CARMEN Project

Introduction to CARMEN Services using the Matlab NDF Toolbox

21st September 2011

Michael Weeks
9/21/2011

Contents

Introduction

This is an introduction to writing applications using the Matlab NDF toolbox, for use as CARMEN services. The Matlab NDF toolbox provides an API that simplifies the manipulation of NDF files and structures. The toolbox must be downloaded and installed, and then the Matlab path pointed at the toolbox.

Documentation is available for the NDF toolbox, and this must be used in conjunction with the following examples.

NDF services follow the same rules as for normal services, in that they should be non-interactive command-line applications, with output details specified in the XML-like <output> tags. However the richer features available to NDF, mean that there are some slight differences, especially for command line parameters.

NDF Service Input XML Form

For CARMEN services, we take a command line application and wrap them in to a service for execution on our servers. Input file (names) or values are passed into the application via the command line parameters. An example of a (non-NDF) High Pass Filter command line might be;

```
hpfNdf.exe  rawdata.mat  1000
```

where “rawdata.mat” is the name of the data file containing the time series data, and “1000” Is the filter’s required cut-off frequency.

However, for an NDF service version of this application, there are additional considerations concerned with the richness of NDF. When running a service which takes an NDF input file, in addition to which NDF file to select, the user has additional choices, such as;

- Data type – several different datatypes can be stored within an NDF file. These are time-series, segment, neural-event, event, generic-matrix, user-defined, and image data;
- Members – a member is a collection of channels within an NDF file that are grouped by a common theme, i.e. all the channels from a particular recording, or service.
- Channels – Inside a member, there may be one or more channels of data, such as a time-series recording from a sensor. Any or all of the channels may be selected.
- Start & end time offsets – The data in the input NDF may have a start and end time. The user can select the whole recording or a section of the recording based on offsets from the start and end.

Al the above selections can be seen working in the CARMEN portal when running an NDF service that takes an input NDF. Figure 1 shows a part of the Service input NDF selection panel, showing channel and time offset selection.

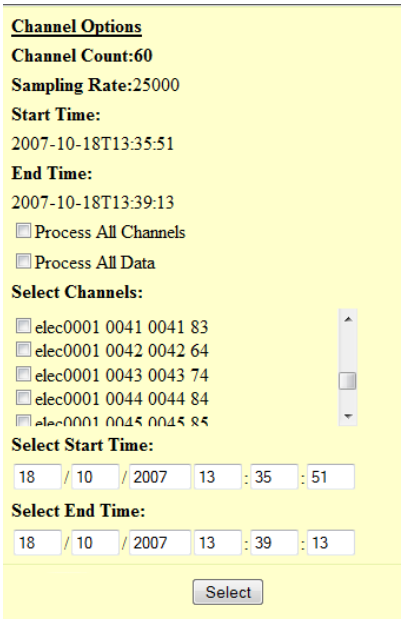

Figure 1 - NDF input selection panel

Once the user has selected an input NDF file, the NDF selection panel appears, allowing the user to select the above details if required. Once the user makes the selection, the portal generates an NDF Service Input XML form containing this information, and this is what we pass into the service's command line parameter. So to correct the above example command line, we would write the code to handle the following command line parameters;

```
hpfNdf.exe  rawdata.xml  1000
```

Where rawdata.xml is the generated NDF form containing the user selections, and 1000 is the filter cut-off frequency.

For developing code, and test purposes, the xml must be hand generated by the service creator.

The layout of the NDF Service Input XML form is as follows:

```
<ndfserviceinput>
   <filename></filename>
   <datatype></datatype>
   <member>
      <memberindex></memberindex>
      <channelindex></channelindex>
      <startindex></startindex>
      <endindex></endindex>
   </member>
</ndfserviceinput>
```

The fields in the NDF Service Input XML form are;

1. Filename – The NDF filename (i.e. the NDF XML header) of the input. Note that this relates to the XML header, which has the ".ndf" file extension. The attached data files are referenced from the XML header. Only one NDF filename may be specified.

2. Data type – this can be one of several NDF data types. The following table shows the supported data types and the value to be entered into the NDF Service Input XML form. We may only specify one data type in an NDF Service Input XML form. If the service requires multiple data types from the same NDF file, a separate input parameter (again as an NDF Service Input XML form) is required.

