we proceed by backsubstitution to solve $Ux = c$, starting with the last row of U:

$$1 \times x_4 = c_4 \implies x_4 = c_4 0 \times x_3 - 2 \times x_4 = c_3 \implies \text{no solution unless } -2x_4 = -2c_4 = c_3$$

For very special right-hand sides, where $c_3 = 2c_4$, we can plug in *any* x_3 and get a solution (infinitely many solutions). Otherwise, we get *no* solutions.

```
In [26]: [1 0 0 0
          2 1 0 0
          3 4 1 0
          1 0 2 1 ] *
         [2 -1  0  3
```

```
     0  1  1  2
     0  0  0 -2
     0  0  0  1 ]
```

```
Out[26]: 4×4 Array{Int64,2}:
          2  -1  0   3
          4  -1  1   8
          6   1  4  15
          2  -1  0   0
```

You may think that singular cases are not very interesting. In reality, **exactly singular square matrices never occur by accident**. There is always some *deep structure of the underlying problem* that causes the singularity, and understanding this structure is *always* interesting.

On the other hand, **nearly singular** matrices (where the pivots are nonzero but very small) *can* occur by accident, and dealing with them is often a delicate problem because they are very sensitive to roundoff errors. (We call these matrices ill-conditioned.) But that's mostly not a topic for 18.06.

Singular **non-square** systems, where you have **more equations than unknowns** are *very* common and important, and lead to *fitting* problems where one *minimizes the error* in the solution. We will talk more about this later in 18.06.

Some matrices are **more singular than others**. For example, they can have **two zero pivots**:

$$\underbrace{\begin{pmatrix}2 & -1 & 0 & 3\\ 4 & -2 & 1 & 8\\ 6 & 3 & 4 & 15\\ 2 & -1 & 0 & 0\end{pmatrix}}_{A} = \underbrace{\begin{pmatrix}1 & 0 & 0 & 0\\ 2 & 1 & 0 & 0\\ 3 & 4 & 1 & 0\\ 1 & 0 & 2 & 1\end{pmatrix}}_{L}\underbrace{\begin{pmatrix}2 & -1 & 0 & 3\\ 0 & 0 & 1 & 2\\ 0 & 0 & 0 & -2\\ 0 & 0 & 0 & 1\end{pmatrix}}_{U}$$

or **three**:

$$\underbrace{\begin{pmatrix}2 & -1 & 0 & 3\\ 4 & -2 & 1 & 2\\ 6 & 3 & 4 & -2\\ 2 & -1 & 0 & 0\end{pmatrix}}_{A} = \underbrace{\begin{pmatrix}1 & 0 & 0 & 0\\ 2 & 1 & 0 & 0\\ 3 & 4 & 1 & 0\\ 1 & 0 & 2 & 1\end{pmatrix}}_{L}\underbrace{\begin{pmatrix}2 & -1 & 0 & 3\\ 0 & 0 & 1 & 2\\ 0 & 0 & 0 & -2\\ 0 & 0 & 0 & 0\end{pmatrix}}_{U}$$

or **four**:

$$\underbrace{\begin{pmatrix}0 & -1 & 0 & 3\\ 0 & -2 & 1 & 2\\ 0 & 3 & 4 & -2\\ 0 & -1 & 0 & 0\end{pmatrix}}_{A} = \underbrace{\begin{pmatrix}1 & 0 & 0 & 0\\ 2 & 1 & 0 & 0\\ 3 & 4 & 1 & 0\\ 1 & 0 & 2 & 1\end{pmatrix}}_{L}\underbrace{\begin{pmatrix}0 & -1 & 0 & 3\\ 0 & 0 & 1 & 2\\ 0 & 0 & 0 & -2\\ 0 & 0 & 0 & 0\end{pmatrix}}_{U}$$

(Notice how changing only one pivot changes only one column of A: each column of U determines one column of A via our "matrix × columns" viewpoint on matrix multiplication.)

