Gaussian-elimination

September 7, 2017

1 Gaussian elimination

This Julia notebook allows us to interactively visualize the process of Gaussian elimination.

Recall that the process of Gaussian elimination involves subtracting rows to turn a matrix A into an upper triangular matrix U. Often we *augment* the matrix with an additional column, representing the right-hand side b of a system of equations $Ax = b$ that we want to solve: by doing the same row operations to both A and b, we arrive at an equivalent equation $Ux = c$ that is easy to solve by *backsubstitution* (solving for one variable at a time, working from the last row to the top row).

For example, suppose we are solving:

$$Ax = \begin{pmatrix} 1 & 3 & 1 \\ 1 & 1 & -1 \\ 3 & 11 & 6 \end{pmatrix} x = \begin{pmatrix} 9 \\ 1 \\ 35 \end{pmatrix} = b$$

We would perform the following elimination process.

$$\left[\begin{array}{ccc|c} 1 & 3 & 1 & 9 \\ 1 & 1 & -1 & 1 \\ 3 & 11 & 6 & 35 \end{array}\right] \rightarrow \left[\begin{array}{ccc|c} 1 & 3 & 1 & 9 \\ 0 & -2 & -2 & -8 \\ 0 & 2 & 3 & 8 \end{array}\right] \rightarrow \left[\begin{array}{ccc|c} 1 & 3 & 1 & 9 \\ 0 & -2 & -2 & -8 \\ 0 & 0 & 1 & 0 \end{array}\right]$$

It is much more fun to let the computer do the arithmetic than to crunch through it ourselves on the blackboard, but usually the computer does things *too* quickly (and it often does some re-ordering of the rows that makes it harder to follow what is going on). For example, in Julia, we can solve the above system of equations by simply:

```
In [1]: A = [1 3  1
             1 1 -1
             3 11 6]
```

```
Out[1]: 3×3 Array{Int64,2}:
         1   3   1
         1   1  -1
         3  11   6
```

```
In [2]: b = [9, 1, 35]
```

```
Out[2]: 3-element Array{Int64,1}:
          9
          1
         35
```

```
In [3]: x = A \ b # solves Ax = b by (essentially) Gaussian elimination
```

```
Out[3]: 3-element Array{Float64,1}:
         -3.0
          4.0
          0.0
```

Of course, the computer can solve *much bigger problems* easily. It can solve 1000 equations in 1000 unknowns in a fraction of a second — nowadays, that is no longer considered a "big" system of equations.

```
In [4]: Ahuge = rand(1000,1000)
```

```
Out[4]: 1000×1000 Array{Float64,2}:
 0.999439   0.218541   0.926735    ...  0.18134    0.440472  0.370998
 0.254565   0.407119   0.892029         0.245744   0.428122  0.417186
 0.442499   0.344993   0.105348         0.788985   0.990621  0.816038
 0.484535   0.87536    0.4178           0.1804     0.986928  0.424028
 0.0200728  0.212215   0.666981         0.546373   0.294638  0.250384
 0.354426   0.0122195  0.693046    ...  0.048007   0.339813  0.90019
 0.173619   0.281971   0.850616         0.808064   0.237943  0.0276862
 0.682416   0.556563   0.166403         0.754182   0.285963  0.0600911
 0.385231   0.745615   0.070297         0.570591   0.757603  0.250422
 0.109478   0.13033    0.21048          0.420667   0.949553  0.190161
 0.416554   0.429042   0.127873    ...  0.742661   0.542352  0.473639
 0.95743    0.18821    0.831533         0.22597    0.640721  0.482018
 0.511667   0.171514   0.61173          0.761556   0.932857  0.290719
 ⋮                                    ⋱
 0.903682   0.458837   0.904612         0.526082   0.969423  0.98074
 0.799626   0.810578   0.397511         0.689131   0.380224  0.787858
 0.923511   0.034355   0.999014    ...  0.97647    0.516341  0.24342
 0.0292043  0.508768   0.722043         0.648136   0.572726  0.522543
 0.794563   0.111256   0.594695         0.862215   0.821355  0.452182
 0.179553   0.342921   0.128901         0.883977   0.405025  0.328367
 0.486401   0.255297   0.53484          0.251865   0.288409  0.825427
 0.649021   0.454363   0.47842     ...  0.30454    0.545873  0.387851
 0.728921   0.586378   0.938405         0.828617   0.413401  0.447249
 0.165084   0.393614   0.00965933       0.413297   0.108621  0.288142
 0.615864   0.103994   0.66932          0.0722988  0.617049  0.322095
 0.778809   0.488329   0.0868491        0.933352   0.892181  0.952343
```

```
In [5]: bhuge = rand(1000)
```

```
Out[5]: 1000-element Array{Float64,1}:
 0.102929
 0.0309441
 0.344267
 0.586685
 0.225116
 0.211111
 0.865491
 0.37005
 0.266376
 0.165927
 0.911616
 0.34382
 0.256057
 ⋮
 0.822513
 0.425997
 0.914556
 0.236236
```

```
 0.797237
 0.3844
 0.0226938
 0.669134
 0.419199
 0.439416
 0.73799
 0.228102
```

```
In [6]: Ahuge \ bhuge
```

```
Out[6]: 1000-element Array{Float64,1}:
  0.059388
  0.170628
  1.87947
  0.984206
 -0.466225
 -0.293058
  0.312031
 -0.290076
 -0.214995
  0.628901
 -0.389962
 -1.33457
 -0.874197
  ⋮
 -0.496138
 -0.591704
 -0.296718
  0.0107591
 -1.31569
  2.23445
 -0.221941
 -0.322383
 -1.60712
 -0.104835
 -0.496091
 -0.0145873
```

```
In [7]: @time Ahuge \ bhuge;
```

```
0.050512 seconds (140 allocations: 7.653 MB)
```

If we want to see the matrix U from above, we use the fact (covered soon in 18.06) that Gaussian elimination is really "LU" factorization, performed by the built-in function `lu`. By default, however, "serious" computer implementations of this process automatically re-order the rows to reduce the effect of roundoff errors, so we need to pass an extra option that tells Julia not to do this. (You should *not* normally do this, except for learning exercises.)

```
In [8]: # LU factorization (Gaussian elimination) of the augumented matrix [A b],
        # passing the undocumented option Val{false} to prevent row re-ordering
        L, U = lu([A b], Val{false})
        U # just show U
```

```
Out[8]: 3×4 Array{Float64,2}:
 1.0   3.0   1.0   9.0
```

```
 0.0  -2.0  -2.0  -8.0
 0.0   0.0   1.0   0.0
```

However, it would be nice to show the individual steps of this process. This requires some programming.

2 Code to interactively visualize Gaussian elimination

The following is some slightly tricky code that lets us visualize the process of Gaussian elimination in Julia. It takes advantage of the Interact package in Julia, which allows us to easily create interactive displays using sliders, pushbuttons, and other widgets.

Implementing this is **not really a beginner exercise** for new Julia programmers, though it is fairly straightforward for people who are used to Julia. It involves defining our own type to control display, our own implementation of Gaussian elimination that allows us to stop partway through, and using the Interact package to create interactive widgets.

You can skip this part if you aren't ready for the programming details.

In [9]:
```
"""
TwoMatrices is just a wrapper type around two matrices or vectors with the same
number of rows, so that they can be displayed side-by-side with a title and
and arrow pointing from left to right.
"""
type TwoMatrices
    left::AbstractVecOrMat
    right::AbstractVecOrMat
    title::AbstractString
    function TwoMatrices(left, right, title="")
        size(left,1) == size(right,1) || throw(DimensionMismatch("two matrices must have same #
        return new(left, right, title)
    end
end
function Base.show(io::IO, ::MIME"text/plain", x::TwoMatrices)
    isempty(x.title) || println(io, x.title)
    m = size(x.left, 1)
    s = [Text(" "^10) for i in 1:m]
    s[(m+1)÷2] = Text("   --->   ")
    Base.showarray(io, [x.left s x.right], false; header=false)
end

"""
    naive_gauss(A, [step])

Given a matrix `A`, performs Gaussian elimination to convert
`A` into an upper-triangular matrix `U`.

This implementation is "naive" because it *never re-orders the rows*.
(It will obviously fail if a zero pivot is encountered.)

If the optional `step` argument is supplied, only performs `step`
steps of Gaussian elimination.

Returns `(U, row, col, factor)`, where `row` and `col` are the
row and column of the last step performed, while `factor`
is the last factor multiplying the pivot row.
"""
```

```
function naive_gauss(A, step=typemax(Int))
    m = size(A,1) # number of rows
    factor = A[1,1]/A[1,1]
    step ≤ 0 && return (A, 1, 1, factor)
    U = copy!(similar(A, typeof(factor)), 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]:
            factor = -U[i,j]/U[j,j]
            U[i,:] = U[i,:] + U[j,:] * factor
            step -= 1
            step ≤ 0 && return (U, i, j, factor)
        end
    end
    return U, m, m, factor
end
```

Out[9]: naive_gauss

In [10]:
```
using Interact

# For display, I only want to show 3 decimal places of floating-point values,
# but I want to show integers and similar types exactly, so I define a little
# function to do this rounding
shorten(x::AbstractFloat) = round(x, 3)
shorten(x) = x # leave non floating-point values as-is

# create an interactive widget to visualize the Gaussian-elimination process for the matrix A.
function visualize_gauss(A)
    m = size(A, 1)
    @manipulate for step in slider(1:(m*(m-1))÷2, value=1, label="gauss step")
        Uprev, = naive_gauss(A, step-1)
        U, row, col, factor = naive_gauss(A, step)
        pivot = U[col,col]
        TwoMatrices(shorten.(Uprev), shorten.(U), "Gaussian elimination for column $col with p
    end
end
```

Out[10]: visualize_gauss (generic function with 1 method)

3 Gaussian elimination examples

Now, let's use this machinery to interact with some examples, starting with our 3×3 matrix from above:

In [11]:
```
visualize_gauss([A b])
```

```
Interact.Slider{Int64}(Signal{Int64}(1, nactions=1),"gauss step",1,1:3,"horizontal",true,"d",true)
```

Out[11]:
```
Gaussian elimination for column 1 with pivot 1.0: add -1.0 * (row 1) to (row 2)
 1   3   1   9             1.0   3.0   1.0   9.0
 1   1  -1   1     --->    0.0  -2.0  -2.0  -8.0
 3  11   6  35             3.0  11.0   6.0  35.0
```

In [12]:
```
visualize_gauss(rand(-9:9,5,5))
```

```
Interact.Slider{Int64}(Signal{Int64}(1, nactions=1),"gauss step",1,1:10,"horizontal",true,"d",true)
```

```
Out[12]: Gaussian elimination for column 1 with pivot -2.0: add 1.5 * (row 1) to (row 2)
          -2   3   1  -6   0              -2.0   3.0   1.0  -6.0   0.0
           3  -8   2   2   3               0.0  -3.5   3.5  -7.0   3.0
           3   8  -2   1   7     --->      3.0   8.0  -2.0   1.0   7.0
          -1  -3  -1  -6  -1              -1.0  -3.0  -1.0  -6.0  -1.0
           6   9   1   7   1               6.0   9.0   1.0   7.0   1.0
```

Of course, because we are not re-ordering the rows, this process can go horribly wrong, most obviously if a zero pivot is encountered:

```
In [13]: Abad = [-3   5   5   3  -7
                  3  -5   8  -8  -6
                  8   2   8   2  -8
                 -6  -2   6   4  -8
                 -8   4  -6  -1   8]
         visualize_gauss(Abad)
```

```
Interact.Slider{Int64}(Signal{Int64}(1, nactions=1),"gauss step",1,1:10,"horizontal",true,"d",true)
```

```
Out[13]: Gaussian elimination for column 1 with pivot -3.0: add 1.0 * (row 1) to (row 2)
          -3   5   5   3  -7              -3.0   5.0   5.0   3.0   -7.0
           3  -5   8  -8  -6               0.0   0.0  13.0  -5.0  -13.0
           8   2   8   2  -8     --->      8.0   2.0   8.0   2.0   -8.0
          -6  -2   6   4  -8              -6.0  -2.0   6.0   4.0   -8.0
          -8   4  -6  -1   8              -8.0   4.0  -6.0  -1.0    8.0
```

But this matrix is not actually singular:

```
In [14]: det(Abad)
```

```
Out[14]: 19211.999999999996
```

So we can fix the problem just by re-ordering the rows, e.g. swapping the first and last rows:

```
In [15]: Aok = [-8   4  -6  -1   8
                 3  -5   8  -8  -6
                 8   2   8   2  -8
                -6  -2   6   4  -8
                -3   5   5   3  -7]
         visualize_gauss(Aok)
```

```
Interact.Slider{Int64}(Signal{Int64}(1, nactions=1),"gauss step",1,1:10,"horizontal",true,"d",true)
```

```
Out[15]: Gaussian elimination for column 1 with pivot -8.0: add 0.375 * (row 1) to (row 2)
          -8   4  -6  -1   8              -8.0   4.0  -6.0   -1.0     8.0
           3  -5   8  -8  -6               0.0  -3.5   5.75  -8.375  -3.0
           8   2   8   2  -8     --->      8.0   2.0   8.0    2.0    -8.0
          -6  -2   6   4  -8              -6.0  -2.0   6.0    4.0    -8.0
          -3   5   5   3  -7              -3.0   5.0   5.0    3.0    -7.0
```

3.1 A bigger example

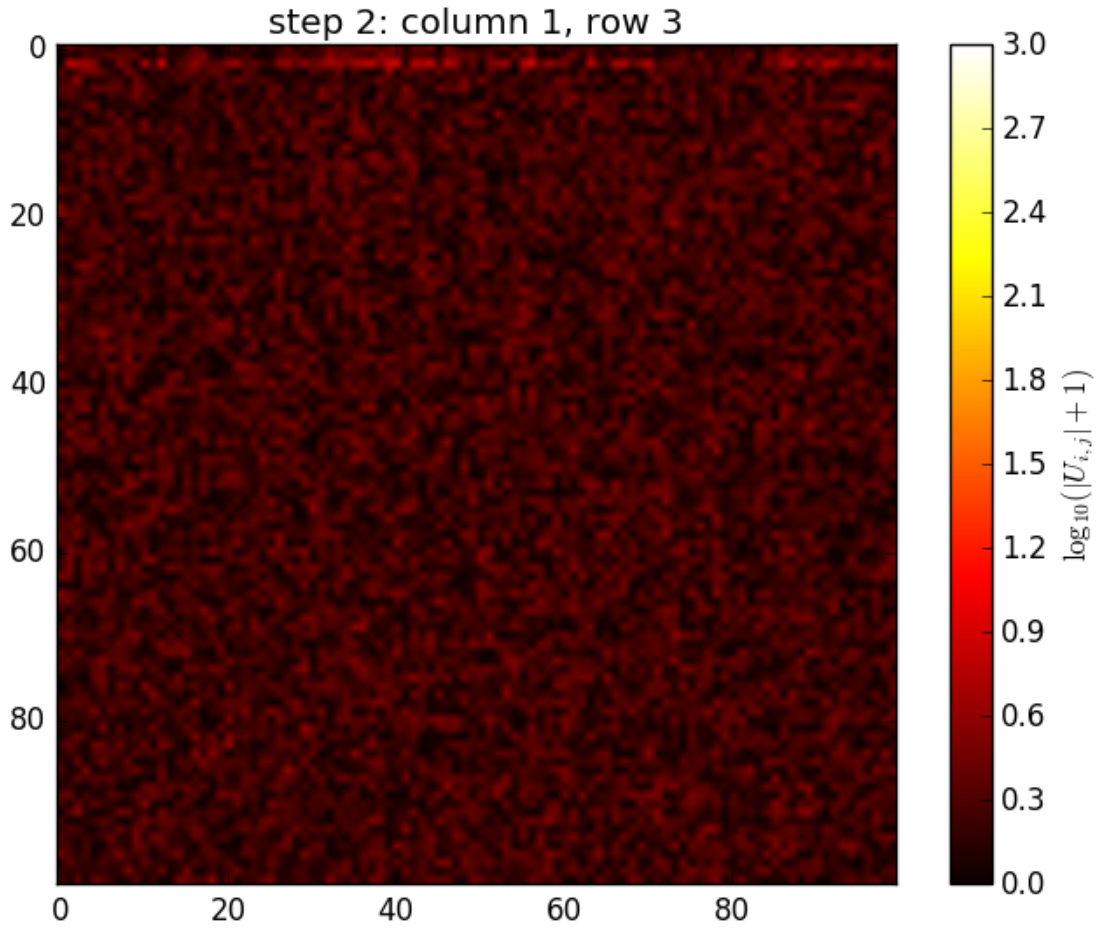
We quickly run out of space for displaying matrices as text, but we can visualize the process for larger matrices by using images, with the PyPlot package (a wrapper around the Python Matplotlib library):

```
In [16]: using PyPlot
```

```
In [18]: m = 100
         Abig = randn(m,m)
         fig = figure()
         nsteps = (m*(m-1))÷2
         @manipulate for step in slider(0:50:nsteps, value=2, label="gauss step")
             withfig(fig) do
                 U, row, col = naive_gauss(Abig, step)
                 # I had to experiment a little to find a nice way to plot this
                 imshow(log10.(abs.(U) .+ 1), cmap="hot", vmin=0, vmax=3)
                 title("step $step: column $col, row $row")
                 colorbar(label=L"\log_{10}(|U_{i,j}| + 1)")
             end
         end
```

```
Interact.Slider{Int64}(Signal{Int64}(2, nactions=1),"gauss step",2,0:50:4950,"horizontal",true,"d",true
```

Out[18]:



Note that it takes a *lot* more steps of Gaussian elimination for a 100×100 matrix (4950 steps) than for a 5×5 matrix (10 steps). Later on in 18.06, we will analyze the computational cost of Gaussian elimination and how it scales with the size of the matrix (in computer science, this is known as the complexity of the algorithm).