Dataset type	value
Time-series dataset set	timeseries
Neural event data set	neuralevent
Event data set	event
Segment data set	segment
Generic matrix data set	matrix
User-defined data set	userdefined
Image data	image

3. Member index – this is the member index extracted from the NDF header. Within an NDF file, there may be multiple collections of recordings based upon a common theme, and these are placed within separate ***members*** within the NDF header file. For instance an NDF file may contain 10 channels of raw data, and 10 channels of high pass filtered data. These would therefore be kept in separate members. Only one member index may be specified.
4. Channel Indices – a comma-separated list of channel indices, or "-1" for all channels (within a member);
5. Start Index – the offset index from the beginning of the recording, or "-1" to use the first index. This field applies to all channels within a member. Note that this is calculated by the NDF input selection panel interface, where it is displayed to the user in time units.
6. End Index – the offset index from the end of the recording, or "-1" to use the last index. This applies to all channels within a member. Note that this is calculated by the NDF input selection panel interface, where it is displayed to the user in time units.

Note that the XML uses the zero-based index convention. When using this with the Matlab NDF toolbox function ***`ndfsvcread()`***, indices are automatically converted to the matlab convention of 1-based indexing.

Below are some examples of NDF Service input forms. The first example describes the required inputs from the input NDF file called opAll.ndf. The data is time-series, and we are looking at the memberID 1 (look at the NDF header to see this), and the channel indexes 1, 2 and 3. Note that we deal with indices and not the actual channel name as displayed in the portal NDF selector panel. The start and end indices specify that the recording section of interest is from 20000 to 79999. Again, this is displayed on the portal NDF selector panel as time from the start and end of the recording.

```
<ndfserviceinput>
  <filename>opAll.ndf</filename>
  <datatype>timeseries</datatype>
  <member>
    <memberindex>1</memberindex>
    <channelindex>1,2,3</channelindex>
    <startindex>20000</startindex>
    <endindex>79999</endindex>
  </member>
```

</ndfserviceinput>

For Neural-Event, Segment and Event data, the start and end index is irrelevant, therefore start and end times must be used, i.e.;

```
<ndfserviceinput>
  <filename>ip.ndf</filename>
  <datatype>neuralevent</datatype>
  <member>
    <memberindex>0</memberindex>
    <channelindex>1,2,3</channelindex>
    <timefrom>0</timefrom>
    <timeto>1</timeto>
  </member>
</ndfserviceinput>
```

For generic matrix, user-defined, and image datasets, there will be no member information, only the base framework.

```
<ndfserviceinput>
  <filename>ip.ndf</filename>
  <datatype>image</datatype>
??????????????????????
</ndfserviceinput>
```

NDF Services and Time-Series Data

A time series consists of a series of signal measurements (digitised or real values) that were measured at discrete time intervals. The data consists of an Nx1 matrix, where N is the number of samples in the recording period. The data may consist of multiple channels of time-series signals.

Next, create the top-level matlab function. In this example we are creating a high pass filter service, and the inputs are an NDF input file containing time-series data (*inputXml*), and a filter cut-off frequency (*cuttoffFreqStr*).

```
function hpfNdf (inputXml, cutoffFreqStr)
cutoffFrq  = str2double(cutoffFreqStr);
```

Note that the all the input parameters coming in via the command line are strings. For the NDF input XML a string is fine, but the cut-off frequency is required to be converted to a numeric, so we need to use the str2double() function.

The next step is to create an NDF output object and populate it with some general information. The name of the output NDF file is fixed at “outputHpf.ndf”, though this could be derived from the input file, or there could be an additional input parameter which allows user’s to specify the output NDF file name. This is totally up to the service creator.

```
% create a general data info object & populate it with info
genInfo = ndfgeneraldatainfo;
genInfo.DataInfo.description = 'High Pass Filtered time-series data';
genInfo.DataInfo.laboratory = 'NDF HPF on the Carmen VLE';
% create an output NDF object, add it's filename and genInfo
opNdfFilename = 'outputHpf.ndf';
opNdfObj = ndfwrite(opNdfFilename);
opNdfObj.adddatainfo(genInfo);
```