Intuitively, having more zero pivots seems "more singular", and requires "more coincidences" in the right-hand side to have a solution, and has a "bigger infinity" of solutions when there *is* a solution. We will quantify all of these intuitions later in 18.06, when we begin discussing the [null space](https://en.wikipedia.org/wiki/Kernel_(linear_algebra) and [rank](https://en.wikipedia.org/wiki/Rank_(linear_algebra) of a matrix.

```
In [27]: [1 0 0 0
          2 1 0 0
          3 4 1 0
          1 0 2 1 ] *
         [2 -1  0  3
```

```
 0  0  1  2
 0  0  0 -2
 0  0  0  1 ]
```

```
Out[27]: 4×4 Array{Int64,2}:
 2  -1  0   3
 4  -2  1   8
 6  -3  4  15
 2  -1  0   0
```

```
In [28]: [1 0 0 0
          2 1 0 0
          3 4 1 0
          1 0 2 1 ] *
         [2 -1  0  3
          0  0  1  2
          0  0  0 -2
          0  0  0  0 ]
```

```
Out[28]: 4×4 Array{Int64,2}:
 2  -1  0   3
 4  -2  1   8
 6  -3  4  15
 2  -1  0  -1
```

```
In [29]: [1 0 0 0
          2 1 0 0
          3 4 1 0
          1 0 2 1 ] *
         [0 -1  0  3
          0  0  1  2
          0  0  0 -2
          0  0  0  0 ]
```

```
Out[29]: 4×4 Array{Int64,2}:
 0  -1  0   3
 0  -2  1   8
 0  -3  4  15
 0  -1  0  -1
```

5 Matrix inverses

It is often conceptually convenient to talk about the *inverse* A^{-1} of a matrix A, which exists **for any non-singular square matrix**. This is the matrix such that $x = A^{-1}b$ solves $Ax = b$ for any b. The inverse is conceptually convenient becuase it allows us to move matrices around in equations *almost* like numbers (except that matrices don't commute!).

Another way of defining the inverse of a matrix involves the *identity* matrix I. Here is a \$ 5 $\times 5\$ identity matrix :$

$$I = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} e_1 & e_2 & e_3 & e_4 & e_5 \end{pmatrix}$$

where the columns $e_1 \cdots e_5$ of I are the **unit vectors** in each component.

The identity matrix, which can be constructed by `eye(5)` in Julia, has the property that $Ix = x$ for any x, and hence $IA = A$ for any (here 5×5) matrix A:

```
In [30]: I₅ = eye(Int, 5)
```

```
Out[30]: 5×5 Array{Int64,2}:
          1  0  0  0  0
          0  1  0  0  0
          0  0  1  0  0
          0  0  0  1  0
          0  0  0  0  1
```

```
In [31]: I₅ * b == b
```

```
Out[31]: true
```

```
In [32]: I₅ * A == A
```

```
Out[32]: true
```

The inverse matrix A^{-1} is the matrix such that $A^{-1}A = AA^{-1} = I$.

Why does this correspond to solving $Ax = b$? Multiplying both sides on the *left* by A^{-1} (multiplying on the *right* would make no sense: we can't multiply vector×matrix!), we get

