Lecture 11: Matrix factorizations

In this lecture, we look at several factorizations of a matrix:

$$A = LU$$

where L is lower triangular and U is upper triangular,

$$A = PLU$$

where P is a permutation matrix, L is lower triangular and U is upper triangular, and

$$A = QR$$

where Q is an orthogonal matrix and R is upper triangular.

The importance of these decomposition is that their component pieces are easy to invert on a computer:

$$\begin{aligned} A = LU \Rightarrow A^{-1} &= U^{-1}L^{-1} \\ A = PLU \Rightarrow A^{-1} &= U^{-1}L^{-1}P^\top \\ A = QR \Rightarrow A^{-1} &= R^{-1}Q^\top \end{aligned}$$

and we saw last lecture that triangular matrices are easy to invert.

First we run some setup code:

In [128]:

```
# backsubstitution from last lecture
function backsubstitution(U,b)
    n=size(U,1)

    if length(b) != n
        error("The system is not compatible")
    end


    x=zeros(n)  # the solution vector
    for k=n:-1:1  # start with k=n, then k=n-1, ...
        r=b[k]  # dummy variable
        for j=k+1:n
            r -= U[k,j]*x[j] # equivalent to r = r-U[k,j]*x[j]
        end
        x[k]=r/U[k,k]
    end
    x
end

# special function that returns the LU Decomposition, without pivoting
function lu_nopivot(A)

    LUF=lufact(A,Val{false})
    if LUF.info == 1
        error("LU Factorization Failed")
    end
    LUF[:L],LUF[:U]
end
```

Out[128]:

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

LU Decomposition

The custom routine `lu_nopivot` defined above returns the LU decomposition of a matrix, if it exists:

In [116]:

```
A=[1 2  3;
   4 6.9 10;
   10 52 3]

L,U=lu_nopivot(A)

A-L*U
```

Out[116]:

```
3x3 Array{Float64,2}:
 0.0  0.0  0.0
 0.0  0.0  0.0
 0.0  0.0  0.0
```

Having the decomposition allows us to reduce inverting A to inverting L and U. We can see this as all the entries are small:

In [117]:

```
(inv(A) - inv(U)*inv(L))
```

Out[117]:

```
3x3 Array{Float64,2}:
 -1.77636e-15   4.44089e-16  -1.52656e-16
  2.22045e-16  -5.55112e-17   3.46945e-18
  6.66134e-16  -1.66533e-16  -6.93889e-18
```

Note that U^{-1} can be calculated column-by-column: $U^{-1}\mathbf{e}_k$ gives the kth column of U^{-1}:

In [120]:

```
Ui=[backsubstitution(U,[1,0,0]) backsubstitution(U,[0,1,0]) backsubstitution
(U,[0,0,1])]
inv(U)-Ui
```

Out[120]:

```
3x3 Array{Float64,2}:
 0.0  0.0  0.0
 0.0  0.0  0.0
 0.0  0.0  0.0
```

LU Decomposition can fail, for example, if the $(1,1)$ entry is zero:

In [129]:

```
A=[0  2   3;
   4  6.9 10;
   10 52  3]

lu_nopivot(A)
```

```
LoadError: LU Factorization Failed
while loading In[129], in expression starting on line 5

 in lu_nopivot at In[128]:25
```

These cases are very special: perturbing the entry by a little bit and we can find the LU Decomposition:

In [130]:

```
A=[1E-12 2    3;
   4     6.9  10;
   10    52   3]

L,U=lu_nopivot(A)
```

Out[130]:

```
(
3x3 Array{Float64,2}:
 1.0      0.0  0.0
 4.0e12   1.0  0.0
 1.0e13   2.5  1.0,


3x3 Array{Float64,2}:
 1.0e-12   2.0       3.0
 0.0      -8.0e12   -1.2e13
 0.0       0.0       74.125 )
```

Unfortunately, the accuracy is lost, we are only accurate to 3 digits:

In [131]:

```
A-L*U
```

Out[131]:

```
3x3 Array{Float64,2}:
 0.0   0.0          0.0
 0.0  -0.000390625  0.0
 0.0   0.0          0.0
```

PLU Decomposition

PLU always exists, and is much better accuracy wise. The matrix P is given by a single vector. For example, if the permutation is given in Cauchy's notation as

$$\begin{pmatrix} 1 & 2 & \cdots & n \\ \sigma_1 & \sigma_2 & \cdots & \sigma_n \end{pmatrix}$$

then Julia returns a vector `p` containing $[\sigma_1, \sigma_2, \ldots, \sigma_n]$.

We can convert this to a permutation matrix via

$$P = [\mathbf{e}_{\sigma_1} | \cdots | \mathbf{e}_{\sigma_n}].$$

In Julia, this can be done by creating a zero matrix `P`, and putting a 1 in the `P`$[\sigma_k, k]$ entry:

In [132]:

```
A=rand(n,n)

L,U,σ=lu(A)

n=3
# simplest way P=eye(n)[:,p]

P=zeros(Int,n,n)
for k=1:n
    P[σ[k],k]=1
end



norm(A-P*L*U)
```

Out[132]:

```
1.1102230246251565e-16
```

In [134]:

```
A=rand(100,100)
L,U,σ=lu(A)
n=size(A,1)
P=eye(n)[:,σ]

norm(A-P*L*U)
```

Out[134]:

```
4.485808827113607e-15
```

Having a PLU Decomposition allows us to invert matrices:

In [136]:

```
norm(inv(U)*inv(L)*P'-inv(A))
```

Out[136]:

```
7.800972055143745e-14
```

QR Decomposition

A QR decomposition decomposes a matrix into an orthogonal matrix Q times an upper triangular matrix R. Again, if $A = QR$ then $A^{-1} = R^{-1} Q^\top$ is computable on a computer.

We can obtain the QR decompostion by calling `qr`:

In [137]:

```
A=[1E-9 2   3;
   4    6.9 10;
   10   52  3]
Q,R=qr(A)
# R is upper triangular
norm(Q'*Q-eye(3))   # Q is an orthogonal matrix

norm(A-Q*R)

norm(inv(A)-inv(R)*Q')
```

Out[137]:

```
9.901609419194922e-16
```

Here is a 100 x 100 example:

In [139]:

```
A=rand(100,100)+2eye(100)
Q,R=qr(A)
# R is upper triangular
n=size(A,1)
norm(Q'*Q-eye(n))   # Q is an orthogonal matrix

norm(A-Q*R)

norm(inv(A)-inv(R)*Q')
```

Out[139]:

```
2.0516294456721667e-13
```