MOORE PENROSE PSEUDOINVERSE

INTRODUCTION

In mathematics, and in particular linear algebra, a **pseudoinverse** A^+ of a matrix A is a generation of the inverse matrix. The most widely known type of matrix pseudoinverse is the **Moore–Penrose pseudoinverse**, which was independently described by E. H. Moore in 1920, Arne Bjerhammar in 1951 and Roger Penrose in 1955**.**
The Moore-Penrose pseudoinverse, as is shown in what follows, it brings great notational and conceptual clarity to the study of solutions to arbitrary systems of linear equations and linear least squares problems.
The pseudoinverse is defined and unique for all matrices whose entries are real or complex numbers. It can be computed using the single value decomposition (SVD).

A matrix inverse A^{-1} is defined as a matrix that produces identity matrix when we multiply with the original matrix A that is we define $AA^{-1} = A^{-1}A = I$. Matrix inverse exists only for square matrices.

Real world data is not always square. Furthermore, real world data is not always consistent and might contain many repetitions. To deal with real world data, generalized inverse for rectangular matrix is needed.

Generalized inverse matrix is defined as $AA^+A = A$. Notice that the usual matrix inverse is covered by this definition because $AA^{-1}A = A$. We use term “generalized” inverse for a general rectangular matrix and to distinguish from inverse matrix that is for a square matrix. Generalized inverse is also called pseudo inverse.
Unfortunately there are many types of generalized inverse. Most of generalized inverse are not unique. Some of generalized inverse are reflexive $(A^+)^+ = A$ and some are not reflexive.

DEFINITION

For $A \in M(m, n; \mathbf{K})$, a Moore–Penrose pseudoinverse (hereafter, just pseudoinverse) of A is defined as a matrix $A^+ \in M(m, n; \mathbf{K})$, Satisfying all of the following four criteria:

1. $AA^+A = A$ (AA^+ need not be the general identity matrix, but it maps all column vectors of A to themselves);
2. $A^+AA^+ = A^+$ (A^+ is a weak inverse for the multiplicative semi group);
3. $(AA^+)^* = AA^+$ (AA^+ is Hermitian); and
4. $(A^+A)^* = A^+A$ (A^+A is also Hermitian).

Where, the following conventions are adopted.

- $\mathbf{K}$ will denote one of the fields of real or complex numbers, denoted $\mathbf{R}, \mathbf{C}$, respectively. The ring of $m \times n$ matrices over is denoted by $M(m, n; \mathbf{K})$.
- For $A \in M(m, n; \mathbf{K})$, A^T and A^* denote the transpose and Hermitian transpose (also called conjugate transpose) respectively. If, $\mathbf{K} = \mathbf{R}$ then $A^* = A^T$.
- For $A \in M(m, n; \mathbf{K})$, then $Im(A)$ denotes the range (image) of A (the space spanned by the column vectors of A) $Ker(A)$ and denotes the kernel (null space) of A.
- Finally, for any positive integer n, $I_n \in M(m, n; \mathbf{K})$ denotes the $n \times n$ identity matrix.

METHODOLOGY

SVD is a factorization of a rectangular matrix A into three matrices U, D and V. The two matrices U and V are orthogonal matrices $(U^T=U^{-1}, VV^T=I)$ while D is a diagonal matrix. The factorization means that we can multiply the three matrices to get back the original matrix

$$A=UDV^T$$

The transpose matrix is obtained through $A^T=VDU^T$ .Since both the orthogonal matrix and diagonal matrix have many nice properties, SVD is one of the most powerful matrix decomposition that used on many applications such as least square, image restoration and 3 D computer vision and many more.

Matrices U and V are not unique, there columns come from the concatenation of eigenvectors of symmetric matrices AA^T and A^TA. Since eigenvectors of symmetric matrix are orthogonal (and linearly independent), they can be used as basis vector (coordinate system) to span a multidimensional space. The absolute value of the determinant of orthogonal matrix is one, thus the matrix always has inverse. Furthermore, each column (and each row) of orthogonal matrix has unit norm.

The diagonal matrix D contains the square of eigenvalues of symmetric matrix A^TA. The diagonal elements are non-negative numbers and they are called singular values. Because they come from a symmetric matrix, the eigenvalues (and the eigenvectors) are all real numbers (no complex numbers).
Numerical computation of SVD is stable in term of round off error. When some of the singular values are nearly zero, we can truncate them as zero and it yields numerical stability.

Since the SVD factor matrix $A=UDV^T$, The diagonal matrix can also be obtained from $D=U^TAV$.

The eigenvalues are actually the many solutions of homogeneous equation. They are not unique and correct up to a scalar multiple. Thus, we can multiply an eigenvector with -1 and still get the same correct result.

PROPERTIES

EXISTENCE AND UNIQUENESS

- The Moore–Penrose pseudoinverse exists and is unique: for any matrix A, there is precisely one matrix A^+, that satisfies the four properties of the definition.

A matrix satisfying the first two conditions of the definition is known as a generalized inverses always exist but are not in general unique.
Uniqueness is a consequence of the last two conditions.

BASIC PROPERTIES

1. If A has real entries, then so does A^+.
2. If A is invertible, its pseudoinverse is its inverse. That is: $A^+=A^{-1}$
3. The pseudoinverse of a zero matrix is its transpose.
4. The pseudoinverse of the pseudoinverse is the original matrix:

   $(A^+)^+=A$.

5. Pseudo inversion commutes with transposition, conjugation, and taking the conjugate transpose: $(A^T)^+= (A^+)^T$, $(A^*)^+=(A^+)^*$

IDENTITIES

The following identities can be used to cancel certain subexpressions or expand expressions involving pseudoinverse.

$$
\begin{aligned}
A^+ &= A^+ \quad A^{+*} \quad A^* \\
A^+ &= A^* \quad A^{+*} \quad A^+ \\
A &= A^{+*} \quad A^* \quad A \\
A &= A \quad A^* \quad A^{+*} \\
A^* &= A^* \quad A \quad A^+ \\
A^* &= A^+ \quad A \quad A^*
\end{aligned}
$$

Reduction to Hermitian case

$$A^+ = A^*(AA^*)^+$$
$$A^+ = (A^*A)^+A^*$$

SINGULAR VALUE DECOMPOSITION (SVD)

A computationally simple and accurate way to compute the pseudoinverse is by using the singular value decomposition. If $A = U\Sigma V^*$ is the singular value decomposition of A, then $A^+ = V\Sigma^+U^*$. For a diagonal matrix such as Σ, we get the pseudoinverse by taking the reciprocal of each non-zero element on the diagonal, leaving the zeroes in place, and transposing the resulting matrix. In numerical computation, only elements larger than some small tolerance are taken to be nonzero, and the others are replaced by zeroes. For example, in the MATLAB or NumPy function pinv, the tolerance is taken to be $t = \varepsilon \bullet max(m, n) \bullet max(\Sigma)$, where ε is the machine epsilon.

The computational cost of this method is dominated by the cost of computing the SVD, which is several times higher than matrix-matrix multiplication, even if a state-of-the art implementation (such as that of LAPACK) is used.

The above procedure shows why taking the pseudoinverse is not a continuous operation: if the original matrix A has a singular value 0 (a diagonal entry of the matrix Σ above), then modifying A slightly may turn this zero into a tiny positive number, thereby affecting the pseudoinverse dramatically as we now have to take the reciprocal of a tiny number.

Moore-Pen o e pseudoinverse

Use the SVD (or URV) decomposition to define an "inverse" for a singular matrix.

$$A = U\Sigma V^* = U \left[\begin{array}{c|c} \begin{matrix}\sigma_1 & & 0 \\ & \ddots & \\ 0 & & \sigma_r\end{matrix} & \begin{matrix}0\end{matrix} \\ \hline \begin{matrix}0\end{matrix} & \begin{matrix}0\end{matrix}\end{array}\right] V^*.$$

Define $A^\dagger = V\Sigma^\dagger U^* = V \left[\begin{array}{c|c} \begin{matrix}1/\sigma_1 & & 0 \\ & \ddots & \\ 0 & & 1/\sigma\end{matrix} & \\ \hline \begin{matrix}0\end{matrix} & \end{array}\right] U^*.$ (Invert the "invertible" part)

Now $A^\dagger A = V\Sigma^\dagger U^* U \Sigma V^* = V \left[\begin{array}{c|c} \begin{matrix}1/\sigma_1 & & 0 \\ & \ddots & \\ 0 & & 1/\sigma_r\end{matrix} & \begin{matrix}0\end{matrix} \\ \hline \begin{matrix}0\end{matrix} & \begin{matrix}0\end{matrix}\end{array}\right] U^*U \left[\begin{array}{c|c} \begin{matrix}\sigma_1 & & 0 \\ & \ddots & \\ 0 & & \sigma_r\end{matrix} & \\ \hline \begin{matrix}0\end{matrix} & \end{array}\right] V^*$

$$= V \begin{bmatrix} I & 0 \\ 0 & 0 \end{bmatrix} V^*.$$

Similarly, $AA^\dagger = U\Sigma V^* V\Sigma^\dagger U^* = U \begin{bmatrix} I & 0 \\ 0 & 0 \end{bmatrix} U^*.$

ALGORITHM

```
moorePenrose (A, m, n, B)

INPUT: A (m x n matrix), B (where System of Linear
       Equations are of from AX = B)
OUTPUT: X (n x 1 matrix, solution of System of
       System of Linear Equations AX = B)

svd (A, m, n, V*, Σ)   /* Decomposes matrix A
                          into form UΣV* and over-
                          writes value of U in A
                          and V* in V* and Σ in Σ
                       */
FOR i = 1 to m
    FOR j = 1 to n
        IF (i==j)
            Σ[i][j] = (1/ Σ[i][j])
        END IF
    END FOR
END FOR

Σ+ = transpose (Σ)     // Returns transpose of Σ
V = transpose (V*)     // Returns transpose of V*
U* = transpose (U)     // Returns transpose of U

A+ = V Σ+ U*           /* A+ is Moore Penrose
                          Inverse of A
                       */

X = A+ B               /* X is the required
                          Solution.
                       */

END
```

C CODE

1. Moore penrose.c

```
#include<stdio.h>
#include<stdlib.h>
#include "svd.c"

void main()
{
    int m,n;
    printf("\nEnter size of Matrix: \n");
    scanf("%d %d",&m,&n);
    float **a, **v, *w, **d, *b, **pinv,
    **temp_pinv, **x,temp,**ut,**vtt,**dt;
    a=malloc(m*sizeof(float*));
    b=malloc(m*sizeof(float));
    v=malloc(n*sizeof(float*));
    w=malloc(n*sizeof(float));
    d=malloc(m*sizeof(float*));
    x=malloc(n*sizeof(float*));
    ut=malloc(m*sizeof(float*));
    vtt=malloc(n*sizeof(float*));
    pinv=malloc(n*sizeof(float*));
    temp_pinv=malloc(n*sizeof(float*));
    dt=malloc(n*sizeof(float*));

    int i,j,k;
    for(i=0;i<m;i++)
    {
        a[i]=malloc(n*sizeof(float));

    }
    for(i=0;i<m;i++)
    {
        ut[i]=malloc(m*sizeof(float));

    }
```

```
  for(i=0;i<n;i++)
  {
      v[i]=malloc(n*sizeof(float));

  }
  for(i=0;i<n;i++)
  {
      vtt[i]=malloc(n*sizeof(float));

  }
  for(i=0;i<m;i++)
  {
      d[i]=malloc(n*sizeof(float));
  }
  for(i=0;i<m;i++)
  {
      dt[i]=malloc(m*sizeof(float));
  }
for(i=0;i<n;i++)
  {
      x[i]=malloc(1*sizeof(float));

  }
for(i=0;i<n;i++)
  {
      pinv[i]=malloc(m*sizeof(float));
  }
for(i=0;i<n;i++)
  {
      temp_pinv[i]=malloc(m*sizeof(float));
  }

  printf("Equation form:AX = B\n");
  printf("Enter Matrix A %d X %d\n",m,n);
  for(i=0;i<m;i++)
  {
      for(j=0;j<n;j++)
      {
          scanf("%f",&a[i][j]);
      }
  }
```

```
printf("Enter Matrix B: \n");
for(i=0;i<m;i++)
    {
        scanf("%f",&b[i]);
    }

printf("\n A is\n");
for(i=0;i<m;i++)
{
    for(j=0;j<n;j++)
    {
        printf("%f ",a[i][j]);
    }
    printf("\n");
}
dsvd(a,m,n,w,v);
printf("\n Singular Vector is\n");
for(i=0;i<n;i++)
{
    printf("%f ",w[i]);
}
printf("\n");

printf("\n U is\n");

for(i=0;i<m;i++)
{
    for(j=0;j<m;j++)
    {
        printf("%f ",a[i][j]);
    }
    printf("\n");
}

printf("\n V is\n");

for(i=0;i<n;i++)
{
    for(j=0;j<n;j++)
    {
        printf("%f ",v[i][j]);
    }
    printf("\n");
```

```
}
printf("\n");

for(i=0;i<m;i++){
  for (j=0;j<n;j++){
    if (i==j){
        d[i][j]=w[i];
    }
    else d[i][j] = 0;
    }
}

//Reciprocal of Sigma

for(i=0;i<m;i++){
  for(j=0;j<n;j++){
    if(d[i][j]!=0){
        d[i][j]=(1/d[i][j]);
    }
  }
}

//Transpose Of Sigma

for(i=0;i<n;i++){
   for(j=0;j<m;j++){
      dt[i][j]=d[j][i];
   }
}

//U transpose
for(i=0;i<m;i++){
    for(j=0;j<m;j++){
        ut[i][j]=a[j][i];
    }
}

// V
for(i=0;i<n;i++){
    for(j=0;j<n;j++){
        vtt[i][j]=v[j][i];
    }
}
```

```
    printf("\n Diagonal Matrix is \n");
    //Diagonal Matrix Formed.
    for(i=0;i<m;i++){
        for(j=0;j<n;j++){
            printf("%f ",dt[i][j]);
        }
        printf("\n");
    }
    printf("\n");

    // Now A-pseudoinverse = V.(diagonal
matrix).(U Transpose)
     for (i=0;i<n;i++){
        for(j=0;j<n;j++){
            for(k=0;k<n;k++){

temp_pinv[i][j]+=vtt[i][k]*dt[k][j];
            }
        }
    }

     for (i=0;i<n;i++){
        for(j=0;j<m;j++){
            for(k=0;k<m;k++){

pinv[i][j]+=temp_pinv[i][k]*ut[k][j];
            }
        }
    }

    //pinv formed of size n X m
    // Our Equation was Ax=b,now x=pinv.b so we
take input b matrix also.(constant's matrix) of
size m X 1
    //final answer in x matrix of size n X 1

    for (i=0;i<n;i++){
        for(j=0;j<1;j++){
            for(k=0;k<m;k++){
                x[i][j]+=pinv[i][k]*b[k];
            }
        }
```

```
    }

    printf("\n Solution of System of Linear
Equations is: \n");
        for(i=0;i<n;i++)
        {
            for(j=0;j<1;j++)
            {
                printf("%f ",x[i][j]);
            }
            printf("\n");
        }
}
```

2. svd.c

```
/*
 * svdcomp - SVD decomposition routine.
 * Takes an mxn matrix a and decomposes it into
udv, where u,v are
 * left and right orthogonal transformation
matrices, and d is a
 * diagonal matrix of singular values.
 * Input to dsvd is as follows:
 *   a = mxn matrix to be decomposed, gets
overwritten with u
 *   m = row dimension of a
 *   n = column dimension of a
 *   w = returns the vector of singular values of
a
 *   v = returns the right orthogonal
transformation matrix
*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "defs_and_types.h"

static double PYTHAG(double a, double b)
{
    double at = fabs(a), bt = fabs(b), ct, result;
```

```
    if (at > bt)       { ct = bt / at; result = at
* sqrt(1.0 + ct * ct); }
    else if (bt > 0.0) { ct = at / bt; result = bt
* sqrt(1.0 + ct * ct); }
    else result = 0.0;
    return(result);
}


int dsvd(float **a, int m, int n, float *w, float
**v)
{
    int flag, i, its, j, jj, k, l, nm;
    double c, f, h, s, x, y, z;
    double anorm = 0.0, g = 0.0, scale = 0.0;
    double *rv1;

    if (m < n)
    {
        fprintf(stderr, "#rows must be > #cols
\n"); // No. of Equations > no. of unknowns
        return(0);
    }

    rv1 = (double *)malloc((unsigned int)
n*sizeof(double));

/* Householder reduction to bidiagonal form */
    for (i = 0; i < n; i++)
    {
        /* left-hand reduction */
        l = i + 1;
        rv1[i] = scale * g;
        g = s = scale = 0.0;
        if (i < m)
        {
            for (k = i; k < m; k++)
                scale += fabs((double)a[k][i]);
            if (scale)
            {
                for (k = i; k < m; k++)
                {
```

```
                    a[k][i] =
(float)((double)a[k][i]/scale);
                    s += ((double)a[k][i] *
(double)a[k][i]);
                }
                f = (double)a[i][i];
                g = -SIGN(sqrt(s), f);
                h = f * g - s;
                a[i][i] = (float)(f - g);
                if (i != n - 1)
                {
                    for (j = l; j < n; j++)
                    {
                        for (s = 0.0, k = i; k <
m; k++)
                            s += ((double)a[k][i]
* (double)a[k][j]);
                        f = s / h;
                        for (k = i; k < m; k++)
                            a[k][j] += (float)(f *
(double)a[k][i]);
                    }
                }
                for (k = i; k < m; k++)
                    a[k][i] =
(float)((double)a[k][i]*scale);
            }
        }
        w[i] = (float)(scale * g);

        /* right-hand reduction */
        g = s = scale = 0.0;
        if (i < m && i != n - 1)
        {
            for (k = l; k < n; k++)
                scale += fabs((double)a[i][k]);
            if (scale)
            {
                for (k = l; k < n; k++)
                {
                    a[i][k] =
(float)((double)a[i][k]/scale);
                    s += ((double)a[i][k] *
```

```
(double)a[i][k]);
                }
                f = (double)a[i][l];
                g = -SIGN(sqrt(s), f);
                h = f * g - s;
                a[i][l] = (float)(f - g);
                for (k = l; k < n; k++)
                    rv1[k] = (double)a[i][k] / h;
                if (i != m - 1)
                {
                    for (j = l; j < m; j++)
                    {
                        for (s = 0.0, k = l; k <
n; k++)
                            s += ((double)a[j][k]
* (double)a[i][k]);
                        for (k = l; k < n; k++)
                            a[j][k] += (float)(s *
rv1[k]);
                    }
                }
                for (k = l; k < n; k++)
                    a[i][k] =
(float)((double)a[i][k]*scale);
            }
        }
        anorm = MAX(anorm, (fabs((double)w[i]) +
fabs(rv1[i])));
    }

    /* accumulate the right-hand transformation */
    for (i = n - 1; i >= 0; i--)
    {
        if (i < n - 1)
        {
            if (g)
            {
                for (j = l; j < n; j++)
                    v[j][i] =
(float)(((double)a[i][j] / (double)a[i][l]) / g);
                    /* double division to avoid
underflow */
                for (j = l; j < n; j++)
```

```
                {
                    for (s = 0.0, k = l; k < n;
k++)
                        s += ((double)a[i][k] *
(double)v[k][j]);
                    for (k = l; k < n; k++)
                        v[k][j] += (float)(s *
(double)v[k][i]);
                }
            }
            for (j = l; j < n; j++)
                v[i][j] = v[j][i] = 0.0;
        }
        v[i][i] = 1.0;
        g = rv1[i];
        l = i;
    }

    /* accumulate the left-hand transformation */
    for (i = n - 1; i >= 0; i--)
    {
        l = i + 1;
        g = (double)w[i];
        if (i < n - 1)
            for (j = l; j < n; j++)
                a[i][j] = 0.0;
        if (g)
        {
            g = 1.0 / g;
            if (i != n - 1)
            {
                for (j = l; j < n; j++)
                {
                    for (s = 0.0, k = l; k < m;
k++)
                        s += ((double)a[k][i] *
(double)a[k][j]);
                    f = (s / (double)a[i][i]) * g;
                    for (k = i; k < m; k++)
                        a[k][j] += (float)(f *
(double)a[k][i]);
                }
            }
```

```
            for (j = i; j < m; j++)
                a[j][i] =
(float)((double)a[j][i]*g);
        }
        else
        {
            for (j = i; j < m; j++)
                a[j][i] = 0.0;
        }
        ++a[i][i];
    }

    /* diagonalize the bidiagonal form */
    for (k = n - 1; k >= 0; k--)
    {                             /* loop over
singular values */
        for (its = 0; its < 30; its++)
        {                         /* loop over
allowed iterations */
            flag = 1;
            for (l = k; l >= 0; l--)
            {                     /* test for
splitting */
                nm = l - 1;
                if (fabs(rv1[l]) + anorm == anorm)
                {
                    flag = 0;
                    break;
                }
                if (fabs((double)w[nm]) + anorm ==
anorm)
                    break;
            }
            if (flag)
            {
                c = 0.0;
                s = 1.0;
                for (i = l; i <= k; i++)
                {
                    f = s * rv1[i];
                    if (fabs(f) + anorm != anorm)
                    {
                        g = (double)w[i];
```

```
                        h = PYTHAG(f, g);
                        w[i] = (float)h;
                        h = 1.0 / h;
                        c = g * h;
                        s = (- f * h);
                        for (j = 0; j < m; j++)
                        {
                            y = (double)a[j][nm];
                            z = (double)a[j][i];
                            a[j][nm] = (float)(y *
c + z * s);
                            a[j][i] = (float)(z *
c - y * s);
                        }
                    }
                }
            }
            z = (double)w[k];
            if (l == k)
            {                   /* convergence */
                if (z < 0.0)
                {               /* make singular
value nonnegative */
                    w[k] = (float)(-z);
                    for (j = 0; j < n; j++)
                        v[j][k] = (-v[j][k]);
                }
                break;
            }
            if (its >= 30) {
                free((void*) rv1);
                fprintf(stderr, "No convergence
after 30,000! iterations \n");
                return(0);
            }

            /* shift from bottom 2 x 2 minor */
            x = (double)w[l];
            nm = k - 1;
            y = (double)w[nm];
            g = rv1[nm];
            h = rv1[k];
            f = ((y - z) * (y + z) + (g - h) * (g
```

```
+ h)) / (2.0 * h * y);
            g = PYTHAG(f, 1.0);
            f = ((x - z) * (x + z) + h * ((y / (f
+ SIGN(g, f))) - h)) / x;

            /* next QR transformation */
            c = s = 1.0;
            for (j = l; j <= nm; j++)
            {
                i = j + 1;
                g = rv1[i];
                y = (double)w[i];
                h = s * g;
                g = c * g;
                z = PYTHAG(f, h);
                rv1[j] = z;
                c = f / z;
                s = h / z;
                f = x * c + g * s;
                g = g * c - x * s;
                h = y * s;
                y = y * c;
                for (jj = 0; jj < n; jj++)
                {
                    x = (double)v[jj][j];
                    z = (double)v[jj][i];
                    v[jj][j] = (float)(x * c + z *
s);
                    v[jj][i] = (float)(z * c - x *
s);
                }
                z = PYTHAG(f, h);
                w[j] = (float)z;
                if (z)
                {
                    z = 1.0 / z;
                    c = f * z;
                    s = h * z;
                }
                f = (c * g) + (s * y);
                x = (c * y) - (s * g);
                for (jj = 0; jj < m; jj++)
                {
```

```
                    y = (double)a[jj][j];
                    z = (double)a[jj][i];
                    a[jj][j] = (float)(y * c + z *
s);
                    a[jj][i] = (float)(z * c - y *
s);
                }
            }
            rv1[l] = 0.0;
            rv1[k] =  f;
            w[k] = (float)x;
        }
    }
    free((void*) rv1);
    return(1);
}
```

3. defs_and_types.h

```
#define PRECISION1 32768
#define PRECISION2 16384
/*#define PI 3.141592653589793 2*/
#define MIN(x,y) ( (x) < (y) ? (x) : (y) )
#define MAX(x,y) ((x)>(y)?(x):(y))
#define SIGN(a, b) ((b) >= 0.0 ? fabs(a) : -
fabs(a))
#define MAXINT 2147483647
#define ASCII_TEXT_BORDER_WIDTH 4
#define MAXHIST 100
#define STEP0 0.01
#define FORWARD 1
#define BACKWARD -1
#define PROJ_DIM 5
#define True 1
#define False 0


typedef struct {
    float x, y, z;
} fcoords;
```

```
typedef struct {
    long x, y, z;
} lcoords;

typedef struct {
    int x, y, z;
} icoords;

typedef struct {
    float min, max;
} lims;


/* grand tour history */
typedef struct hist_rec {
  struct hist_rec *prev, *next;
  float *basis[3];
  int pos;
} hist_rec;
```

CONCLUSION

We had successfully implemented the MOORE PENROSE INVERSE finding algorithm to Solve System of Linear Equations, Also, we had learned that the pseudoinverse is a generalized inverse, independent of value of determinant.
It can also compute inverse of any scalar or vector, also Matrix's inverse whose entries are either real or complex can be found.

We had implemented the code with time complexity $\boldsymbol{O\,(m^2n)}$, where the matrix given was of dimension $\boldsymbol{m \times n}$. Where we used a pre-built code and a user defined Library file (dot h) file for decomposing the matrix via SINGULAR VALUE DECOMPOSITION (SVD), which were both Numerical Recipe creation.

We also developed a critical approach of problem solving, and to work in a team, and refer to research papers and the required text in a better manner.