$$A^{-1}Ax = Ix = x = A^{-1}b$$

How do we find A^{-1}? The key is the equation $AA^{-1} = I$, which looks just like $AX = B$ for the **right-hand sides consisting of the columns of the identity matrix**, i.e. the unit vectors. So, we just solve $Ax = e_i$ for $i = 1, \ldots, 5$, or equivalently do `A \ I` in Julia. Of course, Julia comes with a built-in function `inv(A)` for computing A^{-1} as well:

```
In [33]: Ainv = A \ I₅
```

```
Out[33]: 5×5 Array{Float64,2}:
            0.0109991   0.529789  -0.908341  -0.635197  -0.0879927
            0.131989    0.35747   -0.900092  -0.622365  -0.055912
           -0.235564   -0.179652   0.370302   0.353804  -0.11549
           -0.301558   -0.69172    1.48701    1.16499    0.0791323
            0.2044      0.678582  -1.29667   -1.05408    0.0314696
```

```
In [34]: Ainv - inv(A)
```

```
Out[34]: 5×5 Array{Float64,2}:
            2.10942e-15   2.44249e-15  -2.88658e-15  -2.66454e-15   4.996e-16
            1.88738e-15   2.77556e-15  -3.21965e-15  -2.55351e-15   5.27356e-16
           -8.04912e-16  -9.99201e-16   1.88738e-15   1.11022e-15  -1.94289e-16
           -3.83027e-15  -4.32987e-15   5.9952e-15    5.55112e-15  -7.63278e-16
            3.16414e-15   3.66374e-15  -5.32907e-15  -4.44089e-15   6.73073e-16
```

(The difference is just roundoff errors.)

```
In [35]: Ainv * A
```

```
Out[35]: 5×5 Array{Float64,2}:
            1.0          -8.32667e-15  -1.49325e-14  -4.88498e-15  -1.0103e-14
            1.46549e-14   1.0          -1.82077e-14  -2.22045e-15  -1.15463e-14
           -6.02296e-15  -1.77636e-15   1.0          -1.11022e-15   5.55112e-15
           -1.58068e-14   1.4877e-14    2.36478e-14   1.0           1.4877e-14
            1.38639e-14  -7.68829e-15  -2.05808e-14  -5.77316e-15   1.0
```

(Again, we get I up to roundoff errors because the computer does arithmetic only to 15–16 significant digits.)

```
In [36]: A * Ainv

Out[36]: 5×5 Array{Float64,2}:
           1.0            8.88178e-16   0.0            0.0           -5.55112e-17
           4.44089e-16   1.0            0.0           -8.88178e-16   0.0
           1.72085e-15  -4.21885e-15   1.0           -4.88498e-15  -1.08247e-15
          -4.44089e-15   2.66454e-15   5.32907e-15   1.0            7.77156e-16
          -3.10862e-15   8.88178e-16  -1.77636e-15   5.32907e-15   1.0
```

Normally, $AB \neq BA$ for two matrices A and B. Why can we multiply A by A^{-1} on either the left or right and get the same answer I? It is fairly easy to see why:

$$AA^{-1} = I \implies AA^{-1}A = IA = A = A(A^{-1}A)$$

Since $A(A^{-1}A) = A$, and A is non-singular (so there is a unique solution to this system of equations), we must have $A^{-1}A = I$.

Matrix inverses are funny, however:

- Inverse matrices are very convenient in *analytical* manipulations, because they allow you to move matrices from one side to the other of equations easily.

- Inverse matrices are **almost never computed** in "serious" numerical calculations. Whenever you see $A^{-1}B$ (or $A^{-1}b$), when you go to *implement* it on a computer you should *read* $A^{-1}B$ as "solve $AX = B$ by some method." e.g. solve it by `A \ B` or by first computing the LU factorization of A and then using it to solve $AX = B$.

One reason that you don't usually compute inverse matrices is that it is wasteful: once you have $PA = LU$, you can solve $AX = B$ directly without bothering to find A^{-1}, and computing A^{-1} requires much more work if you only have to solve a few right-hand sides.

Another reason is that for many special matrices, there are ways to solve $AX = B$ *much* more quickly than you can find A^{-1}. For example, many large matrices in practice are sparse (mostly zero), and often for sparse matrices you can arrange for L and U to be sparse too. Sparse matrices are much more efficient to work with than general "dense" matrices because you don't have to multiply (or even store) the zeros. Even if A is sparse, however, A^{-1} is usually non-sparse, so you lose the special efficiency of sparsity if you compute the inverse matrix.