Table of Contents

This notebook accompanies the lectures on LU factorization from January 28 and 30, 2020.

In [1]:

```
using LinearAlgebra, Random
Random.seed!(0);
A = randn(5,5)
```

Out[1]:

```
5×5 Array{Float64,2}:
  0.679107   0.297336   -0.688907  -0.187573  -0.117138
  0.828413   0.0649475  -0.762804  -1.60726   -0.601254
 -0.353007  -0.109017    0.397482  -2.48079    1.14228
 -0.134854  -0.51421     0.81163    2.27623   -0.0886163
  0.586617   1.57433    -0.346355   0.219693   0.279466
```

Computing an LU factorization using operators

The following is the data structure behind the operator that subtracts c times the ith entry of a vector from the jth entry.

In [2]:

```
struct RowOp
    c
    i::Int
    j::Int
end
```

In [3]:

```
op = RowOp(3.5, 2, 3)
```

Out[3]:

```
RowOp(3.5, 2, 3)
```

The following makes the operator callable as a function (if you want to understand, see https://docs.julialang.org/en/v1/manual/methods/#Function-like-objects-1 (https://docs.julialang.org/en/v1/manual/methods/#Function-like-objects-1)) Note that it does not modify the input vector, but rather produces a new one. This slows things down a bit, but it is easier to understand. You could fix it.

In [4]:

```
function (op::RowOp)(x::Vector)
    y = copy(x)
    y[op.j] = y[op.j] - op.c * y[op.i]
    return y
end
```

In [5]:

```
x = collect(1:3)
[x op(x)]
```

Out[5]:

```
3×2 Array{Int64,2}:
 1   1
 2   2
 3  -4
```

Now, we make it applicable to a matrix.

In [6]:

```
function (op::RowOp)(A::Matrix)
    M = copy(A)
    M[op.j,:] = M[op.j,:] - op.c * M[op.i,:]
    return M
end
```

In [7]:

```
M = rand(4,4)
```

Out[7]:

```
4×4 Array{Float64,2}:
 0.0668464  0.838118  0.119653   0.484661
 0.156637   0.914712  0.76707    0.899199
 0.605297   0.300075  0.801924   0.951691
 0.135745   0.72285   0.0353445  0.801119
```

In [8]:

```
op(M)
```

Out[8]:

```
4×4 Array{Float64,2}:
 0.0668464   0.838118   0.119653    0.484661
 0.156637    0.914712   0.76707     0.899199
 0.0570685  -2.90142   -1.88282    -2.19551
 0.135745    0.72285    0.0353445   0.801119
```

Let's use this to make a matrix upper triangular.

In [9]:

```
M = copy(A)
n = size(M,1)
ops = []  # define an empty array
for i in 1:(n-1)
    for j in (i+1):n
        op = RowOp(M[j,i] / M[i,i], i, j)
        push!(ops, op)  # append the operator to the array
        M = op(M)
    end
end
U = copy(M)
```

Out[9]:

```
5×5 Array{Float64,2}:
  0.679107      0.297336     -0.688907   -0.187573   -0.117138
  1.11022e-16  -0.297759      0.0775639  -1.37844    -0.458363
  1.69804e-17   6.93889e-18   0.0512441  -2.78912     1.01128
 -3.54038e-16  -7.53228e-17   0.0        34.6225    -10.3888
  5.66075e-16  -2.24999e-17   0.0         0.0        -5.37728
```

Let's look at the array of operators

In [10]:

```
ops
```

Out[10]:

```
10-element Array{Any,1}:
 RowOp(1.2198563160114704, 1, 2)
 RowOp(-0.5198108381191501, 1, 3)
 RowOp(-0.19857516905347045, 1, 4)
 RowOp(0.8638060079204682, 1, 5)
 RowOp(-0.1529456416507074, 2, 3)
 RowOp(1.5286394701005674, 2, 4)
 RowOp(-4.424677813580107, 2, 5)
 RowOp(10.855157730833723, 3, 4)
 RowOp(11.551047182549553, 3, 5)
 RowOp(0.765393453168502, 4, 5)
```

Now, let's use it to solve Ax = b, for a random b.

In [11]:

```
b = randn(n)
```

Out[11]:

```
5-element Array{Float64,1}:
 -0.32194315289596 94
  0.24634263300826426
 -0.17878873231302242
 -1.477880724921868
 -0.18542418173298963
```

For comparison, here is the solution computed by Julia's default solver, which uses LU.