Now the output NDF object has been prepared, we can open the input NDF file containing the raw data. We have created a special version of ndfread(), called ***ndfsvcread()*** that takes the NDF Service Input XML form as a parameter. We can then access it using ***getallchannelidx()*** to determine the list of channel indices that it needs to process.

```
% read in ndf input XML form
ndfObj = ndfsvcread(inputXml);
% determine which channels to access
indexList = ndfObj.getallchannelidx();
```

We can now loop through all the channels, and retrieve the channel information. From this we can determine the sampling rate for the channel data.

```
for i=1:size(indexList, 1);
   chInf = ndfObj.getspecdatainfo(indexList[i]);
   samp = chInf.DataInfo.samplingRate;
```

We then need to determine the member ID, and find the start and end indices for the channel data. If no start or end indices are set (i.e. the user selected to use all the data), the start and end indices must be set to "**-1**".

```
   memID = chInf.DataInfo.memberID; % memID = zero if no members set
   memInf = ndfObj.getparamfromID(memID);
   startIdx = -1;
   endIdx = -1;
   if ~isempty(memInf) % we have a member!
      startIdx = memInf.startIndex;
      endIdx = memInf.endIndex;
   end
```

Now we can fetch the raw time-series data from the NDF input file and place it in a buffer

```
   rawdata = ndfObj.getspecdata(chIdx, startIdx, endIdx);
   if rawdata.DataInfo.itemCount == 0;
      clear rawdata.Data;
      break;
   end;
```

Data can be stored in a time-series channel as either the digitised ADC step count values or as real values. The key to determining which type of data is to check the channel's ADC resolution setting. If the ADC resolution is zero then the channel data contains real values. If it is zero, the channel data

contains the digitised step count from the ADC, and typically will require conversion to real values before processing can continue.

The following ADC parameters can be read from the channel info's DataInfo struct.

Parameter	Description
adcZeroOffset	The zero offset value
adcPrecision	The ADC precision in bits
adcResolution	The ADC step size or ZERO if real value data is stored
adcValueUnit	The unit of measurement (i.e. volts, amps, etc)

So to check which format the data is in and to automatically convert it, the following Matlab code can be used.

```
if chInf.DataInfo.adcResolution ~= 0
   rawdataBuf = chInf.DataInfo.adcZeroOffset
              + (double(rawdata.Data) * chInf.DataInfo.adcResolution);
else
   rawdataBuf = double(rawdata.Data);
end
clear rawdata.Data;
```

Now you have all you need to perform the low pass filter operation; the data, the sampling frequency, and the required cut-off frequency. We will assume that this is now performed and the resulting high-pass filtered time-series data are placed in a buffer (as doubles) called ***opBuffer***. However, before we do anything with the output data we will record some NDF information for the output channel we have just created. A lot of the information we need is available in the chInf data structure, so rather than copy it all over to a new structure, we can simply re-use it, adding any new information as necessary. For instance, we can record the details of the HPF method that we used.

```
chInf.DataInfo.hpfType = 'butter';
chInf.DataInfo.hpfOrder = int32(8);
chInf.DataInfo.hpfCutoffFreq = cutoffFrq;
```

If the start and end offsets have been selected, we also need to set the end time and the start offset. Note that the Start time remains the same as the input data, but we add an offset.

```
if endIdx ~= -1
   % get start time in serial date number format
   dateStr = [ chInf.DataInfo.startDate 'T' chInf.DataInfo.startTime '.'
             num2str(int32((chInf.DataInfo.startDecimalSecond * 1000)))];
   dateNum2 = datenum(dateStr, 'yyyy-mm-ddTHH:MM:SS.FFF');
   % set serial date number to end date
   daysecs = 86400;
   runTime = (endIdx/samp)/daysecs;
   dateNum2 = dateNum2 + runTime;
   chInf.DataInfo.endDate = datestr(dateNum2, 'yyyy-mm-dd');
   chInf.DataInfo.endTime = datestr(dateNum2, 'HH:MM:SS');
   chInf.DataInfo.endDecimalSecond = str2double(['0.' datestr(dateNum2,
'FFF')]);
end
% set the output start-time offset
if (startIdx ~= -1 && startIdx ~= 1)
   chInf.DataInfo.timeOffset = chInf.DataInfo.timeOffset + ((startIdx-1)
                             / chInf.DataInfo.samplingRate);
```

```
end
```

