BCI2000 and MATLAB

Robert Oostenveld, Ph.D.
Donders Institute
Nijmegen, NL
What is Matlab?

“MATLAB® is a high-level language and interactive environment that enables you to perform computationally intensive tasks faster than with traditional programming languages such as C, C++, and Fortran.”
Advantages of Matlab

interactive
simple syntax
no explicit declaration of variables and functions required
garbage collection
standard for neuroscience data analysis
many toolboxes available
many algorithms implemented
many data visualisation tools
Disadvantages of Matlab

- slower than compiled code
- needs more memory
- not open source
- expensive
- programming language suitable for RAD, not so much for large projects
How does Matlab compare to BCI2000

Matlab is a programming language
C++ is a programming language

BCI2000 is a software solution
  real-time acquisition
  signal processing
  feedback
  standardized interfaces and protocols
  lab-ready
To get started, select MATLAB Help or Demo from the Help menu.

>>
Function [filt] = preproc_bandpassfilter(dat, Fs, Fbp, B, type, dir)

% B PREPROC_BANDPASSFILTER applies a bandpass filter to the data and thereby
% removes the spectral components in the data except for the ones in the
% specified frequency band
%
% Use as
% [filt] = preproc_bandpassfilter(dat, Fs, Fbp, B, type, dir)

% Nyquist frequency
Fn = Fs/2;

% Compute filter coefficients
switch type
    case 'butter'
        if isempty(B)
            N = 4;
        end
        [B, A] = butter(N, [min(Fbp)/Fn max(Fbp)/Fn]);
    case 'fir'
        if isempty(B)
            N = 25;
        end
        [B, A] = fir1(N, [min(Fbp)/Fn max(Fbp)/Fn]);
end

% Apply filter to the data
switch dir
    case 'onpass'
        filt = filter(B, A, dat);
    case 'onpass-reverse'
        dat = flipd(dat);
        filt = filter(B, A, dat);
    case 'offpass-reverse'
        dat = flipd(dat);
        filt = flipd(filt);
end

copyright.m * preproc_bandpassfilter... *
Disadvantages of Matlab

- slower than compiled code
- needs more memory
- not open source
- expensive
- language suitable for RAD, not so much for large projects
Disadvantages of Matlab

*slower than compiled code*
needs more memory
not open source
expensive
language suitable for RAD,
    not so much for large projects
Real time processing for BCI

EEG alpha oscillation @10Hz
  duration ~100 ms
  decision every xx ms?

Processing as fast as possible
  real time
In computer science, real-time computing is the study of hardware and software systems which are subject to a "real-time constraint", i.e., operational deadlines from event to system response.

By contrast, a non-real-time system is one for which there is no deadline, even if fast response or high performance is desired or even preferred.
Real time processing for BCI

EEG alpha oscillation @10Hz
  duration ~100 ms
  decision every xx ms?

Processing as fast as possible
  real time

Deadline requirements vary, e.g.
  continuous classification -> hard deadline
  incremental evidence -> soft deadline
Real time processing for BCI
Motivation for combining BCI2000 and Matlab

- rapid application development
- try out various algorithms
- offline analysis of data
- port offline analysis to online

Matlab is fast enough for quite some computations
Matlab background

command line
  command.exe (DOS box)
  linux shell (bash, csh)
scripts
functions
BCI2000 background

System configuration and visualization are the main components of the BCI2000 architecture. The system consists of a source for brain signals, storage, signal processing, and an application for user interaction.

The spatial filter and normalizer are key components for preprocessing the data.
What is different in Matlab?
What is different in Matlab?
Limitations of Matlab engine

- Gets input from two sides
  - BCI2000 MatlabFilter
  - End-user (via command line)
- Executes only one task at a time
First hands-on experience: plotting data

start BCI2000 with
- SignalGenerator.exe
- MatlabSignalProcessing.exe
- FeedbackDemo.exe
load *matlabfilterdemo.prm* configuration
set config
start
First hands-on experience: plotting data

```matlab
whos

plot(bci_InSignal')

plot(bci_InSignal(1,:))

while(1); plot(bci_InSignal(1,:)); pause(0.01); end

while(1); bufplot(bci_InSignal'); pause(0.01); end
```
First hands-on experience: plotting data
Matlab in the pipeline

BCI2000 filters are pipelined
each filter is an C++ object
  data processing function
    Process()
  helper functions
    Preflight()
    Initialize()
    StartRun()
    StopRun()
  constructor
destructor
Matlab in the pipeline
Matlab in the pipeline

\texttt{constructor()} \quad % \text{define states and parameters,}
\quad % \text{start Matlab engine}

\texttt{Preflight()} \quad % \text{get parameter values}
\texttt{Initialize()} \quad % \text{check validity of params}
\texttt{StartRun()} \quad % \text{setup computational space}

\texttt{while data is streaming}
\quad \texttt{Process()} \quad % \text{process a single data block}

\texttt{StopRun()} \quad % \text{cleanup computational space}

\texttt{destructor()} \quad % \text{stop Matlab engine}
Matlab in the pipeline

\[\textit{bci_Construct()} \quad \%\text{ define states and parameters,}\]
\[\quad \%\text{ start Matlab engine}\]

\[\textit{bci_Preflight()} \quad \%\text{ get parameter values}\]
\[\textit{bci_Initialize()} \quad \%\text{ check validity of params}\]
\[\textit{bci_StartRun()} \quad \%\text{ setup computational space}\]

\[\textit{while data is streaming}\]
\[\quad \textit{bci_Process()} \quad \%\text{ process a single data block}\]

\[\textit{bci_StopRun()} \quad \%\text{ cleanup computational space}\]

\[\textit{destructor()} \quad \%\text{ stop Matlab engine}\]
Advanced hands-on: linear classification using a beamformer

beamforming

adaptive spatial filtering

biophysically motivated

adapted to data covariance

selective bandpass filter

optimal suppression
Adaptive spatial filtering
Adaptive spatial filtering

Ingredients: forward model

Region in the brain: \( r \)

Forward model: \( H(r) \)
Adaptive spatial filtering

Ingredients: forward model

Region in the brain: \( r \)
Forward model: \( H(r) \)
Assume neural activity: \( y_r(t) \)

Model for the projection of the source to the channels: \( x(t) = H(r) y_r(t) + n(t) \)
Adaptive spatial filtering

Ingredients: ‘inverse’ model

Model for the projection of the source to the channels:

\[ x(t) = H(r) y_r (t) + n(t) \]

Estimate the strength of activity of the neural tissue at location \( r \):

\[ s_r (t) = w(r)^T x(t) \]
Adaptive spatial filtering

Ingredients: ‘inverse’ model

Model for the projection of the source to the channels:

\[ x(t) = H(r) y_r(t) + n(t) \]

Estimate the strength of activity of the neural tissue at location \( r \):

\[ s_r(t) = w(r)^T y_r(t) + n(t) \]
Adaptive spatial filtering

Ingredients: ‘inverse’ model

Model for the projection of the source to the channels:

\[ x(t) = H(r) y_r(t) + n(t) \]

Estimate the strength of activity of the neural tissue at location \( r \):

\[ s_r(t) = w(r)^T H(r) y_r(t) + w(r)^T n(t) \]
An ideal spatial filter should pass activity from a location of interest with unit gain:

\[ w(r_0) H(r) = 1, \ r = r_0 \]

while suppressing others

\[ w(r_0) H(r) = 0, \ r \neq r_0 \]

However, this is not always possible, and we compute an optimal spatial filter by minimizing the variance of the filter output (source activity):

\[
\min \ var(s_r) \iff \min \ \text{trace}[w(r)^T \ \text{cov}(x) \ w(r)]
\]
Adaptive spatial filtering

Two constraints:
1. \( w(r) H(r) = 1 \)
2. \( \min \text{var}(s) \)

After some algebra:

\[
\begin{align*}
    w(r) &= \left[ H^T(r) \text{Cov}^{-1}(x) H(r) \right]^{-1} H^T(r) \text{Cov}^{-1}(x) \\
    \text{Spatial filter}
\end{align*}
\]
Adaptive spatial filtering

Two constraints:
1. \( w(r) H(r) = 1 \)
2. \( \min \text{var}(s) \)

After some algebra:

\[
w(r) = \left[ H^T(r) \text{Cov}^{-1}(x) H(r) \right]^{-1} H^T(r) \text{Cov}^{-1}(x)
\]

