hw 3 solution

Statistical Computing, Jieun Shin

Autumn 2022

문제 1.

(a) 가우스 소거법으로 해를 구하고 `solve`함수에 의해 구한 해와 비교해보자.

1. 먼저 행렬 A와 벡터 b를 정의한다.

```
#  정의
set.seed(123)
A = matrix(c(8, -4, 4,
             -6, 11, -7,
             2, -7, 6),
           3, 3)
b = matrix(c(28, -40, 33), 3, 1)
```

2. 가우스 소거법과 삼각행렬을 풀기위한 후방대입법 함수를 정의한다 (강의안 참고).

```
gaussianelimination = function(Ab){
  n = nrow(Ab)
  for (k in (1:(n-1))){
    for (i in ((k+1):n)){
      mik = Ab[i,k]/Ab[k,k]
      Ab[i,k]=0

      for (j in ((k+1):(n+1))){
        Ab[i,j] = Ab[i,j] - mik*Ab[k,j]
      }
    }
  }
  return(Ab)
}

backwardsub = function(U,b){
  x = c(0)
  n = nrow(U)
  for (i in (n:1)){
    x[i] = b[i]
      if (i < n){
        for (j in ((i+1):n)){
          x[i] = x[i] - U[i,j]*x[j]
        }
      }

    x[i] = x[i]/U[i,i]
    }
```

```
  return(cbind(x))
}
```

3. 선형방정식 $Ax = b$의 해를 가우스 소거법으로 구한 해와 solve함수로 구한 해와 비교한다. 두 방법으로 구한 해가 같음을 확인할 수 있다.

```
# 가우스 소거법
Ab = cbind(A, b)
ge = gaussianelimination(Ab)
backwardsub(ge[, 1:3], ge[, 4])
```

```
##       x
## [1,]  2
## [2,] -1
## [3,]  3
```

```
# solve
solve(A) %*% b
```

```
##      [,1]
## [1,]    2
## [2,]   -1
## [3,]    3
```

(b). 행렬 A에 대한 LU분해를 구하고 LU분해에 기반하여 해를 구해보자. 먼저 전방대입법을 사용하기 위한 함수를 지정해주고 lu함수를 사용하여 각 L과 U에 해당하는 행렬을 구한다. 그리고 순차적으로 전방대입법으로 $L\boldsymbol{y} = \boldsymbol{b}$와 후방대입법으로 $U\boldsymbol{x} = \boldsymbol{y}$를 풀어 해를 구한다. 결론적으로 $\boldsymbol{x} = (2, 1, 3)^T$을 잘 구한 것을 확인하였다.

```
# 전방 대입법
forwardsub = function(L,b){
    x = c(0)
    n = nrow(L)
    for (i in (1:n)){
        x[i] = b[i]
        if (i > 1){
            for (j in (1:(i-1))){
                x[i] = x[i] - L[i,j]*x[j]
            }
        }
        x[i] = x[i]/L[i,i]
    }
    return(cbind(x))
}

library(Matrix)
```

```
## Warning: 패키지 'Matrix'는 R 버전 4.2.2에서 작성되었습니다
```

```
lum = lu(A)
L = expand(lum)$L
U = expand(lum)$U

# 전방대입법, Ly = b
y = forwardsub(L, b)

# 후방대입법, Ux = y
backwardsub(U, y)
```

```
##        x
## [1,]  2
## [2,] -1
## [3,]  3
```

문제 2.

R 예시

1. 문제에서 정의한 행렬 A를 정의한다.

```
set.seed(123)
A = matrix(c(1, 1, 2,
             2, 3, 5,
             3, 4, 7),
           3, 3)
```

2. svd함수를 이용하여 일반화 역행렬을 구하는 함수 ginvsvd를 정의한다.

```
ginvsvd = function(A){
  svd = svd(A)  # 특이값 분해
  u = svd$u
  s = svd$d
  v = svd$v

  dim = dim(A)
  d = min(dim)
  S = diag(1/s, d, d)    # S+는 S의 대각원소 중 0이 아닌 것을 역수로 취한 행렬

  invA = v %*% S %*% t(u)    # A+ = V S+ U^t

  out = list()
  out$u = u
  out$S = S
  out$v = v
  out$invA = invA
  return(out)

}
```

3. ginvsvd와 limSolve의 Solve함수와 비교하여 두 방법이 계산한 역행렬이 동일한 것을 확인하였다.

```
ginvsvd(A)
```

```
## $u
##            [,1]        [,2]       [,3]
## [1,] -0.3440436  0.74047328 -0.5773503
## [2,] -0.4692469 -0.66818713 -0.5773503
## [3,] -0.8132905  0.07228615  0.5773503
##
## $S
##            [,1]     [,2]         [,3]
## [1,] 0.09208725 0.000000 0.000000e+00
## [2,] 0.00000000 3.619756 0.000000e+00
## [3,] 0.00000000 0.000000 2.711197e+15
##
## $v
```

```
##             [,1]       [,2]       [,3]
## [1,] -0.2246810  0.7849746  0.5773503
## [2,] -0.5674674 -0.5870668  0.5773503
## [3,] -0.7921485  0.1979078 -0.5773503
##
## $invA
##               [,1]          [,2]          [,3]
## [1,] -9.037323e+14 -9.037323e+14  9.037323e+14
## [2,] -9.037323e+14 -9.037323e+14  9.037323e+14
## [3,]  9.037323e+14  9.037323e+14 -9.037323e+14
```

```
limSolve::Solve(A)
```

```
##            [,1]       [,2]       [,3]
## [1,]  2.1111111 -1.8888889  0.2222222
## [2,] -1.5555556  1.4444444 -0.1111111
## [3,]  0.5555556 -0.4444444  0.1111111
```

파이썬 예시

```
import numpy as np
A = np.array([[1,2,3],
             [2,3,5],
             [3,4,7]])
A

# 함수 지정
```

```
## array([[1, 2, 3],
##        [2, 3, 5],
##        [3, 4, 7]])
```

```
def ginsvdv(A) :
  svd = np.linalg.svd(A)
  u = svd[0]
  s = svd[1]
  v = svd[2]

  dim = A.shape
  d = min(dim)
  S = np.diag(1/s)

  invA = np.matmul(np.matmul(v, S), np.transpose(u))
  return u, s, v, invA

# 실행
ginsvdv(A)[0] # u
```

```
## array([[-0.33228884,  0.85024553,  0.40824829],
##        [-0.54944908,  0.17731059, -0.81649658],
##        [-0.76660931, -0.49562435,  0.40824829]])
```

```
ginsvdv(A)[1] # s
```

```
## array([1.12185998e+01, 3.78179165e-01, 1.61978941e-16])
```

```
ginsvdv(A)[2] # v
```

```
## array([[-0.33257403, -0.47950388, -0.81207791],
##        [-0.7456951 ,  0.66086511, -0.08482999],
##        [ 0.57735027,  0.57735027, -0.57735027]])
```

```
ginsvdv(A)[3] # invA
```

```
## array([[-2.04674396e+15,  4.09348792e+15, -2.04674396e+15],
##        [-2.13803712e+14,  4.27607423e+14, -2.13803712e+14],
##        [-1.45514138e+15,  2.91028276e+15, -1.45514138e+15]])
```

문제 3.

R 예시

1. Gram-Schmidt 직교화 알고리즘을 구현한 함수를 정의한다.

```
norm = function(x) sqrt(sum(x^2))

GramSchmidt = function(A){
  dim = dim(A)
  n = dim[2]
  p = dim[1]

  # define the normal orthogonal basis q
  q = matrix(0, nrow = p, ncol = n)
  q[,1] = A[, 1] / norm(A[, 1])      # q_1 = x_1 /norm(x_1)

  for( i in 2:n ){

    val = 0
    for( j in 1:(i-1) ){
    v = t(A[, i]) %*% q[, j]
    v = c(v) * q[, j]

    val = val + v                # sum_{j=1}^{i-1} [ t(x_i) %*% q_j ] * q_j
    }

    qval = A[, i] - val
    q[, i] = qval / norm(qval)   # q_i = q_i / norm(q_i)
  }

  return(q)
}
```

3. 위에서 정의한 `GramSchmidt`와 `qr.Q`로부터 구한 직교행렬이 같음을 확인하였다.

```
A = matrix(c(8, -4, 4,
             -6, 11, -7,
             2, -7, 6),
           3, 3)
GramSchmidt(A)
```

```
##            [,1]       [,2]       [,3]
## [1,]  0.8164966  0.5345225 -0.2182179
## [2,] -0.4082483  0.8017837  0.4364358
## [3,]  0.4082483 -0.2672612  0.8728716
```

```
qr = qr(A)
qr.Q(qr)
```

```
##            [,1]       [,2]       [,3]
## [1,] -0.8164966 -0.5345225 -0.2182179
## [2,]  0.4082483 -0.8017837  0.4364358
## [3,] -0.4082483  0.2672612  0.8728716
```

파이썬 예시

R코드와 동일하게 함수를 정의하고 실행하면 동일한 결과가 나온다. `np.linalg.qr`을 사용해도 결과가 같다.

```
# 행렬 정의
A = np.array([[8,-6, 2],
             [-4, 11, -7],
             [4, -7, 6]])
A

# 함수 정의
```

```
## array([[ 8, -6,  2],
##        [-4, 11, -7],
##        [ 4, -7,  6]])
```

```
def norm(x) :
  return np.sqrt(np.sum(np.power(x, 2)))

def GramSchmidt(A) :
  dim = A.shape
  n = dim[0]
  p = dim[1]

  # define the normal orthogonal basis q
  q = np.zeros([3,3])             # make 3 by 3 zero matrix
  q[:,0] = A[:,0] / norm(A[:,0]) # q_1 = x_1 /norm(x_1)

  for i in range(1,n):
    val = 0
    for j in range(i):
      v = np.matmul(np.transpose(A[:,i]), q[:,j])
      v = v * q[:,j]
      val = val + v     # sum_{j=1}^{i-1} [ t(x_i) %*% q_j ] * q_j
    qval = A[:,i] - val
    q[:,i] = qval / norm(qval)   # q_i = q_i / norm(q_i)

  return q

# 실행
GramSchmidt(A)     # 정의한 함수 결과
```

```
## array([[ 0.81649658,  0.53452248, -0.21821789],
##        [-0.40824829,  0.80178373,  0.43643578],
##        [ 0.40824829, -0.26726124,  0.87287156]])
```

```
np.linalg.qr(A)[0] # 내장함수 결과
```

```
## array([[-0.81649658, -0.53452248, -0.21821789],
```

```
##         [ 0.40824829, -0.80178373,  0.43643578],
##         [-0.40824829,  0.26726124,  0.87287156]])
```