We now need to create a container for the output NDF datatype, in this case it's time series data

```
opTsData = ndftimeseriesdata;
```

We can now add the output time-series data to that object, in this case we are digitising the output into 16-bit integers, so that it can be stored in a quarter of the space required for real numbered doubles. Note that the digitise function is described later. The **Data** buffer inside of the **opTsData** container is used as the destination for the digitised data, and the ADC settings are added to **chInf**.

When writing the data to NDF, remember to set the above ADC parameters accordingly. By default the adcResolution is set to zero.

```
adcBits = 14;
[opTsData.Data, chInf.DataInfo.adcZeroOffset, chInf.DataInfo.adcResolution]
= digitise (opBuffer, adcBits);
chInf.DataInfo.adcPrecision = adcBits;
clear opBuffer;
```

Then we can attach the channel information to the **opTsData** container.

```
opTsData.DataInfo.dataFilename = [outNdfFile '.mat'];
% set variable names to the same as the ones in the input NDF
opTsData.DataInfo.varName = chInf.DataInfo.varName;
opTsData.DataInfo.matLabel = chInf.DataInfo.matLabel;
```

And write the channel data and information to the NDF file

```
opNdfObj.writedata(opTsData);
opNdfObj.updatedatainfo(opTsData, chInf);
opNdfObj.adddatainfo(chInf);
```

You can now do tidying up, clearing any buffers that are no longer used, etc, and complete the **for** loop that processes all the channels. Therefore, if there are multiple channels to process, the above process occurs again within the loop.

Once all the channels have been processed, you can update the NDF history information and write the NDF header file to disk.

```
opNdfObj.history = ndfhistory(['ndfHPF ' inputXml ' ' cutoffFreqStr],
             'NDF High Pass Filtered Time Series Data - output redigitised
             using non-original ADC values','');
opNdfObj.history + ipNdfObj.history; % add input data history (provenance)
opNdfObj.writendfheader;
```

Lastly, we need to inform the CARMEN system of our outputs, clear any open figures, and then close the top-level function. There will be one output NDF file containing the filtered time-series data, so we add the usual XML-like output tags.

```
close all force;         % kill figures
disp(['<output>'opNdfFilename '</output>']);
return;
```

Now you can test the service code in the matlab environment. You will need to create your own NDF Service input XML file, with details that match your input NDF file containing the raw time-series

data. The result should be an NDF file, and data (mat) file on disk, and the name of the NDF file (within the XML-like output tags) printed to the stdout output screen.

Time-Series Digitisation

A useful function for digitising a time-series signal is shown below. This takes an input buffer (typically 64-bit double) containing time-series data, and the number of bits for the digitised result. The function returns a buffer containing the digitised values (16-bit integer), the zero-offset value and the step size;

```
%% function to digitise the output buffer, so reducing disk space
function [opBuf, zeroOffset, adcResolution] = digitise(ipBuf, adcBits)
    minVal = double(min(min(ipBuf)));
    maxVal = double(max(max(ipBuf)));
    numberSteps = (2 ^ adcBits) - 1;
    range = double(maxVal - minVal);
    zeroOffset = minVal + (range/2);
    adcResolution = (range * 1.05) / numberSteps;
    % fix to stop divbyzero error if all values the same
    if adcResolution == 0
        adcResolution = 1;
    end
    opBuf = int16((ipBuf - zeroOffset) / adcResolution);
end
```

Adding Image files to the NDF Output

In the previous HPF example, we took some raw time-series data, processed it and produced a new NDF output file. You may want to graph the output (or input, or both) of each of the channels and add them to the NDF output, along with the data. The following snippet shows how this can be achieved. In this example there is a variable called display which may be set via an additional command-line parameter, which if set to “yes” will generate the plot, save it to file and add it to the NDF. This may be ignored if the jpeg is always to be created.

```
opJpegFile = ['opHpf_' chInf.DataInfo.matLabel];
if (strcmpi(display, 'yes'))
   plot (opBuffer);
   opJpegFname = [opJpegFile '_hpf.jpg'];

   % write jpeg to file and gather info without needing X-server
   set(gcf, 'Units', 'Pixels');
   pos=get(gcf, 'Position')/100;
   set(gcf,'PaperUnits','Inches');
   set(gcf, 'PaperPosition', pos);
   print ('-djpeg', jpegFileName, '-r100');

   % add file & info to NDF file
   imagedatainfo = ndfimagedatainfo;
   info = imfinfo( jpegFileName);
   imagedatainfo.DataInfo.imageWidth = info.Width;
   imagedatainfo.DataInfo.imageHeight = info.Height;
   imagedatainfo.DataInfo.dataFilename = jpegFileName;
   imagedatainfo.DataInfo.acquisitionEquipment
       = ['HPF time-series data for channel ' ...
         chInfHpf.DataInfo.matLabel '. Generated by NDF HPF service'];
   opNdfObj.adddatainfo(imagedatainfo);
end
```

Note that we have to determine the width and height of the image without any graphics display help. This is due to the service ultimately running on a remote server with possibly no graphics display available.

NDF and Neural-Event Data

Neural-Event data consists of a collection of indices where an event has occurred, such as the occurrence of spikes in a time-series. A channel consists of an Nx1 matrix, where N is the number of spikes found. The data may consist of multiple channels of data.

To load some NDF Neural event data into some code to be used as a CARMEN service, we need the NDF Service input form again to pass in the additional details. This passed into the service via the command line as shown above, and loaded into the ndfsvcread function to create a new NDF object. From this we can determine the channels indices to process.

```
function nevService (ipSpikesFileXml, . . .)
   ipSpkObj = ndfsvcread(ipSpikesFileXml);
   indexListSpk = ipSpkObj.getallchannelidx();
```

We then need to loop through all the channel indices, extracting the data info from each as we go.

```
for channels = 1:size(indexListSpk, 1);
    % get input channel indices
    chIdxSpk = indexListSpk(channels);
    % get input channel info from both input NDFs
    chInfSpk = ipSpkObj.getspecdatainfo(chIdxSpk);
    % get sampling rate
    sampSpk = chInfSpk.DataInfo.samplingRate;
```

Then get the start and end indices for the data as selected by the user (and specified in the NDF service input XML)

```
% get start/end indices from input xml files
memIDSpk = chInfSpk.DataInfo.memberID; % memID = 0 if no members set
memInfSpk = ipSpkObj.getparamfromID(memIDSpk);
startTimeSpk = -1;
endTimeSpk = -1;
if ~isempty(memInfSpk) % we have a member!
   startTimeSpk = memInfSpk.timeFrom;
   endTimeSpk = memInfSpk.timeTo;
end
```

Then get the spike times

```
[startIdxSpk endIdxSpk] = ipSpkObj.getspecindexes(startTimeSpk,
                            endTimeSpk, chIdxSpk);
```

```
spkidx = ipSpkObj.getspecdata(chIdxHpf, startIdxSpk, endIdxSpk);
if spkidx.DataInfo.itemCount == 0;
   clear spkidx.Data;
   break;
end;
```

The spike time data now resides in the spkidx.Data buffer and can be used, along with the data information in the DataInfo structure.

To save the neural event data, we need to have an NDF object and give it some general information about its contents.

```
% create a new NDF container object
opNdfFilename = ['output.ndf'];
ndfOpObj = ndfwrite(opNdfFilename);
% Add general information to NDF
genInfo = ndfgeneraldatainfo;
genInfo.DataInfo.description = 'NDF Neural-event data example';
genInfo.DataInfo.laboratory = 'Generated from the CARMEN VLE';
ndfOpObj.adddatainfo(genInfo);
% create new data info object
opNeDataInfo = ndfneuraleventdatainfo;
```

and then loop through all the channel indices as before and perform whatever functionality is required by the service.

```
%% Loop through the channel indices
for i = 1:size(indexList, 1);
   ...
   % do processing
   ...
```

Then once the results data have been collated for each channel, create a container for the neural event data

```
opNeData = ndfneuraleventdata;
```

Then set the sample rate, start/end times and start offsets in the datainfo object. Note that in this example the output data was derived from an input NDF file containing high-pass filtered time-series data. As such, "***samp***" (sampling rate), "***chInf***" and "***hpsig***" are taken from the input NDF file.

```
% set the sample rate
opNeDataInfo.DataInfo.samplingRate = samp;
%% Set the start time, end time and offset if we chop the recording down
    % set start time
    opNeDataInfo.DataInfo.startDate = chInf.DataInfo.startDate;
    opNeDataInfo.DataInfo.startTime = chInf.DataInfo.startTime;
    opNeDataInfo.DataInfo.startDecimalSecond = chInf.DataInfo.startDecimalSecond;
    % set end time
    % note that it is unlikely that we will get an empty end time as the
    % portal should generate one of it is missing
    if endIdx ~= -1 || strcmp(chInf.DataInfo.endDate, '')
        % get start time in serial date number format
        dateStr = [ chInf.DataInfo.startDate 'T' chInf.DataInfo.startTime '.'
                    num2str(int32((chInf.DataInfo.startDecimalSecond * 1000)))];
        dateNum2 = datenum(dateStr, 'yyyy-mm-ddTHH:MM:SS.FFF');
        % set serial date number to end date
        daysecs = 86400;
```

```
        if endIdx ~= -1
            runTime = (endIdx/samp)/daysecs;
        else
            runTime = (double(hpsig.DataInfo.itemCount)/samp)/daysecs;
        end
        dateNum2 = dateNum2 + runTime;
        opNeDataInfo.DataInfo.endDate = datestr(dateNum2, 'yyyy-mm-dd');
        opNeDataInfo.DataInfo.endTime = datestr(dateNum2, 'HH:MM:SS');
        opNeDataInfo.DataInfo.endDecimalSecond = str2double(['0.'
                                                datestr(dateNum2, 'FFF')]);
    else
        opNeDataInfo.DataInfo.endDate = chInf.DataInfo.endDate;
        opNeDataInfo.DataInfo.endTime = chInf.DataInfo.endTime;
        opNeDataInfo.DataInfo.endDecimalSecond = chInf.DataInfo.endDecimalSecond;
    end
    % set the output time offset
    if (startIdx ~= -1 && startIdx ~= 1)
        opNeDataInfo.DataInfo.timeOffset = chInf.DataInfo.timeOffset
                    + ((startIdx - 1) / chInf.DataInfo.samplingRate);
    end
```

We can now update the relevant NDF info and save the channel data to disk. Note that results from the service are stored in the working buffer called "***opNEvBuffer***", and contains the event times.

```
    %% save the results
    opNeData.DataInfo.dataFilename = ['outputNEv.mat'];
    % set the variable names to the ones in the input files
    opNeData.DataInfo.varName = chInf.DataInfo.varName;
    opNeData.DataInfo.matLabel = chInf.DataInfo.matLabel;
    opNeData.Data = opNEvBuffer;

    % Now write the channel data to file & tidy up temp buffers
    ndfOpObj.writedata(opNeData);
    ndfOpObj.updatedatainfo(opNeData, opNeDataInfo);
    ndfOpObj.adddatainfo(opNeDataInfo);
    clear hpsig.Data;
    clear opNeData.Data;
    clear opNEvBuffer;
```

Once all the channels have been processed and the results data stored to disk, we can exit the channel iterator loop, update the history, save the NDF header file, and finalise the service.

```
%% add history to output
cmdLine = 'service.exe ip.ndf';
ndfOpObj.history = ndfhistory(cmdLine, 'Spike time generator service' , '');
ndfOpObj.history + ndfIpObj.history;
%% write the NDF header to disk
ndfOpObj.writendfheader;
%% kill figures, and inform the Carmen system of the output file
close all force
disp(['<output>' 'outputNev.ndf' '</output>']);
% close the top-level function
end
```

NDF and Segment Data

Segment data consists of an array of sections of time-series data. The time-series data is typically taken from a single channel and each time-series section is based upon an event. The data includes an event time matrix, and can also include a classification matrix should the segments be sorted and classified.

NDF and Generic Matrix Data