User manual sigViewer

Necessary functions and information for raw data extraction.

The raw data is gotten from the buffer.m through the function updateData:

```
%Updates the data: Waits for samples from buffer.m, and tracks at which
%sample it's currently at, how many updates it has gone through, and
%whether the loop in which it is called (for ~endTraining) should
%terminate.
%rawdat: Row for each electrode,column for each sample. The oldest data is
% deleted, and the new samples are inserted at the end. 1260 samples are
% remembered at a time.
%cursamp: Which sample it's currently at.
%nUpdate: Amount of updates it's done.
%loopBool: Whether the loop for ~endTraining should terminate. Only one
%that relies on the fig input.
function[rawdat, cursamp, nUpdate, loopBool] =  updateData(buffhost, buffport,opts, fig,
cursamp,updateSamp, rawdat, nUpdate, blkIdx, fs, iseeg)
%-------------------------------------------------------------------------
% wait for new data,
% N.B. wait_data returns based on Number of samples, get_data uses sample index = #samples-1
loopBool = true;
status=buffer('wait_dat',[cursamp+updateSamp+1 inf opts.timeOut_ms],buffhost,buffport); %wait for
either enough data or timeout
if( status.nSamples < cursamp+updateSamp ) %In case the timeout has passed.
        fprintf('Buffer stall detected...\n');
        pause(1);
        cursamp=status.nSamples; %Which sample it's currently at.
        if ( ~ishandle(fig) );
        loopBool = false;
        return;
        end;
elseif ( status.nSamples > cursamp+1*fs) % missed a 1 seconds of data
        fprintf('Warning: Cant keep up with the data!\n%d Dropped
samples...\n',status.nSamples-updateSamp-1-cursamp);
        cursamp=status.nSamples - updateSamp-1; % jump to the current time
end;
% Get the samples inbetween the current point and current+amount of samples added per update.
dat     =buffer('get_dat',[cursamp cursamp+updateSamp-1],buffhost,buffport); %Struct with one
field: .buf, which is 63x4.
cursamp=cursamp+updateSamp; %Update at which sample it's at for the next loop.
nUpdate=nUpdate+1;

% shift and insert into the data buffer
rawdat(:,  1     : blkIdx) =rawdat(:,updateSamp+1:end); %The data is shifted one update to the left
rawdat(:,blkIdx+1 : end) =dat.buf(iseeg,:); % The new data is inserted into rawdat

%if ( cursamp-hdr.nSamples > hdr.fSample*30 ) keyboard; end;
if ( opts.verb>0 ); fprintf('.'); end;
if ( ~ishandle(fig) );
        loopBool = false;
end;
end
```

The `wait_dat` function (17) from buffer returns how many samples it delivers, and the number of events. When the number of samples is sufficient, the get_dat function (31) gets the actual data.
this data consists of a row for each electrode and a column for each datapoint (sample). the get_dat buffer command gets 63 datapoints per electrode at a time.

this data is added to dat.buf (technically only the ones that conform to the iseeg boolean, but this should not be relevant for us).
rawdat contains a running history of the 1260 most recent datapoints. it removes the oldest points and then proceeds to add the data of dat.buf at every execution of updateData.
finally, the `rawdat` is preprocessed and stored in `ppdat`
my assumption is that the 63 samples is dictated by fs/updatefreq, rounded upwards. the updatefreq is set to 4, and fs to 250, giving 62.5, which rounded up matches with the number of samples.
to actually use any data we need to know how to interact with buffer. there are a few commands of note:

`get_hdr`
returns a struct-type header that denotes the characteristics of the data. this header consists of information regarding the number of channels, samples, events, the sample frequency, labels and datatype. this information is stored twice for compatibility reasons.

`get_dat`
takes in a 4 column by 63 row of data through bufClient, each column an electrode, each row a datapoint. this is transposed to the 63 column by 4 row format used throughout and passed on.

`get_evt`
returns events. this is not used in sigViewer so can be skipped for now.

`wait_dat`
in case there have not been enough new samples for a full update wait_dat waits until there either are enough samples or until a pre-set amount of time has passed. It returns the amount of samples it currently has.

