This file contains some short functions in Julia from lectures and assignmnets

LU Decomposition

In [1]:
```
function getrfOuter!(A)
    n = size(A,1)
    for k = 1:n
        for i = k+1:n
            A[i,k] /= A[k,k]
        end
        for i = k+1:n
            for j = k+1:n
                A[i,j] -= A[i,k] * A[k,j]
            end
        end
    end
end
```

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

In the function above, A[k,k] should not be too small (making the whole term large); it should not be smaller than any of A[i,k] s.t. the division works well.

LU with Pivoting

In [2]:
```
function getrf!(A)
    n = size(A,1)
    for k = 1:n
        kmx = k - 1 + argmax(abs.(A[k:end,k]))
        for j = 1:n
            A[k,j], A[kmx,j] = A[kmx,j], A[k,j]
        end
        for i = k+1:n
            A[i,k] /= A[k,k]
        end
        for j = k+1:n, i = k+1:n
            A[i,j] -= A[i,k] * A[k,j]
        end
    end
end
```

Out[2]:
```
getrf! (generic function with 1 method)
```

Gauss-Jordan

This problem explores an extension of the LU factorization called the Gauss-Jordan transformation. This procedure involves more flops than Gaussian elimination but provides more parallelism on parallel computers and so, for some hardware, may actually perform better than the Gaussian elimination.

Matrices in $\mathbb{R}^{n\times n}$ of the form $N(y,k) = I - ye_k^T$ are called Gauss-Jordan transformations.

Write a Julia code that uses Gauss-Jordan transformations and overwrites A with A^{-1}. Run your code with matrix A provided below, and report the computed values

$$(A^{-1})_{1,1},\ (A^{-1})_{1,6},\ (A^{-1})_{6,1} \text{ and } (A^{-1})_{6,6}.$$

In [3]:
```
function simple_rand(state)
    rand_max = 2^32; a = 11-3515245; c = 54321
    state = (a*state+c)%rand_max
    return state
end
```

Out[3]:
```
simple_rand (generic function with 1 method)
```

In [4]:
```
n = 6
A = zeros(n,n)
state = 2018

for j = 1:n
    for i = 1:n
        state = simple_rand(state)
        A[i,j] = state / 2^32
    end
end

display(A)
```

```
6×6 Array{Float64,2}:
 -0.651628  -0.482565  -0.705652  -0.355389  -0.879925   -0.700237
  0.611103   0.618275   0.660996   0.160847   0.409138    0.799763
 -0.772449  -0.748267  -0.871555  -0.898968  -0.918987   -0.200237
  0.589116   0.716795   0.848698   0.900349   0.237263    0.799763
 -0.260389  -0.277687  -0.952892  -0.307659  -0.0752373  -0.200237
  0.59456    0.288844   0.349018   0.827106   0.549763    0.799763
```

In [5]:
```
x = zeros(n,1)
y = zeros(n,1)
identity = zeros(n,n)
for i = 1:n
    identity[i,i] = 1
end

for i = 1:n
    x = A[:,i]
    for j = 1:n
        if j!= i
            y[j] = x[j]/x[i]
        else
            y[j] = 1 - 1/x[i]
        end
    end
    e_i = zeros(n,1)
    e_i[i] = 1
    N = identity - y*transpose(e_i)

    A = N*A
    A[:,i] = N*e_i
end

display(A)
```

```
6×6 Array{Float64,2}:
  6.94971    6.08415   -2.45942    -5.89024    -2.19161    4.72647
 -1.42857   -0.658267  -0.143833    2.92574     2.24247   -2.99282
 -0.777332  -0.626518   0.389444    0.0283801  -1.35837   -0.325052
 -0.187689  -1.47181   -0.0464306   1.05975     0.210123   0.288708
 -3.36434   -2.3542     0.457706    1.203       0.98636   -1.43292
 -1.80461   -0.871507   1.44377     1.38694     0.516852  -0.354212
```

Power Iteration (pseudo)

In [ ]:
```
while not_converged
    zk = A*qk
    eval = dot(zk,qk)
    qk = zk / norm(zk)
end
```

Inverse Iteration (pseudo)

In [ ]:
```
while not_converged
    zk = (A - mu * I) \ qk
    qk = zk / norm(zk)
    eval = dot(qk, A*qk)
end
```

Orthogonal iteration (pseudo)

In [ ]:
```
while not_converged
    Qk = A*Qk
    Qk,Rk = qr(Qk)
end
```

QR with shift (pseudo)

In [ ]:
```
Tk = A
while not_converged
    mu = Tk[n,n]
    Uk, Rk = qr(Tk - mu*I)
    Tk = Rk * Uk + mu * I
end
```

In [ ]: