Optimised MVDR Coherence*

Neil Carter
Swansea University, UK

4th December 2008

Abstract

The script `coherence_MVDR.M` contains a number of redundancies that reduce its performance. This article explains the redundancies and how they can be optimised. An optimised script is given at the end. In tests, the original script had $n^{2.15}$ performance, but the optimised script achieved $n^{1.94}$, which is effectively twice as fast. Certain optimisations produced little or no benefit, indicating that MatLab's built-in optimiser is very effective. Interestingly, the inv() function was actually *slower* than the naive calculation.

Green text explains things that *can* be optimised.

Red text explains things that can *not* be optimised.

Black text is just ordinary explanation.

1 Calling the Function

The **`loadcnt()`** function loads a continuous EEG file, and returns three values. Variable name, rows x columns, value type:

- **`cnt`**: 32 x 10 char

*Based on the MatLab script published on the MathWorks File Exchange by Jacob Benesty

- **signal1**: struct
  - **data**: 32 x 64128 double
  - **header**: 1 x 1 struct
  - **electloc**: 1 x 32 struct
  - **Teeg**: 1 x 1 struct
  - **event**: 1 x 1 struct
  - **tag**: 0
- **H**: struct
  - **stimtype**
  - **keyboard**
  - **keypad_accept**
  - **offset**
  - **type**
  - **code**
  - **latency**
  - **epochevent**
  - **accept**
  - **accuracy**

signal1.data contains the electrode data. Samples for each electrode are stored in a single row. Samples over time are stored in columns 1 to 64128.

```
[signal1, cnt, H] = loadcnt('013EC06.cnt');

```

The following is the calling code. It consists of two nested loops. The outer loop refers to the *source* electrode of the pair, and steps through all the electrodes except for the last one, since by that time, all the pair combinations will have been analysed. That is, $1 \rightarrow 2$ up to $31 \rightarrow 32$ for 32 electrodes.

The inner loop refers to the *destination* electrode of the pair. It steps through all electrodes, starting from the one after the source electrode because it will have

already done all the pairs for all electrodes up to that one, and it doesn't analyse pairs where the source and destination electrode are the same. For instance, after doing electrode pairs $1 \rightarrow 2$ up to $1 \rightarrow 32$, the next pair should be $2 \rightarrow 3$, since $2 \leftarrow 1$ will give the same result as $1 \rightarrow 2$. The final pair will be $31 \rightarrow 32$ since the 32^{nd} electrode will have been cohered (as destination electrode) with all the preceding electrodes by the time its turn to be source electrode comes around.

```
for src_electrode = 1 : num_electrodes - 1
        for dst_electrode = src_electrode + 1 : num_electrodes
                sprintf( 'Analysing %d, %d ', src_electrode, dst_electrode )
                x1 = signal1.data(src_electrode, 1:window_length);
                x2 = signal1.data(src_electrode, 1:window_length);
                [MSC] = coherence_MVDR(x1, x2, window_length, K );
        end % for destination electrode
end % for source electrode
```

Now follows the coherence function header:

```
function [MSC]=coherence_MVDR(x1,x2,L,K);

%% This program computes the coherence function between 2 signals
%% x1 and x2 with the MVDR method.
%% This algorithm is based on the paper by the same authors:
%% J. Benesty, J. Chen, and Y. Huang, "A generalized MVDR spectrum,"
%% IEEE Signal Processing letters, vol. 12, pp. 827-830, Dec. 2005.

%% x1, first signal vector of length n
%% x2, second signal vector of length n
%% L is the length of MVDR filter or window length
%% K is the resolution (the higher K, the better the resolution)

```

2 Optimising the Function

The following code constructs column-vectors **xx1**, etc., with **L** rows, all values being set to zero. (**L** is the length of the window, typically 1024, always powers of two.)

$$\begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ - \\ L \end{matrix} \begin{bmatrix} 0_1 \\ 0_2 \\ 0_3 \\ 0_4 \\ \vdots \\ 0_L \end{bmatrix}$$

The effect of this code does not change between calls of this function. It generates identical matrices with identical values (zero). Also, any existing values are overwritten by each call of the function. Therefore, this code can be optimised by moving it outside both the electrode loops and passing the variables into the function, or by making them persistent variables local to the function. Where variables are created or calculated once only upon the first iteration, the code can be moved inside an **`if isempty()`** condition, so that it is executed only upon the first iteration.

```
14 %initialization
15 xx1     = zeros(L,1);
16 xx2     = zeros(L,1);
17 r11     = zeros(L,1);
18 r22     = zeros(L,1);
19 r12     = zeros(L,1);
20 r21     = zeros(L,1);
21
```

F is a matrix of **L** rows and **K** columns.

$$F = \begin{bmatrix} 0_{1,1} & 0 & 0 & \dots & 0_{1,K} \\ \vdots & 0 & 0 & \ddots & 0 \\ 0_{L,1} & 0 & 0 & \dots & 0_{L,K} \end{bmatrix}$$

K is the desired frequency resolution, which should be at most half of L, because of the Nyquist problem. Typically 128.

l is a column vector containing values of 0 to **L**-1.

$$l = \begin{bmatrix} 0 \\ 1 \\ 2 \\ 3 \\ \vdots \\ 1023 \end{bmatrix}$$

f is a scalar, referring to the Euler equation $e^{2\pi lj/K}$.

These three lines produce results that do not change during a run of this program. Therefore, this code can be optimised by moving it outside both the electrode loops and passing the variables into the function, or by using persistent variables local to the function.

```
22 %construction of the Fourier Matrix
23 F       = zeros(L,K);
24 l       = [0:L-1]';
25 f       = exp(2*pi*l*j/K);
```

This code generates a matrix where the **k**+1$^{\text{th}}$ column of each row of **F** has the value $\frac{f^k}{\sqrt{L}}$, where $k = [0, 1, 2, ..., K-1]$.

This code produces results that do not change during the run of this program. Therefore, this code can be optimised by moving it outside both the electrode loops and passing the variables into the function, or by using persistent variables local to the function.

```
26 for k = 0:K-1
27     F(:,k+1) = f.^k;
28 end
29 F       = F/sqrt(L);
30
```

The following code steps through each sample from both electrodes, building column vectors (**xx1** and **xx2**) where the last (bottom) element is the earliest (leftmost) sample. The values are added into the vector in the manner of a stack; the first value is pushed to the bottom.

n is constant during the run, so this can be optimised by moving the code outside the loops, or by using a persistent variable local to the function. The **r11** and **r22** matrices can be optimised since they have the same values for the same electrodes. So, when the *source* electrode is, say, 5, then **r11** will have the same value as **r22** had when the *destination* electrode was 5.

The loop that builds the **xx** matrices must be computed at each call of the function. The reason for this is that each call of the function is made with a unique combination of electrodes. Therefore, the **r12** and **r21** matrices are unique for each call. Moreover, their values depend on the **xx** matrices containing zero in all elements at the start of the loop, since they are both cumulative sums and are generated concurrently.

```
31 %number of samples, equal to the lenght of x1 and x2
32 n       = length(x1);
33
34 for i = 1:n
35     xx1 = [x1(i);xx1(1:L-1)];
36     xx2 = [x2(i);xx2(1:L-1)];
37     r11 = r11 + xx1*conj(xx1(1));
38     r22 = r22 + xx2*conj(xx2(1));
39     r12 = r12 + xx1*conj(xx2(1));
40     r21 = r21 + xx2*conj(xx1(1));
41 end
```

The following lines average the values in the elements of the **r** matrices.

The **r12** and **r21** matrices are unique for each call of the function so these cannot be optimised.

The **r11** and **r22** matrices will have the same value when they correspond to a given electrode. That is, **r22** will have the same value when the *destination* electrode was (say) 5 as **r11** had when the *source* electrode was 5, and vice versa. After the first iteration of the outer (source) loop, the destination electrode will have taken on all values except 1. Therefore, this code can be optimised by re-using in **r11** the value of **r22** that was computed when the present source electrode was the destination one. The first electrode will have to be computed also, although it is never used as a destination.

```
42 r11 = r11/n;
43 r22 = r22/n;
44 r12 = r12/n;
45 r21 = r21/n;
```

The following lines perform a Toeplitz operation on the averaged **r** vectors. The **R** matrices are square with a size equal to the number of samples (**n**.

R11 and **R22** can be computed once for each electrode, for the same reasons as **r11** and **r22**.