In [12]:

```
x0 = A \ b
```

Out[12]:

```
5-element Array{Float64,1}:
 -2.0307264324467953
  0.6509783157425806
 -0.9902783709508121
 -0.31034732409580573
 -1.051384191424946
```

In [13]:

```
y = copy(b)
for op in ops   # could have written i in 1:length(ops); op = ops[i]
    y = op(y)
end
y
```

Out[13]:

```
5-element Array{Float64,1}:
 -0.3219431528959694
  0.6390670214650591
 -0.2483957567908181
  0.17766140564925426
  5.653588149085766
```

In [14]:

```
# Let's check that we get the same answer
x = U \ y
[x x0]
```

Out[14]:

```
5×2 Array{Float64,2}:
 -2.03073   -2.03073
  0.650978   0.650978
 -0.990278  -0.990278
 -0.310347  -0.310347
 -1.05138   -1.05138
```

In [15]:

```
# And check that both are correct
[A*x b]
```

Out[15]:

```
5×2 Array{Float64,2}:
 -0.321943  -0.321943
  0.246343   0.246343
 -0.178789  -0.178789
 -1.47788   -1.47788
 -0.185424  -0.185424
```

We solved the system!

Now, let's construct L from this.

In [16]:

```
L = zeros(n,n)
for op in ops
    L[op.j, op.i] = op.c
end
L = L + I  # set the diagonals to 1
L
```

Out[16]:

```
5×5 Array{Float64,2}:
  1.0        0.0        0.0     0.0       0.0
  1.21986    1.0        0.0     0.0       0.0
 -0.519811  -0.152946   1.0     0.0       0.0
 -0.198575   1.52864   10.8552  1.0       0.0
  0.863806  -4.42468   11.551   0.765393  1.0
```

We now see that LU = A

In [17]:

```
L*U - A
```

Out[17]:

```
5×5 Array{Float64,2}:
 0.0           0.0           0.0  0.0           0.0
 0.0           1.38778e-17   0.0  0.0           0.0
 0.0          -1.38778e-17   0.0  4.44089e-16   0.0
 0.0           0.0           0.0  5.32907e-15  -1.04083e-15
 1.11022e-16   0.0           0.0  8.32667e-17   1.11022e-16
```

Backsolves

Here are the two versions of the backsolve that we saw in class.

In [18]:

```
function backsolve1(L, b)
    n = length(b)
    x = zeros(n)
    for i in 1:n
        x[i] = b[i]
        for j in 1:(i-1)
            x[i] = x[i] - L[i,j]*x[j]
        end
        x[i] = x[i] / L[i,i]
    end
    return x
end

function backsolve2(L,b)
    n = length(b)
    x = copy(b)
    for i in 1:n
        x[i] = x[i] / L[i,i]
        for j in (i+1):n
            x[j] = x[j] - L[j,i]*x[i]
        end
    end
    return x
end
```

Out[18]:

```
backsolve2 (generic function with 1 method)
```

Let's see how we would use them to solve the system above.

In [18]:

```
y1 = backsolve1(L, b)
x1 = U \ y1

y2 = backsolve2(L, b)

y3 = L \ b  # the built-in backsolve

[y1 y2 y3]
```

Out[18]:

```
5×3 Array{Float64,2}:
 -0.321943  -0.321943  -0.321943
  0.639067   0.639067   0.639067
 -0.248396  -0.248396  -0.248396
  0.177661   0.177661   0.177661
  5.65359    5.65359    5.65359
```

In [19]:

```
A * x1 - b
```

Out[19]:

```
5-element Array{Float64,1}:
  1.6653345369377348e-16
  2.7755575615628914e-16
  2.7755575615628914e-17
  2.220446049250313e-16
 -2.7200464103316335e-15
```

Let's see that they do the same thing on some random lower triangular matrix.

In [20]:

```
Lrand = tril(randn(5,5))
```

Out[20]:

```
5×5 Array{Float64,2}:
  1.26972     0.0         0.0          0.0        0.0
 -0.162504   -1.17303     0.0          0.0        0.0
 -0.0671867  -0.879915   -0.0303032    0.0        0.0
  0.577282   -0.494043   -1.31924     -1.53806    0.0
  0.891315    0.511959   -0.482604    -1.22742   -1.05092
```

In [21]:

```
[backsolve1(Lrand, b) backsolve2(Lrand,b)]
```

Out[21]:

```
5×2 Array{Float64,2}:
 -0.253554  -0.253554
 -0.17488   -0.17488
 11.5402    11.5402
 -8.97646   -8.97646
  5.06075    5.06075
```

Partial pivoting

The Julia function `lu` implements LU factorization with partial pivoting. By default, it hides what it returns from the user, so that the user can easily use the result to solve linear equations, like this:

In [22]:

```
F = lu(A);
x = F \ b
[A*x b]
```

Out[22]:

```
5×2 Array{Float64,2}:
 -0.321943  -0.321943
  0.246343   0.246343
 -0.178789  -0.178789
 -1.47788   -1.47788
 -0.185424  -0.185424
```

But, we can extract the matrices L and U, the permutation as a matrix, P, or as a vector, p.