**Spatial filter**

1. the forward model for a given position \( r \)
Adaptive spatial filtering

Two constraints:
1. \( w(r) \ H(r) = 1 \)
2. \( \min \ \text{var}(s) \)

After some algebra:

\[
w(r) = \left[ H^T(r) \ \text{Cov}^{-1}(x) \ H(r) \right]^{-1} H^T(r) \ \text{Cov}^{-1}(x)
\]

Spatial filter

1. the forward model for a given position \( r \)
2. the covariance matrix of the data
Adaptive spatial filtering
Advanced hands-on: linear classification using a beamformer

\begin{verbatim}
constructor() % define states and parameters,  
             % start Matlab engine

bci_Preflight() % get parameter values
bci_Initialize() % check validity of params
bci_StartRun() % setup computational space

while data is streaming
  bci_Process() % process a single data block

bci_StopRun() % cleanup computational space

destructor() % stop Matlab engine
\end{verbatim}
Advanced hands-on: linear classification using a beamformer

start BCI2000 with
   SignalGenerator.exe
   MatlabSignalProcessing.exe
   FeedbackDemo.exe
load matlabfilterdemo.prm configuration
set config
start

look into Matlab source code, especially
   bci_StartRun.m and bci_Process.m
function bci_StartRun

% shared BCI2000 Parameters and states are global variables.
global bci_Parameters bci_States

% the following variables are used in the computation, and are needed over multiple iterations
global fsample nchans nsamples
global sum_covariance sum_count
global norm_s norm_ss norm_n
global H C w

fsample  = sscanf(bci_Parameters.SamplingRate{1}, '%fHz');
nchans   = sscanf(bci_Parameters.SourceCh{1}, '%d');
nsamples = sscanf(bci_Parameters.SampleBlockSize{1}, '%d');

% these are for the accumulated normalization
norm_n  = 0;
norm_s  = 0;
norm_ss = 0;

% these are for the accumulated data covariance estimate
sum_covariance = zeros(nchans, nchans);
sum_count      = 0;

% this is the forward model, which describes how the source projects onto the channels
% in real applications the forward model would be computed using a biophysical model
% but here the source projects equally strong onto all channels
H = ones(nchans,1)/nchans;

% these are empty to start with
C = zeros(nchans, nchans);
w = zeros(1,nchans);
function out_signal = bci_Process( in_signal )

% shared BCI2000 Parameters and states are global variables.
global bci_Parameters bci_States

<...>

flt_signal = in_signal;

% apply baseline correction
for i=1:nchan
    flt_signal(i,:) = flt_signal(i,:) - mean(flt_signal(i,:));
end

% compute the covariance using a running sum
sum_covariance = sum_covariance + flt_signal*flt_signal';
sum_count = sum_count + 1;

% compute the beamformer spatial filter
C = sum_covariance/sum_count;
w = inv(H' * inv(C) * H) * H' * inv(C);

% apply the beamformer spatial filter to the data
out_signal = w * flt_signal;

% compute the total power in the signal for the present block
out_signal = sqrt(sum(out_signal.^2));

% we could in principle stop here, but normalization of the control signal is required as well

<...>
function out_signal = bci_Process( in_signal ) % continued ... 

<...>

% the normalization could also be done with the BCI2000 C++ normalize filter

% if the signal is all zero, then the inverse covariance cannot be computed
% which results in a filter output that is not a number (nan)
if ~isnan(out_signal)
    % compute the running sum of the beamformer power
    for i=1:numel(out_signal)
        norm_n  = norm_n  + 1;
        norm_s  = norm_s  + out_signal(i);
        norm_ss = norm_ss + out_signal(i)^2;
    end
end

% compute the normalized output
norm_avg   = norm_s / norm_n;
norm_std   = sqrt(norm_ss/norm_n - norm_s^2/norm_n^2);
out_signal = (out_signal - norm_avg)/norm_std;
Matlab in the pipeline

Beamformer source estimate
  adaptive spatial filter
  implementation in few lines of Matlab code
  incremental updating of data covariance
  incremental updating of mean and stdev
  output used as control signal
Conclusions

Processing of small blocks of data in Matlab
Incremental algorithms
Within the pipeline

Useful for Rapid Application Development
Fast for Research & Development
Fast enough for many online applications
Once proven, port algorithm to C++