R12 must be computed for each unique combination of electrodes (i.e. every call).

```
46 %
47 R11 = toeplitz(r11);
48 R22 = toeplitz(r22);
49 R12 = toeplitz(r12,conj(r21));
50 %
```

The following lines take 1/100th of the first (top) element of each **r** columns and convert it into a matrix with values only along the diagonal. The **Dt** matrices are square with a size equal to the number of samples (**n**.

diag(diag(ones(L))) is equivalent to **Eye(L)**. Also, **Dt1** and **Dt2** can be computed once for each electrode, in the same manner as **r11** and **r22**.

```
51 %for regularization
52 Dt1     = 0.01*r11(1)*diag(diag(ones(L)));
53 Dt2     = 0.01*r22(1)*diag(diag(ones(L)));
54 %
```

Ri11 and **Ri22** can be computed once for each electrode, in the same manner as **r11** and **r22**.

Rn12 must be computed for each unique combination of electrodes (i.e. every call).

The **inv(R11 + Dt1)** operation is equivalent to **(R11 + Dt1)/eye(L)**. However, MathWorks recommend that the **inv()** function is avoided due to poor speed. Strangely, in the optimised function, the **inv()** function was not slower.

```
55 Ri11    = inv(R11 + Dt1);
56 Ri22    = inv(R22 + Dt2);
57 Rn12    = Ri11*R12*Ri22;
58 %
```

Si and **S** are column vectors with a number of rows equal to the resolution factor (**K**).

Si11 and **Si22** can be computed once for each electrode, in the same manner as **r11** and **r22**.

S12 must be computed for each unique combination of electrodes (i.e. every call).

```
59 Si11    = real(diag(F'*Ri11*F));
60 Si22    = real(diag(F'*Ri22*F));
61 S12     = diag(F'*Rn12*F);
62 %
```

The following is the final return of the function. There is nothing here to optimise.

```
63 %Magnitude squared coherence function
64 MSC     = real(S12.*conj(S12))./(Si11.*Si22);
```

3 The Original Function

```
1  function [MSC]=coherence_MVDR(x1,x2,L,K);
2
3  %% This program computes the coherence function between 2 signals
4  %% x1 and x2 with the MVDR method.
5  %% This algorithm is based on the paper by the same authors:
6  %% J. Benesty, J. Chen, and Y. Huang, "A generalized MVDR spectrum,"
7  %% IEEE Signal Processing letters, vol. 12, pp. 827-830, Dec. 2005.
8
9  %% x1, first signal vector of length n
10 %% x2, second signal vector of length n
11 %% L is the length of MVDR filter or window length
12 %% K is the resolution (the higher K, the better the resolution)
13
14 %initialization
15 xx1     = zeros(L,1);
16 xx2     = zeros(L,1);
17 r11     = zeros(L,1);
18 r22     = zeros(L,1);
19 r12     = zeros(L,1);
20 r21     = zeros(L,1);
21
```

```
%construction of the Fourier Matrix
F       = zeros(L,K);
l       = [0:L-1]';
f       = exp(2*pi*l*j/K);
for k = 0:K-1
    F(:,k+1) = f.^k;
end
F       = F/sqrt(L);

%number of samples, equal to the lenght of x1 and x2
n       = length(x1);

for i = 1:n
    xx1 = [x1(i);xx1(1:L-1)];
    xx2 = [x2(i);xx2(1:L-1)];
    r11 = r11 + xx1*conj(xx1(1));
    r22 = r22 + xx2*conj(xx2(1));
    r12 = r12 + xx1*conj(xx2(1));
    r21 = r21 + xx2*conj(xx1(1));
end
r11 = r11/n;
r22 = r22/n;
r12 = r12/n;
r21 = r21/n;
%
R11 = toeplitz(r11);
R22 = toeplitz(r22);
R12 = toeplitz(r12,conj(r21));
%
%for regularization
Dt1     = 0.01*r11(1)*diag(diag(ones(L)));
Dt2     = 0.01*r22(1)*diag(diag(ones(L)));
%
Ri11    = inv(R11 + Dt1);
Ri22    = inv(R22 + Dt2);
Rn12    = Ri11*R12*Ri22;
%
Si11    = real(diag(F'*Ri11*F));
Si22    = real(diag(F'*Ri22*F));
S12     = diag(F'*Rn12*F);
%
%Magnitude squared coherence function
MSC     = real(S12.*conj(S12))./(Si11.*Si22);
```

4 The Optimised Function

```
function [MSC]=coherence_MVDR_8(x1,x2,L,K,electrode1,electrode2,num_electrodes)

%% This program computes the coherence function between 2 signals
```

```
%% x1 and x2 with the MVDR method.
%% This algorithm is based on the paper by the same authors:
%% J. Benesty, J. Chen, and Y. Huang, "A generalized MVDR spectrum,"
%% IEEE Signal Processing letters, vol. 12, pp. 827-830, Dec. 2005.

%% x1, first signal vector of length n
%% x2, second signal vector of length n
%% L is the length of MVDR filter or window length
%% K is the resolution (the higher K, the better the resolution)

persistent pre_r
persistent pre_Ri
persistent pre_Si
persistent n
persistent F
persistent xx1
persistent xx2
persistent r12
persistent r21

%% Initialise constants once only
if isempty(F)
    n = L;
    %construction of the Fourier Matrix
    F       = zeros(L,K);
    l       = [0:L-1]';
    f       = exp(2*pi*l*j/K);
    for k = 0:K-1
        F(:,k+1) = f.^k;
    end
    F       = F/sqrt(L);
    pre_r      = zeros(L,num_electrodes);
    pre_Ri     = zeros(n,n);
    pre_Si     = zeros(K);
end

%% Initialise variables each time. Must be reset to zero since they are
% used later on in a summation
xx1      = zeros(L,1);
xx2      = zeros(L,1);
if electrode1 == 1
    r11_temp = zeros(L,1);
    r22_temp = zeros(L,1);
end
r12      = zeros(L,1);
r21      = zeros(L,1);

%%
for i = 1:n
    xx1 = [x1(i);xx1(1:L-1)];
    xx2 = [x2(i);xx2(1:L-1)];
    if electrode1 == 1
        if electrode2 == 2
            r11_temp = r11_temp + xx1 * conj(xx1(1));
        end
        r22_temp = r22_temp + xx2 * conj(xx2(1));
```

```
60      end
61      r12 = r12 + xx1*conj(xx2(1));
62      r21 = r21 + xx2*conj(xx1(1));
63  end
64
65  if electrode1 == 1
66      if electrode2 == 2
67          pre_r = r11_temp;
68      end
69      pre_r = [pre_r r22_temp];
70  end
71
72  %% Compute each electrodes matrices once only per electrode.
73  % in the first iteration of the source electrode, we do all the destination
74  % electrodes, apart from the first. So we only do the code here once (when
75  % electrode 1 is the source).
76  if electrode1 == 1
77      if electrode2 == 2
78          % First of all, do electrode 1, but only do it once
79          [Ri_temp Si_temp ] = PreCompute( pre_r(:,electrode1), n, L, F );
80          pre_Ri = Ri_temp;
81          pre_Si = Si_temp;
82      end
83      % Now do all the destination electrodes, but only once each
84      [Ri_temp Si_temp ] = PreCompute( pre_r(:,electrode2), n, L, F );
85      pre_Ri = cat( 3, pre_Ri, Ri_temp);
86      pre_Si = [pre_Si Si_temp];
87  end
88
89  %% Compute electrode-combination matrices
90  r12   = r12/n;
91  r21   = r21/n;
92  R12   = toeplitz(r12,conj(r21));
93  Rn12  = pre_Ri(:,:,electrode1)*R12*pre_Ri(:,:,electrode2);
94  %
95  S12   = diag(F'*Rn12*F);
96  %
97  %Magnitude squared coherence function
98  MSC   = real(S12.*conj(S12))./(pre_Si(:,electrode1).*pre_Si(:,electrode2));
99  end
100
101
102 %%
103 function [Ri Si] = PreCompute( r_temp, n, L, F )
104 persistent C_I
105 persistent R
106 persistent Dt
107
108 if isempty(C_I)
109     C_I = 0.01 * eye(L);
110 end
111     r = r_temp/n;
112     R = toeplitz(r);
113     Dt = r(1)*C_I;
114 %    Ri = (R + Dt)\eye(L);
115     Ri = inv(R + Dt);
```

```
116     Si = real(diag(F'*Ri*F));
117 end
```