In [23]:

```
L = F.L
U = F.U
P = F.P
p = F.p
```

Out[23]:

```
5-element Array{Int64,1}:
 2
 5
 4
 3
 1
```

In [24]:

```
L*U - P*A
```

Out[24]:

```
5×5 Array{Float64,2}:
  0.0           0.0           0.0   0.0           0.0
 -1.11022e-16   0.0           0.0   8.32667e-17   0.0
  0.0          -1.11022e-16   0.0   0.0           0.0
  0.0          -1.38778e-17   0.0   0.0           0.0
 -1.11022e-16   0.0           0.0  -2.22045e-16  -1.38778e-17
```

And, we see that this produced the LU factorization of PA

Bad examples for Partial Pivoting

The following code produces a matrix that has large growth in U.

In [20]:

```
function badmat(n::Int, ep=0.0)
    A = tril(-(1-ep) .* ones(n,n)) + 2*I
    A[:,n] .= 1
    return A
end
```

Out[20]:

```
badmat (generic function with 2 methods)
```

In [21]:

```
badmat(4)
```

Out[21]:

```
4×4 Array{Float64,2}:
  1.0   0.0   0.0  1.0
 -1.0   1.0   0.0  1.0
 -1.0  -1.0   1.0  1.0
 -1.0  -1.0  -1.0  1.0
```

In [22]:

```
lu(badmat(4)).U
```

Out[22]:

```
4×4 Array{Float64,2}:
 1.0  0.0  0.0  1.0
 0.0  1.0  0.0  2.0
 0.0  0.0  1.0  4.0
 0.0  0.0  0.0  8.0
```

The number going down the right hand size blow up. As you can imagine, this is a problem. Let's see what happens when we try to solve a random 80-by-80 system.

In [23]:

```
n = 80
A = badmat(n)
F = lu(A)
b = randn(n)
x = F \ b
norm(A*x-b)
```

Out[23]:

```
6.446393282368332
```

We want something much smaller than that.

To get some understanding of why this is rare, consider adding a very small random perturbation to A.

In [24]:

```
A_pert = A + randn(size(A))*1e-8
```

Out[24]:

```
80×80 Array{Float64,2}:
  1.0  -1.03289e-8   4.05691e-9  …   1.1596e-8    -6.65777e-9   1.0
 -1.0   1.0          4.33524e-9     -2.6948e-9    -8.87735e-9   1.0
 -1.0  -1.0          1.0           -2.78774e-10    1.29841e-8   1.0
 -1.0  -1.0         -1.0            1.31949e-8    -1.7242e-8    1.0
 -1.0  -1.0         -1.0           -2.90145e-9     2.35664e-11  1.0
 -1.0  -1.0         -1.0         …   8.09318e-9     2.71153e-10  1.0
 -1.0  -1.0         -1.0           -3.54839e-9     8.74589e-9   1.0
 -1.0  -1.0         -1.0            1.46633e-8     1.80119e-10  1.0
 -1.0  -1.0         -1.0           -9.03467e-9    -2.90539e-9   1.0
 -1.0  -1.0         -1.0           -1.35998e-8     3.9422e-9    1.0
 -1.0  -1.0         -1.0         …   1.36314e-8     7.51207e-9   1.0
 -1.0  -1.0         -1.0            6.2755e-9      8.53502e-9   1.0
 -1.0  -1.0         -1.0            8.22802e-9     6.66529e-9   1.0
  ⋮                              ⋱
 -1.0  -1.0         -1.0           -4.63022e-9    -7.8578e-9    1.0
 -1.0  -1.0         -1.0            9.47773e-9     2.41237e-9   1.0
 -1.0  -1.0         -1.0         …   3.99271e-9     8.83972e-9   1.0
 -1.0  -1.0         -1.0           -1.67328e-8    -1.44875e-8   1.0
 -1.0  -1.0         -1.0            2.88583e-9    -6.07448e-9   1.0
 -1.0  -1.0         -1.0           -5.78811e-9     2.5772e-10   1.0
 -1.0  -1.0         -1.0            7.9141e-9     -2.18598e-8   1.0
 -1.0  -1.0         -1.0         …   8.31238e-9     1.16291e-8   1.0
 -1.0  -1.0         -1.0           -2.08328e-9     2.64571e-9   1.0
 -1.0  -1.0         -1.0            1.0           -6.43127e-9   1.0
 -1.0  -1.0         -1.0           -1.0            1.0          1.0
 -1.0  -1.0         -1.0           -1.0           -1.0          1.0
```

In [25]:

```
F_pert = lu(A_pert)
x_pert = F_pert \ b
norm(A_pert * x_pert - b)
```

Out[25]:

```
1.6058520737947044e-14
```

Now, we've got 15 digits of precision! So, the bad examples are *very* unstable.

