4 Linear Algebra and Regresions

In this chapter we will see how basic linear algebra concepts will serve us to create models to interpret a set of measured points.

With basic linear algebra and matrix notation, we will be ablo to estimate parameters in a model (i.e. by making least square fittig) or explore data by using non parametric descriptions like Principal Component Analysis.

Before diving into direct applications we will start by understading what is the best way to use matrices in C and how to take advantage of the GNU Scientific Library.

4.1 Matrices

We will consider a matrix as a rectangular array of real (or complex) numbers aranged in sets of m rows with n entries. The set of matrices with real coeficients with size $m \times n$ is called $\mathbb{R}^{m \times n}$. In the context of this chapter vectors will be understood as a column vectors, that is as a matrix of one column belonging to the set $\mathbb{R}^m$.

We will refer to the a_{ij} element of matrix $\mathbf{A}$ to indicate the element located in row i and column j. In the case of a vector $\mathbf{x}$ we will refer to its components as x_i.

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & \ddots & & \vdots \\ \vdots & & \dots & a_{mn} \end{bmatrix} \qquad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix}$$

4.2 Matrices as pointers in C

Matrices in C are best stored as one dimensional arrays. This implies that there is convention to translate the position i, j in the matrix into the index p in the one dimensional array.

This convention in C is called **raw major order**, where the second index moves continously in memory. It means that the following 3×3 matrix:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

will be stored in memory as the following one dimensional array

$$\mathbf{a} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{21} & a_{22} & a_{23} & a_{31} & a_{32} & a_{33} \end{bmatrix}$$

.

In FORTRAN the convention is the opposite, the first index moves continously in memory. This convention is known as **column major order**.

In the following example we define a matrix 5×5 and initialize it to be the identity.

```
/*
  Creates an identity matrix following the
  row major order convention in C.
*/
#include <stdio.h>
#include <stdlib.h>
int main(void){
  float *I;
  int n_x=5;
  int n_y=5;
  int pos;
  int i,j;

  if(!(I = malloc(sizeof(float)*n_x *n_y))){
      fprintf(stderr, "Problem with memory allocation");
      exit(1);
  }

  /*Initialization*/
  for(i=0;i<n_x;i++){
    for(j=0;j<n_y;j++){
      pos = j + (n_y * i) ;/*position in the array*/
      if(i==j){
I[pos]=1.0;
      }else{
I[pos]=0.0;
      }
    }
  }

  /*Prints to screen*/
  for(i=0;i<n_x;i++){
    for(j=0;j<n_y;j++){
      pos = i + (n_x * j);/*position in the array*/
fprintf(stdout, " %f ",I[pos]);
    }
    fprintf(stdout, "\n");
  }

  return 0;
}
```

4.3 Functions taking pointers in C

We go into the details of dealing with a array/matrix operation in C. Most of the operations we make onto a matrix will be done using funcitions. Functions

in C work with variables passed as reference, not as values. That means that if you want to have a variable modified, you have to give the address where the variable lives in memory. In practice, that means that you should work only with pointers.

In the next example we define a matrix 3×3 an take its transpose.

```
/*
  Creates an identity matrix following the
  row major order convention in C.
*/
#include <stdio.h>
#include <stdlib.h>
void transpose(float * m, int n_x, int n_y);
void print_matrix(float * m, int n_x, int n_y);

int main(void){
  float *I;
  int n_x=5;
  int n_y=5;
  int pos;
  int i,j;

  if(!(I = malloc(sizeof(float)*n_x *n_y))){
      fprintf(stderr, "Problem with memory allocation");
      exit(1);
  }

  /*Initialization*/
  for(i=0;i<n_x;i++){
    for(j=0;j<n_y;j++){
      pos = j + (n_y * i) ;/*position in the array*/
      I[pos] = pos;
    }
  }

  print_matrix(I, n_x, n_y);

  transpose(I, n_x, n_y);

  print_matrix(I, n_x, n_y);

  return 0;
}
```

```
void print_matrix(float * m, int n_x, int n_y){
  int i,j,pos;

  fprintf(stdout, "\n");
  /*Prints to screen*/
  for(i=0;i<n_x;i++){
    for(j=0;j<n_y;j++){
      pos = i + (n_x * j);/*position in the array*/
      fprintf(stdout, " %f ",m[pos]);
    }
    fprintf(stdout, "\n");
  }
  fprintf(stdout, "\n");
}

void transpose(float * m, int n_x, int n_y){
  int i,j,pos_ij, pos_ji;
  float tmp;
  /*Prints to screen*/
  for(i=0;i<n_x;i++){
    for(j=i;j<n_y;j++){
      pos_ij = i + (n_x * j);
      pos_ji = j + (n_x * i);

      tmp = m[pos_ij];
      m[pos_ij] = m[pos_ji];
      m[pos_ji] = tmp;
    }
  }

}
```

What would happen if you try to modify the variable `n_x` inside a function?

4.4 GNU Scientific Library

Most of the computations that we will perform in C will be done using external libraries. We will tap on the work done by other people. However simple it may seem it is important that you understand how external libraries can be used in C.

Let's start by considering the following example that uses the gsl library to compute the the Bessel function $J_0(x)$ for $x = 5$:

```
#include <stdio.h>
#include <gsl/gsl_sf_bessel.h>
```

```
int main (void){
       double x = 5.0;
       double y = gsl_sf_bessel_J0 (x);
       printf ("J0(%g) = %.18e\n", x, y);
       return 0;
}
```

If the setup for the library are the standard ones, the program can be compiled as

```
cc -lgsl -lm test.c
```

If you are under UBUNTU probably the compilation order matters and you should instead write

```
cc test.c -lgsl -lm
```

Executing the program will produce the following results:

```
J0(5) = -1.775967713143382642e-01
```

GSL also has special libraries to deal with linear algebra operations. A detailed webpage including the manual and useful examples can be found here:

```
http://www.gnu.org/software/gsl/manual/html_node/
```

Now try to follow the structure in the following program that takes a square matix and computes its eigenvalues[1]):

```
#include <stdio.h>
#include <gsl/gsl_math.h>
#include <gsl/gsl_eigen.h>

int main (void){
  double data[] = { 1.0  , 1/2.0, 1/3.0, 1/4.0,
    1/2.0, 1/3.0, 1/4.0, 1/5.0,
    1/3.0, 1/4.0, 1/5.0, 1/6.0,
    1/4.0, 1/5.0, 1/6.0, 1/7.0 };

  gsl_matrix_view m
    = gsl_matrix_view_array (data, 4, 4);

  gsl_vector *eval = gsl_vector_alloc (4);
  gsl_matrix *evec = gsl_matrix_alloc (4, 4);

  gsl_eigen_symmv_workspace * w = gsl_eigen_symmv_alloc (4);
```

[1]Example taken from the example manual of the GSL webpage.

```
  gsl_eigen_symmv (&m.matrix, eval, evec, w);

  gsl_eigen_symmv_free (w);

  gsl_eigen_symmv_sort (eval, evec,
GSL_EIGEN_SORT_ABS_ASC);

  {
    int i;

    for (i = 0; i < 4; i++)
      {
double eval_i
  = gsl_vector_get (eval, i);
gsl_vector_view evec_i
  = gsl_matrix_column (evec, i);

printf ("eigenvalue = %g\n", eval_i);
printf ("eigenvector = \n");
gsl_vector_fprintf (stdout,
    &evec_i.vector, "%g");
      }
  }

  gsl_vector_free (eval);
  gsl_matrix_free (evec);

  return 0;
}
```