`put_hdr, put_dat, put_evt`
sends (modifies) the header/data/event on the server, there are further commands, but they are not relevant for us at this point.

buffer connects with the server through reconnect. It creates a javaObject, which is an encapsulation of a java class within Matlab. The server/buffer construction appears to be made in java, which is good news for us. `nl.fcdonders.fieldtrip.bufferclient.BufferClientClock` keeps Matlab and java synced and connected. Fieldtrip (the toolbox upon which everything is built) has a reference implementation available. This reference implementation is not found in the described subdirectories in the buffer_bci we are working from. There is a very similar implementation in `\dataAcq\buffer\java\nl\fcdonders\fieldtrip\bufferclient`, however. We might want to abandon starting from buffer_bci and try to base our work on the fieldtrip source, or at least give it due attention. It can be downloaded here.

The frequencies for updating, sampling and other information needed for updateData are initialized in the function freqSamp:

```
%fs: sampling frequency. (250)
%times: [1x1260], gradual increasing values from ~(-5) to 0.
%blkIdx: Index where to insert new data. (max amount of data - amount of
% new samples per update)
%opts: trialLengthSamp gets updated (amount of data to plot for each channel).
%freqInd: Filter of frequencies that fit into the frequency bands in opts.
%noiseNormInd: Filter for noisefrac (not yet closely examined)
%updateSamp: How many samples are updated per round, in integer (63 in
% testing).
%noiseInd: Filter of frequencies that fit into the noise bands in opts
% (around 50 hz). Used for preprocessing
%freqs: list of all frequencies that can have data (0-125 in quickstart),
%used in the frequency domain.
%trialLengthSamp: part of opts, nr of samples that are retained in memory (1260)
function [fs, times, blkIdx, opts,freqInd, noiseNormInd,updateSamp,noiseInd,freqs] =
freqSamp(header,opts)
%Header may contain duplicate fields with(out) camel case..
if ( isfield(header,'fSample') );
        fs=header.fSample;
else
        fs=header.fsample;
end;

%Initialize trialLengthSamp in case it hasn't.
if ( isempty(opts.trialLengthSamp) && ~isempty(opts.trlen_ms) );
        opts.trialLengthSamp=round(opts.trlen_ms*fs/1000);
end;

updateSamp=ceil(fs/opts.updateFreq);
opts.trialLengthSamp=ceil(opts.trialLengthSamp/updateSamp)*updateSamp;
fprintf('tr_samp = %d updateSamp = %d\n',opts.trialLengthSamp,updateSamp);
blkIdx = opts.trialLengthSamp-updateSamp; % index for where new data gets inserted
if ( isempty(opts.downsample) || opts.downsample>fs )
        times=(-opts.trialLengthSamp+1:0)./fs;
else
        times=(-ceil((opts.trialLengthSamp+1)*opts.downsample/fs):0)/opts.downsample;
end

%Filter for the frequency domain.
freqs=0:1000/opts.welch_width_ms:fs/2; % 0 to (fs/2), with step size 1000/welch_width_ms
freqInd =getfreqInd(freqs,opts.freqbands);
if ( ~iscell(opts.noisebands) )
        opts.noisebands={opts.noisebands};
end;

%Filters of frequency for the 50hz and noisefrac filters.
noiseInd  =false(numel(freqInd),numel(opts.noisebands));
noiseNormInd=false(size(noiseInd)); %array with zeroes with the same size as noiseInd.
for ni=1:numel(opts.noisebands); % find the noise ind for each of the noise bands.
        noiseInd(:,ni)=getfreqInd(freqs,opts.noisebands{ni});
        nWidth=ceil(sum(noiseInd(:,ni))/2);
        nIdx=[find(noiseInd(:,ni)>0,1,'first') find(noiseInd(:,ni)>0,1,'last')]; % start/end of the
noise band
        noiseNormInd( max(1,   nIdx(1)-(1:nWidth))) =true; % nWidth bins before
        noiseNormInd( min(end, nIdx(2)+(1:nWidth))) =true; % nWidth bins after
end
end
```

In order to make updateData work properly, the following variables created here are necessary: fs, updateSamp and blkIdx. All variables with -Ind at the end are used for visualization and/or filtering in other modalities than a standard time domain plot. Freqs is used in the frequency domain. This function makes a lot of computations that shouldn't be too hard to replicate, but a deep understanding of the workings apart from what the output won't be necessary.

The 'header' that's needed as input for freqSamp is gotten through the buffer.m file as well, through the function chanInfo:

```
%Reads the header from the buffer and checks whether the head should be
%drawn.
%The header contains: number of channels, number of samples, number of
%events, the sampling frequency, the data type and the names of the
%channels.
function [header, opts] = chanInfo(buffhost, buffport, opts)
header=[];
header=buffer('get_hdr',[],buffhost,buffport); %Read data from the buffer server.
while ( isempty(header) || ~isstruct(header) || (header.nchans==0) ) % wait for the buffer to
contain valid data
        try
        header=buffer('get_hdr',[],buffhost,buffport);
        catch
        header=[];
        fprintf('Invalid header info... waiting.\n');
        end;
        pause(1);
end;

if ( isempty(opts.drawHead) )
        opts.drawHead=true; end;
end
```

This function is quite straightforward: it attempts to receive the header from buffer from r7-r17. It has to wait for this data, hence the loop. The last part at r19 is concerning whether a head should be drawn behind the plots.

The struct opts is used everywhere in the code, and in the functions above as well. it’s initialized in the beginning with these values:

```
function[opts] = initializeOpts()
%Find buffer.m in dataAcq. If it's not found, initPaths is called, which adds the required folders
to the path.
wb=which('buffer'); %find location of the buffer.m file
if ( isempty(wb) || isempty(strfind('dataAcq',wb)) );
        fprintf('Running
%s\n',fullfile(fileparts(mfilename('fullpath')),'../utilities/initPaths.m'));
        run(fullfile(fileparts(mfilename('fullpath')),'../utilities/initPaths.m'));
end;

%Makes a structure array. The structure is as such: name-field. Fields can
%be called with opts.name. General storage for important variables.
opts=struct('endType','end.training','verb',1,'timeOut_ms',1000,...
        'trlen_ms',5000,'trialLengthSamp',[],'updateFreq',4,...
        'detrend',1,'fftfilter',[.1 .3 45 47],'freqbands',[],'downsample',[],'spatfilt','car',...
        'adapthalflife_s',15,...
        'adaptspatialfiltFn','','whiten',0,'rmemg',0,...
        'rmartch',0,'artChBands',[.5 1 45 48],...
        'artCh',{{'EOG' 'AFz' 'EMG' 'AF3' 'FP1' 'FPz' 'FP2' 'AF4' '1'}},...
        'useradaptspatfiltFn','',...
        'badchrm',0,'badchthresh',3,'capFile',[],'overridechnms',0,...
        'welch_width_ms',1000,'spect_width_ms',500,'spectBaseline',1,...
        'noisebands',{{[23 24 26 27] [45 47 53 55] [97 98 102
103]}},'noiseBins',[],'noisefracBins',[.5 10],...%[0 1.75],...
        'sigProcOptsGui',1,'dataStd',2.5,'drawHead',1);
end
```

The variables in this struct are explained at the start of sigViewer. The first part searches for the buffer.m file, which is used to extract information from the bufferClient.

The only variables that are used as input but are not explained in the functions above, are fig and rawdat. Rawdat is initialized before the loop in which updateData is called:
`rawdat = zeros(sum(iseeg),outputSize(1));`
Which uses outputSize, computed after freqSamp is called:

```
outputSize=[opts.trialLengthSamp opts.trialLengthSamp]; %output size; amount of
data to plot for each channel x
if(~isempty(opts.downsample));
outputSize(2)=min(outputSize(2),
round(opts.trialLengthSamp * opts.downsample /fs));
end;
```

This should be a sufficient amount of building blocks for the visualization of the raw data.