Note that the matrix A is well conditioned. The following computes its condition number.

In [26]:

```
cond(A)
```

Out[26]:

```
35.801948504751344
```

This means that the solutions to A_pert and A should be similar, and they are!

In [27]:

```
norm(A * x_pert - b)
```

Out[27]:

```
4.764447327192292e-7
```

The fact that a small perturbation makes this bad example good would be more impressive if we only needed to perturb the nonzero entries. In fact, only perturbing the nonzeros fixes this example.

In [28]:

```
A_pert = A + (A .!= 0).*randn(size(A))*1e-8
F_pert = lu(A_pert)
x_pert = F_pert \ b
norm(A_pert * x_pert - b)
```

Out[28]:

```
1.1931969935863511e-14
```

But, we can make a slightly more robust example for which this does not suffice by scaling up the diagonal.

In [29]:

```
A = badmat(4,0.01)
```

Out[29]:

```
4×4 Array{Float64,2}:
  1.01   0.0    0.0   1.0
 -0.99   1.01   0.0   1.0
 -0.99  -0.99   1.01  1.0
 -0.99  -0.99  -0.99  1.0
```

In [30]:

```
n = 80
A = badmat(n, 0.01)
A_pert = A + (A .!= 0).*randn(size(A))*1e-8
```

Out[30]:

```
80×80 Array{Float64,2}:
  1.01   0.0    0.0    0.0    0.0    0.0   …   0.0    0.0    0.0
0.0   1.0
 -0.99   1.01   0.0    0.0    0.0    0.0       0.0    0.0    0.0
0.0   1.0
 -0.99  -0.99   1.01   0.0    0.0    0.0       0.0    0.0    0.0
0.0   1.0
 -0.99  -0.99  -0.99   1.01   0.0    0.0       0.0    0.0    0.0
0.0   1.0
 -0.99  -0.99  -0.99  -0.99   1.01   0.0       0.0    0.0    0.0
0.0   1.0
 -0.99  -0.99  -0.99  -0.99  -0.99   1.01  …   0.0    0.0    0.0
0.0   1.0
 -0.99  -0.99  -0.99  -0.99  -0.99  -0.99       0.0    0.0    0.0
0.0   1.0
 -0.99  -0.99  -0.99  -0.99  -0.99  -0.99       0.0    0.0    0.0
0.0   1.0
 -0.99  -0.99  -0.99  -0.99  -0.99  -0.99       0.0    0.0    0.0
0.0   1.0
 -0.99  -0.99  -0.99  -0.99  -0.99  -0.99  …   0.0    0.0    0.0
0.0   1.0
 -0.99  -0.99  -0.99  -0.99  -0.99  -0.99       0.0    0.0    0.0
0.0   1.0
 -0.99  -0.99  -0.99  -0.99  -0.99  -0.99       0.0    0.0    0.0
0.0   1.0
  ⋮                                   ⋮     ⋱    ⋮
 -0.99  -0.99  -0.99  -0.99  -0.99  -0.99       0.0    0.0    0.0
0.0   1.0
 -0.99  -0.99  -0.99  -0.99  -0.99  -0.99       0.0    0.0    0.0
0.0   1.0
 -0.99  -0.99  -0.99  -0.99  -0.99  -0.99  …   0.0    0.0    0.0
0.0   1.0
 -0.99  -0.99  -0.99  -0.99  -0.99  -0.99       0.0    0.0    0.0
0.0   1.0
 -0.99  -0.99  -0.99  -0.99  -0.99  -0.99       0.0    0.0    0.0
0.0   1.0
 -0.99  -0.99  -0.99  -0.99  -0.99  -0.99  …   1.01   0.0    0.0
0.0   1.0
 -0.99  -0.99  -0.99  -0.99  -0.99  -0.99      -0.99   1.01   0.0
0.0   1.0
 -0.99  -0.99  -0.99  -0.99  -0.99  -0.99      -0.99  -0.99   1.01
0.0   1.0
 -0.99  -0.99  -0.99  -0.99  -0.99  -0.99      -0.99  -0.99  -0.99
1.01  1.0
 -0.99  -0.99  -0.99  -0.99  -0.99  -0.99      -0.99  -0.99  -0.99  -
0.99  1.0
```

In [31]:

```
F_pert = lu(A_pert)
x_pert = F_pert \ b
norm(A_pert * x_pert - b)
```

Out[31]:

```
1.6125163273340238e7
```

But, if we perturb all entries than everything is fine again.

In [32]:

```
A_pert = A + randn(size(A))*1e-8
F_pert = lu(A_pert)
x_pert = F_pert \ b
norm(A_pert * x_pert - b)
```

Out[32]:

```
2.4281596638541593e-8
```

In [ ]: