INAF - ISTITUTO DI RADIOASTRONOMIA
Via P. Gobetti, 101 40129 Bologna, Italy - Phone: +39-051-6399385 Fax: +39-051-6399431

EMPIRICAL ESTIMATE OF THE COEFFICIENTS COMPUTATION TIME IN THE *MVDR* BEAMFORMING ALGORITHM APPLIED TO *BEST-1* SYSTEM

G. Naldi

Dipartimento di Astronomia, Università di Bologna
INAF-IRA, Istituto Nazionale di Astrofisica, Istituto di Radioastronomia di Bologna

IRA 454/12

TABLE OF CONTENTS

1. Description of the system

In order to fulfil the requirements of the Description of Work (DoW) of SKADS (*Square Kilometer Array Design Study*), different in size pathfinders, based on cylindrical concentrators of the existing Northern Cross radio telescope, have been designed and named BEST (*Basic Element for SKA Training*) 1, 2 and 3lo (low frequency).

BEST-1 is a first level prototype (Figure 1) equipped with 4 front-ends directly installed on the focal line of a single North-South arm cylindrical concentrator (about 24×7.5 m) and with the IF stages in the receiver room. It allowed to test the new low cost and high performance electronics and to start understanding some concepts in order to better define the following BEST-2 test bed architecture.

BEST-1 has a total geometrical area of about 176 square meters.

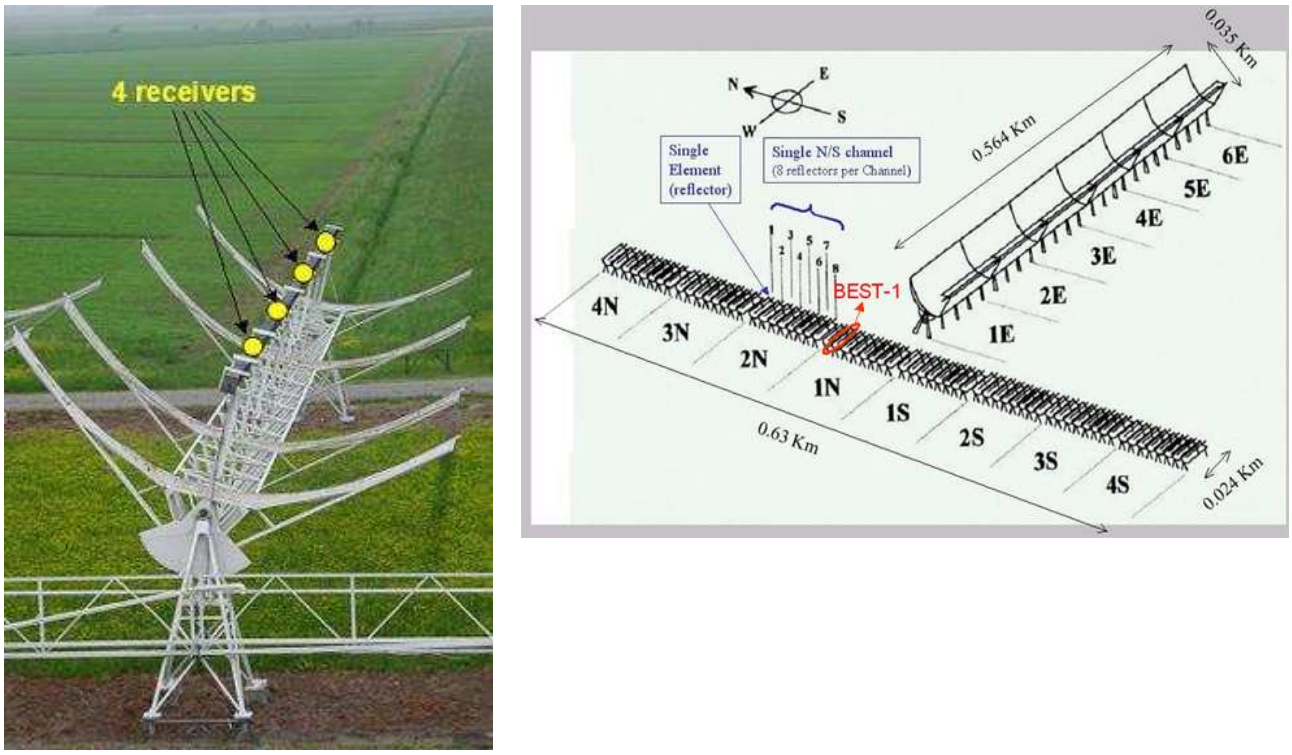

Figure 1: BEST-1 system.

The architecture of the BEST-1 pathfinder is a standard single conversion receiver as sketched in Figure 2. The front ends are installed on the focal line and the RF signals are directly transported to the receiver room via an analogue optical link. After a down conversion at 30 MHz, they are digitised (with the sufficient number of bits for the required dynamic range) and then processed. The IF boards provide an extra output at the RF level (408 MHz) as well. This allows to perform

quick tests on the RF part of the receiver chain (i.e. signal level, RFI monitoring, Front End or optical TX fault). A direct RF output gives also the opportunity to implement some tests on the direct RF sampling.

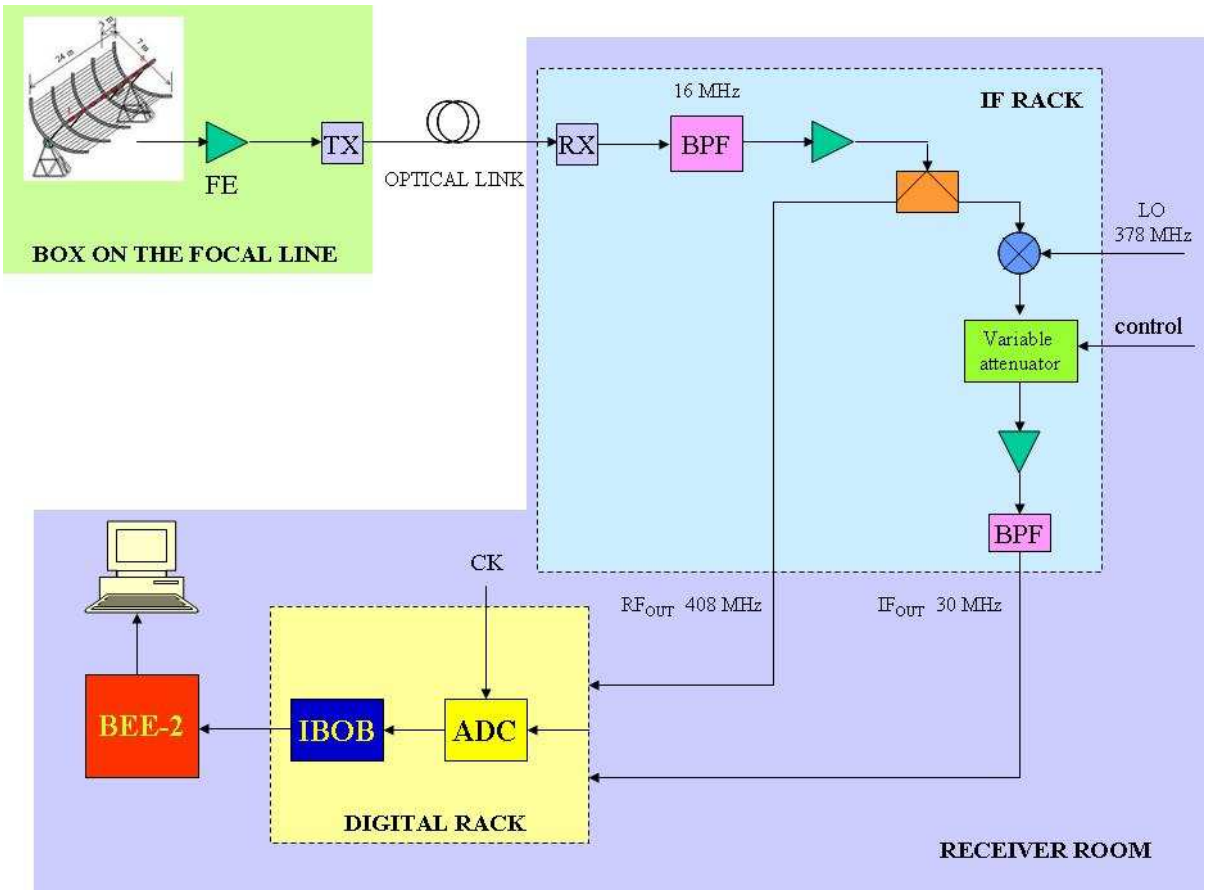

Figure 2: Block diagram of the receiver chain of BEST-1 system.

2. Summary of theory

In the **MVDR (*Minimum Variance Distortionless Response*)** adaptive beamforming algorithm the coefficients vector is calculated to minimize the output power of the beamformer ($P_y = \boldsymbol{w}^H \hat{\boldsymbol{R}}_x \boldsymbol{w}$), satisfying the linear constraint of having the beamformer response equal to *1* in the direction of the desired signal ($\boldsymbol{w}^H \boldsymbol{d}_0 = 1$, $\boldsymbol{d}_0 = \boldsymbol{d}(\theta_0)$).

Supposing the array is calibrated and the single antenna radiation pattern is very similar to the omni-directional one, the ideal steering vector $\boldsymbol{d}_0$ can be considered in place of the Estimated Steering Vector $\hat{\boldsymbol{d}}_0$.

$\boldsymbol{d}_0$ is calculated with the information about the DOA (Direction Of Arrival) of the desired signal (θ_0) and the array geometry. For a linear array the expression is:

$$\boldsymbol{d}_0 = \boldsymbol{d}(\theta_0) = \begin{bmatrix} 1 & e^{j2\pi\frac{d}{\lambda}\sin\theta_0} & \dots & e^{j2\pi(N-1)\frac{d}{\lambda}\sin\theta_0} \end{bmatrix}^H \tag{2-1}$$

The MVDR constrained minimum problem can be expressed as:

$$\boldsymbol{w}_{opt} = arg\left[\min_{\boldsymbol{w}}\left(P_y\right)\right] = arg\left[\min_{\boldsymbol{w}}\left(\boldsymbol{w}^H\,\hat{\boldsymbol{R}}_x\,\boldsymbol{w}\right)\right] \quad with \quad \boldsymbol{w}^H\boldsymbol{d}_0 = 1 \tag{2-2}$$

$\hat{\boldsymbol{R}}_x$ is the Estimated Auto-Covariance Matrix (see next pages for the details about its calculus) and $\boldsymbol{d}_0$ is the array steering vector related to the look direction θ_0.

This constrained minimum problem can be solved through the Lagrange multipliers:

$$\boldsymbol{w}_{opt} = arg\left\{\min_{\boldsymbol{w}}\left[\boldsymbol{w}^H\,\hat{\boldsymbol{R}}_x\,\boldsymbol{w} + \lambda\left(\boldsymbol{w}^H\,\boldsymbol{d}_0 - 1\right)\right]\right\} \tag{2-3}$$

The point of minimum ($\boldsymbol{w}_{opt}$) can be obtained forcing the partial derivatives of the argument, with respect to $\boldsymbol{w}$ and λ, to *0*:

$$\begin{cases} \hat{\boldsymbol{R}}_x\,\boldsymbol{w} + \lambda\,\boldsymbol{d}_0 = 0 \\ \boldsymbol{w}^H\,\boldsymbol{d}_0 = 1 \end{cases} \tag{2-4}$$

that becomes:

$$\begin{cases} \boldsymbol{w} = \lambda\,\hat{\boldsymbol{R}}_x^{-1}\,\boldsymbol{d}_0 \\ \lambda = \dfrac{1}{\boldsymbol{d}_0^H\,\hat{\boldsymbol{R}}_x^{-1}\,\boldsymbol{d}_0} \end{cases} \tag{2-5}$$

and the solution results to be:

$$\boldsymbol{w}_{opt} = \frac{\hat{\boldsymbol{R}}_x^{-1}\boldsymbol{d}_0}{\boldsymbol{d}_0^H\,\hat{\boldsymbol{R}}_x^{-1}\boldsymbol{d}_0} \tag{2-6}$$

The denominator is a scalar value that guarantees the compliance with the linear constraint $\boldsymbol{w}^H\boldsymbol{d}_0 = 1$, which means having a gain equal to *1* in direction θ_0.

Therefore MVDR beamformer implies a response that is *1* in the direction of the wanted signal θ_0 and very attenuated in the directions of the interferers θ_i, $1 \le i \le I$, with *I* = number of interfering signals.

The block diagram of the MVDR beamforming system is represented in the scheme below (Figure 3).

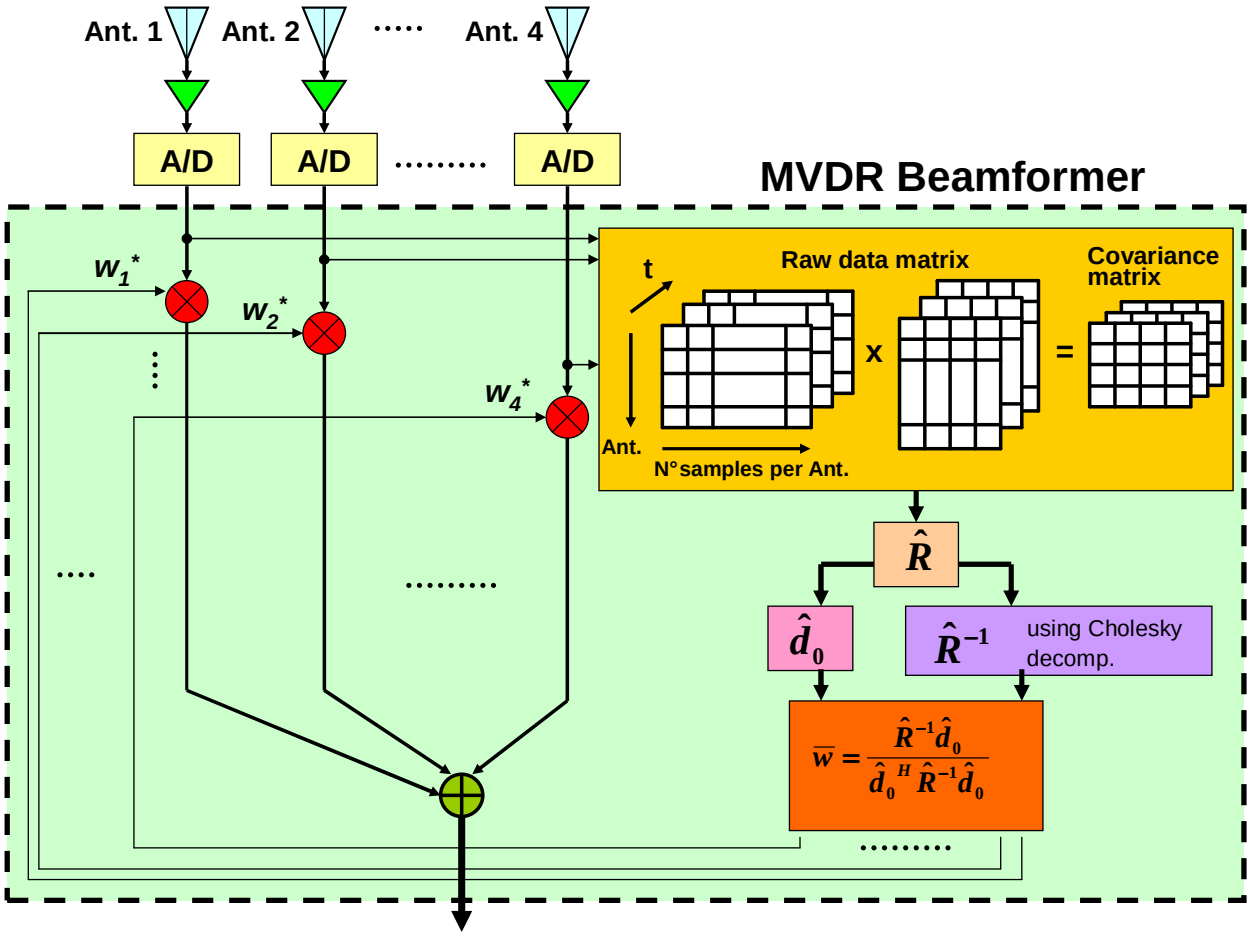

Figure 3: Block diagram of the MVDR beamforming algorithm.

The time samples coming from the A/D converters are collected in a matrix ***X***, called Raw Data Matrix, whose number of rows corresponds to the number of antennas *N* whereas its number of columns corresponds to the number of samples *K* recorded during an appropriate time interval.

This interval (number of samples) has to be opportunely evaluated in order to guarantee a robust data statistics.

The Estimated Auto-Covariance Matrix $\hat{\boldsymbol{R}}_x$ is calculated thanks to a time-domain average of $\boldsymbol{X}$:

$$\hat{\boldsymbol{R}}_x = \sum_{k=1}^{K} \boldsymbol{x}\left(t_k\right) \boldsymbol{x}^H\left(t_k\right) = \boldsymbol{X}\ \boldsymbol{X}^H \tag{2-7}$$

In practice the signals are supposed to be ergodic and the calibration parameters are supposed to be constant in a time interval of *K* samples. With these hypothesis $\hat{\boldsymbol{R}}_x\left(t\right)$ is therefore supposed to be constant in time and consequently can be estimated with a time-domain average. This assumption is reasonable since only relatively short time intervals are considered.

If the array is composed of *N* antennas $\hat{\boldsymbol{R}}_x$ is a square Hermitian matrix of size $N \times N$.

The data storage in matrix $\boldsymbol{X}$ and the calculus of matrix $\hat{\boldsymbol{R}}_x$ have to be periodically repeated to make the MVDR algorithm adaptive. This periodicity has to be accurately determined taking into account the scenario (e.g. movement speed of interferers and radio sources…) and the characteristics of the antenna radiation beampattern. This is very important in order to maintain a high level of gain in direction of the radio source to be received and, simultaneously, a good suppression of the interfering signals as long as the algorithm works.

The calculus of $\hat{\boldsymbol{R}}_x^{-1}$ can present some difficulties; anyway it is recommended not to invert the matrix $\hat{\boldsymbol{R}}_x$ directly but, for a computationally efficient implementation, it is advisable to use the *Cholesky Decomposition* (see next section).

3. Implementation in C language

Preliminary note: MVDR adaptive beamforming algorithm is based on the calculus of the amplitudes and phases of specific coefficients (weights) that have to be multiplied by the input signals (as described in the previous section). These coefficients are complex numbers.
Therefore it is necessary to define the "complex number" type in the program, for example with the use of a structure. Consequently all the arithmetic operations have to be redefined or, more simply, extended to the case of complex numbers.

After having calculated the matrix $\hat{\boldsymbol{R}}_x$ (the Estimated Auto-Covariance Matrix, as stated before) from the data acquired by the receiver system, it's necessary to calculate the inverse of the matrix $\hat{\boldsymbol{R}}_x$ itself.

This matrix is, by definition, a Hermitian definite positive matrix, so an efficient implementation of $\hat{\boldsymbol{R}}_x^{-1}$, in terms of calculus time, can be obtained using the algorithm of the *Cholesky* Decomposition. With this kind of decomposition the matrix $\hat{\boldsymbol{R}}_x$ can be expressed in the following way:

$$\hat{\boldsymbol{R}}_x = \boldsymbol{L} \cdot \boldsymbol{L}^{\boldsymbol{H}} \tag{3-1}$$

where $\boldsymbol{L}$ is a lower triangular matrix.

Once $\boldsymbol{L}$ matrix is calculated, $\hat{\boldsymbol{R}}_x^{-1}$ can be obtained with the following mathematical operations:

$$\hat{\boldsymbol{R}}_x^{-1} = \left(\boldsymbol{L} \cdot \boldsymbol{L}^{\boldsymbol{H}}\right)^{-1} = \left(\boldsymbol{L}^{\boldsymbol{H}}\right)^{-1} \cdot \boldsymbol{L}^{-1} \tag{3-2}$$

This expression highlights the great advantage offered by the *Cholesky* algorithm for the inversion of $\hat{\boldsymbol{R}}_x$ matrix as regards both the memory required to store the data and the number of operations and, as a consequence, even in terms of computation time. In fact it should be noted that, starting from the knowledge of only $\boldsymbol{L}$ matrix, after some calculations, it is possible to derive $\hat{\boldsymbol{R}}_x^{-1}$.

It is important to remark that the computational cost related to the *Cholesky* Decomposition is proportional to $\frac{N^3}{6}$, where N is the matrix $\hat{\boldsymbol{R}}_x$ dimension, whereas the proportionality factor associated to the *Gauss* factorization with partial pivoting is $\frac{N^3}{3}$.

This means that the *Cholesky* Decomposition causes a considerable saving in terms of operations carried out by the CPU, in particular considering that the application context of this adaptive beamforming algorithm foresees a periodic very frequent update of the matrix $\hat{\boldsymbol{R}}_x$ and of its inverse of course. Moreover it is worth mentioning that, in the specific case of this program, it has been taken into consideration an optimized version of the *Cholesky* Decomposition, which requires the smallest possible quantity of memory.

4. Results

The software program (C language) that implements the *MVDR* algorithm in the case of BEST-1 system has been tested on a Xeon computer (dual CPU @ 2.8 GHz, 1,00 GByte RAM) in order to have some realistic information about the achievable performances in terms of computation time.

Some simulations have been carried out: in particular the input data (stored in the Raw Data Matrix ***X***, mentioned before) has been generated using MATLAB, supposing to point the antenna beampattern in the direction of a radio source with $\theta_0 = 2°$.

To demonstrate that the results obtained with the C code were correct, the beamformer coefficients $\boldsymbol{w}_{MVDR}$ calculated by the C program have been compared with the ones calculated by a MATLAB program, already used for previous simulations of the same algorithm.

The main goal of the simulations is to have a measure, as accurate as possible, of the time required by the program to calculate the coefficients. For this purpose it has been set up a timing test, which takes into account the time interval that elapses from the reading of the input data up to the calculations of the final coefficients. This test has been run many times: the time values obtained as output data of the test have been increasingly added up and accumulated, and then averaged on the total number of times the test has been executed; in other words a statistical measurement of computation time has been accomplished.

Table 1 shows synthetically the results:

Time window [samples]	**Number of test repetitions**	**Average user time [sec]**	**Average machine time [sec]**
200	3750000 (~ 10 min. of sim.)	0.000149	0.000148
200	**11250000 (~ 30 min. of sim.)**	**0.000149**	**0.000149**
1000	1000000 (~ 12 min. of sim.)	0.000699	0.000699
1000	**3000000 (~ 34 min. of sim.)**	**0.000699**	**0.000699**

Table 1: Timing test results.

Two different kinds of time measurement are present in this table: average user time and average machine time.

The average user time is calculated with the use of the standard C function *clock* (Time.h): it provides the number of CPU "clock ticks" elapsed from the moment the program starts running.

The test procedure was as follow: first of all the number of CPU clock ticks measured at the instant that the main program reads the input data (beginning of the time test) are subtracted to the ones measured at the instant the main program calculates the beamformer coefficients; then the result of this subtraction has to be divided by the number of CPU clock ticks elapsed in one second, so, in this way, the average user time expressed in [s] is finally derived.

On the other hand, the average machine time is obtained making use of the standard C function *time* (Time.h), which returns the current calendar time. In this case, in order to have the machine time already expressed in [s], it's sufficient to subtract the calendar time at the instant the main program begins to read the input data to the one the coefficients are calculated.

As shown on the table above, the two measurements of time present more or less the same values: this fact represents a sort of correctness confirmation of the obtained results.

Moreover, two cases of measurement have been reported in bold to highlight that we are dealing with the values achieved setting a sufficient number of test repetitions (simulation time) as to have statistically valid results.

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <unistd.h>
#include <time.h>

#define PI 3.14159265358979323846264338327
#define tipo double

/* Definizione di una struttura per la rappresentazione dei numeri complessi */
typedef struct nComplesso {
        tipo re,im;
        } nComplex;

/* Definizione di una struttura necessaria per memorizzare i dati di tempo */
/*struct tms {
    clock_t tms_utime;    // user   CPU time
    clock_t tms_stime;    // system CPU time
    clock_t tms_tcutime;  // user   CPU time (terminated children)
    clock_t tms_tcstime;  // system CPU time (terminared children)
    }; */


/* Variabili GLOBALI: variabili secondarie necessarie alla misura del tempo di simul
azione */
/* clock_t real_total_time = 0;  //Tempo REALE   totale
clock_t user_total_time = 0;  //Tempo UTENTE  totale
clock_t syst_total_time = 0;  //Tempo SISTEMA totale
tipo clktck = 0.0; */
double tempo_utente = 0.0;
double tempo_macchina = 0.0;


/* Funzioni che gestiscono le operazioni tra numeri complessi */

/* Addizione tra due numeri complessi */
nComplex Cadd (nComplex a, nComplex b){
         nComplex c;
         c.re = a.re + b.re;
         c.im = a.im + b.im;
         return c;
         }

/* Sottrazione tra due numeri complessi */
nComplex Csub (nComplex a, nComplex b){
         nComplex c;
         c.re = a.re - b.re;
         c.im = a.im - b.im;
         return c;
         }

/* Moltiplicazione tra due numeri complessi */
nComplex Cmul (nComplex a, nComplex b){
         nComplex c;
         c.re = a.re*b.re - a.im*b.im;
         c.im = a.im*b.re + a.re*b.im;
         return c;
         }

/* Costruzione di un numero complesso a partire da due numeri reali */
nComplex Complex (tipo reale, tipo immaginario){
         nComplex c;
         c.re = reale;
         c.im = immaginario;
         return c;
         }

/* Operazione di coniugio di un numero complesso */
nComplex Conjg (nComplex z){
```

```
          nComplex c;
          c.re = z.re;
          c.im = -z.im;
          return c;
          }

/* Divisione tra due numeri complessi */
nComplex Cdiv (nComplex a, nComplex b){
          nComplex c;
          tipo r,den;
          if (fabs(b.re) >= fabs(b.im)) {
              r = b.im/b.re;
              den = b.re + r*b.im;
              c.re = (a.re + r*a.im)/den;
              c.im = (a.im - r*a.re)/den;
              }
          else {
               r = b.re/b.im;
               den = b.im + r*b.re;
               c.re = (a.re*r + a.im)/den;
               c.im = (a.im*r - a.re)/den;
               }
          return c;
          }

/* Valore assoluto di un numero complesso */
tipo Cabs (nComplex z){
       tipo x,y,ans,temp;
       x = fabs(z.re);
       y = fabs(z.im);
       if (x == 0.0) ans=y;
       else if (y == 0.0) ans=x;
       else if (x > y) {
                temp = y/x;
                ans = x*sqrt(1.0+temp*temp);
                }
       else {
            temp = x/y;
            ans = y*sqrt(1.0+temp*temp);
            }
       return ans;
       }

/* Radice quadrata di un numero complesso */
nComplex Csqrt (nComplex z){
          nComplex c;
          tipo x,y,w,r;
          if ((z.re == 0.0) && (z.im == 0.0)) {
              c.re = 0.0;
              c.im = 0.0;
              return c;
              }
          else {
               x = fabs(z.re);
               y = fabs(z.im);
               if (x >= y) {
                   r = y/x;
                   w = sqrt(x)*sqrt(0.5*(1.0 + sqrt(1.0 + r*r)));
                   }
               else {
                    r = x/y;
                    w = sqrt(y)*sqrt(0.5*(r + sqrt(1.0 + r*r)));
                    }
               if (z.re >= 0.0) {
                   c.re = w;
                   c.im = z.im/(2.0*w);
                   }
               else {
                    c.im = (z.im >= 0.0) ? w : -w;
```

```
                    c.re = z.im/(2.0*c.im);
                    }
                return c;
                }
        }

/* Moltiplicazione tra un numero reale ed un numero complesso */
nComplex RCmul (tipo x, nComplex a){
        nComplex c;
        c.re = x*a.re;
        c.im = x*a.im;
        return c;
        }

/*****************************************************************************
Decomposizione di Cholesky

Questa funzione calcola la decomposizione di Cholesky di una matrice quadrata
simmetrica definita positiva.

Si ricordi che la decomposizione di Cholesky di una matrice quadrata simmetrica
definita positiva A produce come risultato A = L*L'.

Parametri di ingresso:
    A       -   matrice quadrata, simmetrica e definita positiva.
    n       -   dimensione della matrice A.

Parametri di uscita:
    A       -   la matrice triangolare inferiore risultante dalla decomposizione
                di Cholesky L viene inserita nel triangolo inferiore della
                matrice A; gli altri elementi della matrice A invece non vengono
                modificati.
*****************************************************************************/
//void CholeskyDecomposition (nComplex a[2][2], int n){
void CholeskyDecomposition (nComplex **a, int n){
     // void nrerror (char error_text []);
     int i,j,k;
     nComplex sum;
     for (j=0;j<n;j++){
         sum.re = 0.0;
         sum.im = 0.0;
         for (k=0;k<=j-1;k++){
             sum = Cadd(sum,Cmul(a[j][k],a[j][k]));
             }
         a[j][j] = Csub(a[j][j],sum);
         a[j][j] = Csqrt(a[j][j]);
         //a[j][j].im = 0.0;
         if (a[j][j].re==0.0 && a[j][j].im==0.0){
             // nrerror ("Cholesky Decomposition failed");
             printf ("Cholesky Decomposition failed/n");
             }
         if (j<n-1){
             for (i=j+1;i<n;i++){
                 sum.re = 0.0;
                 sum.im = 0.0;
                 for (k=0;k<=j-1;k++){
                     sum = Cadd(sum,Cmul(a[i][k],a[j][k]));
                     }
                 a[i][j] = Csub(a[i][j],sum);
                 a[i][j] = Cdiv(a[i][j],a[j][j]);
                 }
             }
         }
     }



/*****************************************************************************
Metodo alternativo della decomposizione di Cholesky secondo il libro "Numerical
```

```
Recipes in C"
*****************************************************************************/
void choldc (nComplex **a, int n){
     int i,j,k;
     nComplex sum;
     for (i=0;i<n;i++){
         for (j=i;j<n;j++){
             for (sum.re=a[i][j].re,sum.im=a[i][j].im,k=i-1;k>=0;k--)  sum = Csub(su
m,Cmul(Conjg(a[i][k]),a[j][k]));
             //for (sum.re=a[i][j].re,sum.im=a[i][j].im,k=0;k<i;k++) sum = Csub(sum,
Cmul(a[i][k],a[j][k]));
             if (i==j){
                      if (a[j][j].re==0.0 && a[j][j].im==0.0){
                          printf ("Cholesky Decomposition failed/n");
                          }
                      /*a[i][i].re = sqrt(sum.re);
                      a[i][i].im = 0.0;*/
                      a[i][i] = Csqrt(sum);
                      }
             else a[j][i] = Cdiv(sum,a[i][i]);
             }
         }
     for (i=1;i<n;i++){
         for (j=0;j<i;j++){
             a[i][j] = Conjg(a[i][j]);
             }
         }
     }




/*****************************************************************************
Matrice inversa della matrice decomposta secondo il metodo di Cholesky

Questa funzione calcola la matrice inversa di una matrice quadrata simmetrica
definita positiva a partire dalla matrice L fornita dalla decomposizione di
Cholesky.

Parametri di ingresso:
    A       -   matrice ottenuta con la decomposizione di Cholesky della
                matrice che deve essere invertita (A = L*L').
                Uscita della subroutine CholeskyDecomposition.
    n       -   dimensione della matrice A.

Parametri di uscita:
    A       -   matrice inversa di una matrice quadrata simmetrica definita
                positiva decomposta secondo il metodo di Cholesky.
*****************************************************************************/
void InverseCholesky (nComplex **a, int n){
     int i,j,k;
     nComplex sum,aii,unita;
     unita.re = 1.0;
     unita.im = 0.0;

     /* Calcolo di inv(L) metodo 1*/
     for (i=0;i<n;i++){
         a[i][i] = Cdiv(unita,a[i][i]);
         for (j=i+1;j<n;j++){
             sum.re = 0.0;
             sum.im = 0.0;
             for (k=i;k<j;k++){
                 sum = Csub(sum,Cmul(a[j][k],a[k][i]));
                 }
             a[j][i] = Cdiv(sum,a[j][j]);
             }
         }
```

```
     /* Calcolo di inv(L) metodo 2 */
     /* nComplex ajj;
     nComplex v[n+1];
     for (j=n-1;j>=0;j--){
          if ((a[j][j].re==0.0) && (a[j][j].im==0.0)){
              printf ("Matrix inversion failed/n");
              }
          a[j][j] = Cdiv(unita,a[j][j]);
          ajj.re = -a[j][j].re;
          ajj.im = -a[j][j].im;
          if (j<n-1){
              for (i=j+1;i<n;i++){
                  v[i] = a[i][j];
                  }
              for (i=j+1;i<n;i++){
                  sum.re = 0.0;
                  sum.im = 0.0;
                  for (k=j+1;k<=i;k++){
                      sum = Cadd(sum,Cmul(a[i][k],v[k]));
                      }
                  a[i][j] = sum;
                  }
              for (i=j+1;i<n;i++){
                  a[i][j] = Cmul(ajj,a[i][j]);
                  }
              }
          }*/




     /* inv(A) = inv(L') * inv(L) */
     for (i=0;i<n;i++){
         aii = a[i][i];
         if (i<n-1){
             sum.re = 0.0;
             sum.im = 0.0;
             for (j=i;j<n;j++){
                 sum = Cadd(sum,Cmul(Conjg(a[j][i]),a[j][i]));
                 }
             a[i][i] = sum;
             for (j=0;j<=i-1;j++){
                 sum.re = 0.0;
                 sum.im = 0.0;
                 for (k=i+1;k<n;k++){
                     sum = Cadd(sum,Cmul(a[k][j],Conjg(a[k][i])));
                     }
                 a[i][j] = Cadd(sum,Cmul(aii,a[i][j]));
                 a[j][i] = Conjg(a[i][j]);
                 }
             }
        else {
             for (j=0;j<=i-1;j++){
                 a[i][j] = Cmul(aii,a[i][j]);
                 a[j][i] = Conjg(a[i][j]);
                 }
             a[n-1][n-1] = Cmul(aii,a[n-1][n-1]);
             }
        }
    }

/* Funzione che calcola il prodotto tra una matrice ed un vettore */
void ProdottoMatriceVettore (nComplex **a, nComplex *b, nComplex *c, int n, int m){
     int i,j;
     for (i=0;i<n;i++){
         c[i].re = 0.0;
         c[i].im = 0.0;
```

```
          for (j=0;j<m;j++){
              c[i] = Cadd(c[i],Cmul(a[i][j],b[j]));
              }
          }
     }

/* Funzione che trasforma un vettore con il suo Hermitiano */
void VettoreHermitiano (nComplex *a, nComplex *b, int n){
     int i;
     for (i=0;i<n;i++){
         b[i] = Conjg(a[i]);
         }
     }

/* Funzione che calcola il prodotto tra un vettore ed una matrice */
void ProdottoVettoreMatrice (nComplex *a, nComplex **b, nComplex *c, int n, int m){
     int i,j;
     for (i=0;i<m;i++){
         c[i].re = 0.0;
         c[i].im = 0.0;
         for (j=0;j<n;j++){
             c[i] = Cadd(c[i],Cmul(a[j],b[j][i]));
             }
         }
     }

/* Funzione che calcola il prodotto tra due vettori */
void ProdottoVettoreVettore (nComplex *a, nComplex *b, nComplex *c, int n){
     int i;
     c->re = 0.0;
     c->im = 0.0;
     for (i=0;i<n;i++){
         *c = Cadd(*c,Cmul(a[i],b[i]));
         }
     }

void mvdr_1D (nComplex **dati,nComplex **Cov,nComplex *SteeringVec,int n,int k,int d
ist,tipo dir){
     int i,j,t;
     nComplex mediaTempDati;

     /* Costruzione della matrice di covarianza a partire dalla matrice dei dati
     grezzi */
     for (i=0;i<n;i++){
         for (j=0;j<n;j++){
             if (i<=j){
                 mediaTempDati.re = 0.0;
                 mediaTempDati.im = 0.0;
                 for (t=0;t<k;t++){
                     mediaTempDati = Cadd(mediaTempDati,Cmul(dati[i][t],Conjg(dati[j
][t])));
                     }
                 Cov[i][j] = mediaTempDati;
                 }
             else {
                  Cov[i][j] = Conjg(Cov[j][i]);
                  }
             }
         }

     /* Costruzione dello steering vector della schiera BEST-1 a partire dalla
        conoscenza della direzione di arrivo (DOA) del segnale utile */
     for (i=0;i<n;i++){
         SteeringVec[i] = Complex(cos(2.0*PI*i*dist*sin(dir)),sin(2.0*PI*i*dist*sin(
dir)));
         }
     }
```

```
void test (nComplex **dati,nComplex **Cov,nComplex *SteeringVec,nComplex *coeff,nCom
plex *den,int n,int m,int k,int dist,tipo dir){
          int i;
          //struct tms tmsstart,tmsend;
          //time_t tmsstart,tmsend;
          clock_t Start,End;
          time_t StartTime, EndTime;
          //double time_elapsed;
          double time_elapsed;
          double machine_time_elapsed;

          /*  *** START CRONOMETRO *** */
          //Start = time(&tmsstart);
          Start = clock();
          StartTime = time(NULL);

          //  *** Sequenza di istruzioni per il calcolo dei coefficienti del Beamfor
mer ***
          mvdr_1D(dati,Cov,SteeringVec,n,k,dist,dir);

          choldc(Cov,n);
          InverseCholesky(Cov,n);

          ProdottoMatriceVettore(Cov,SteeringVec,coeff,n,m);

          VettoreHermitiano(SteeringVec,SteeringVec,n);

          ProdottoVettoreVettore(SteeringVec,coeff,den,n);

          for (i=0;i<n;i++) coeff[i] = Cdiv(coeff[i],*den);

          /*  *** STOP CRONOMETRO *** */
          //End = time(&tmsend);
          End = clock();
          EndTime = time(NULL);

          /* Annotazione dei tempi di calcolo */
          //real_total_time += (End-Start);
          //user_total_time += (tmsend.tms_utime - tmsstart.tms_utime);
          //syst_total_time += (tmsend.tms_stime - tmsstart.tms_stime);
          time_elapsed = ((double)End - (double)Start) / (double)CLOCKS_PER_SEC;
          machine_time_elapsed = difftime(EndTime,StartTime);
          tempo_utente += (double)time_elapsed;
          tempo_macchina += machine_time_elapsed;
          }



main()
{
    /* Parametri di simulazione immessi dall'utente da console */
    int N;
    int d_sensori;
    int K;
    tipo DOA;
    tipo T;

    printf("Inserire il numero dei ricevitori della schiera (N):\n");
    scanf("%d", &N);
    printf("Inserire la distanza tra i ricevitori della schiera (d_sensori):\n");
    scanf("%d", &d_sensori);
    printf("Inserire il numero di campioni utilizzati per il calcolo della matrice d
i covarianza (K):\n");
    scanf("%d", &K);
    printf("Inserire la direzione di puntamento (DOA) espressa in gradi:\n");
    scanf("%lf", &DOA);
    DOA = (DOA*PI)/180.0;
    printf("Inserire il numero di ripetizioni del test per la media statistica:\n");
```

```
    scanf("%lf", &T);

    int i,j;
    tipo real,imaginary;
    nComplex **X;
    nComplex **MatriceCovarianza;
    nComplex *SteeringVector;
    nComplex denominatore;
    nComplex *w;

    X = malloc(N*sizeof(nComplex*));
    for (i=0;i<N;i++){
        X[i] = malloc(K*sizeof(nComplex));
        }

    MatriceCovarianza = malloc(N*sizeof(nComplex*));
    for (i=0;i<N;i++){
        MatriceCovarianza[i] = malloc(N*sizeof(nComplex));
        }

    SteeringVector = malloc(N*sizeof(nComplex));

    w = malloc(N*sizeof(nComplex));

    FILE *f_dati;
    FILE *f_coefficienti;
    FILE *f_tempi;

    f_dati = fopen("Dati.txt","rt");
    if(!f_dati)
    {   printf("\n\nImpossibile aprire il file...");
        exit(1);
    }

    for (i=0;i<N;i++){
        for (j=0;(j<K) && (!feof(f_dati));j++){
            fscanf(f_dati,"%lf %lfi",&real,&imaginary);
            X[i][j].re = real;
            X[i][j].im = imaginary;
            //printf("RE=%lf\tIM=%lf\t",X[i][j].re,X[i][j].im);
            }
        //printf("\n\n");
        }

    fclose(f_dati);


    /*****************************************************************/
    /***************** Corpo centrale del programma ******************/

    printf("Test in corso. Attendere, prego...\n");
    for (i=0;i<T;i++) test(X,MatriceCovarianza,SteeringVector,w,&denominatore,N,N,K,
d_sensori,DOA);

    /*****************************************************************/

    /* Visualizzazione dei TEMPI medi */
    //clktck = sysconf(_SC_CLK_TCK);
    //clktck = clock();

    f_tempi = fopen("Tempi.txt","wt");
    if(!f_tempi) {
        printf("\n\nImpossibile aprire il file...");
        exit(1);
        }

    fprintf(f_tempi,"Test effettuato %lf volte su %d campioni di segnale ricevuti da
 ognuno dei %d ricevitori della schiera BEST-1.\n\n\n",T,K,N);
    //fprintf(f_tempi,"Tempo REALE    totale : %.6f s.\n",real_total_time/(tipo)clkt
```

```
ck);
    fprintf(f_tempi,"Tempo UTENTE totale : %5.8f s.\n",tempo_utente);
    fprintf(f_tempi,"Tempo MACCHINA totale : %5.8f s.\n",tempo_macchina);
    //fprintf(f_tempi,"Tempo SISTEMA  totale : %.6f s.\n",syst_total_time/(tipo)clkt
ck);
    //fprintf(f_tempi,"Tempo REALE    medio  : %.6f s.\n",(real_total_time/(tipo)T)/
(tipo)clktck);
    fprintf(f_tempi,"Tempo UTENTE medio  : %5.8f s.\n",((double)tempo_utente/(double
)T));
    fprintf(f_tempi,"Tempo MACCHINA medio  : %5.8f s.\n",((double)tempo_macchina/(do
uble)T));
    //fprintf(f_tempi,"Tempo SISTEMA  medio  : %.6f s.\n",(syst_total_time/(tipo)T)/
(tipo)clktck);

    fclose (f_tempi);


    f_coefficienti = fopen("Coefficienti.txt","wt");
    if(!f_coefficienti) {
        printf("\n\nImpossibile aprire il file...");
        exit(1);
        }

    fprintf(f_coefficienti,"Coefficienti del beamformer MVDR:\n\n\n");
    for (i=0;i<N;i++) fprintf(f_coefficienti,"RE=%lf\tIM=%lf\t",w[i].re,w[i].im);


    fclose(f_coefficienti);

    for (i=0;i<N;i++) free(X[i]);
    free(X);

    for (i=0;i<N;i++) free(MatriceCovarianza[i]);
    free(MatriceCovarianza);

    free(SteeringVector);

    free(w);

    return(0);
}
```