The knife-alien sound processor

User manual

Project assignment in the course
Engineering applications using MATLAB, TNG016
at The Institute of Technology at Linköping University

by

Patrik Dahlström, Daniel Josefsson, Staffan Sjöqvist

ED3

Supervisor: Qingxiang Zhao
ITN, Linköping university

Examiner: Qingxiang Zhao
ITN, Linköping university

Norrköping, 19. May 2011

Purpose

The knife-alien sound processor is a powerful tool to visualize and modify audio data in real time. It captures audio from the computer's sound card, displays the spectrum of the captured signal and can optionally use filters to modify the signal. By also displaying the spectrum of the processed signal the effects of the filtering can be clearly seen. The audio filters can be easily reconfigured using sliders, even during processing.

After filtering, the processed audio is saved to disk in a wav file.

System requirements

Knife-alien has been developed and tested using the GNU/Linux operating system and MATLAB 7.12.0.635 (R2011a). Therefore, the application cannot be guaranteed to work properly with other operating systems or other versions of MATLAB. The project group does not offer support for other configurations than the one mentioned in this paragraph.

Quick introduction to the application

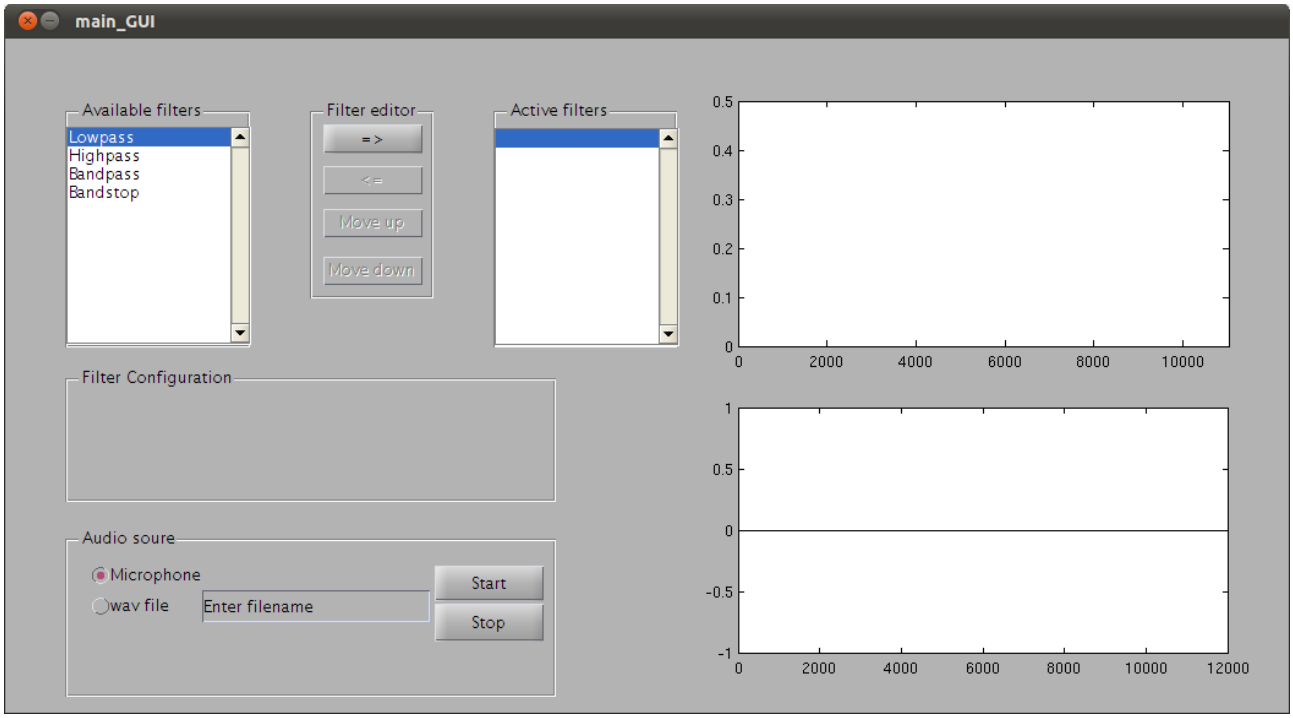

Figure 1: GUI after opening the application.

The design team concentrated on developing a clean GUI where the functionality were prioritized over the looks. The GUI can be divided in to three parts which manages different tasks within the application.

Selection of audio source

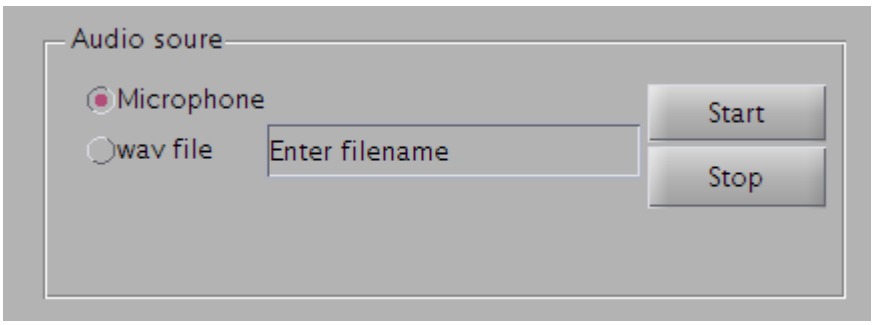

Figure 2: Panel holding options for audio source selection.

The audio source selection panel is very straight forward and gives the user the option to choose between using a microphone or a pre-made wave file as input to the application. The microphone is selected as the default input choice.

The start and stop button will activate and deactivate the audio player or recorder.

The GUI will run even if a microphone is not detected.

Filter handling

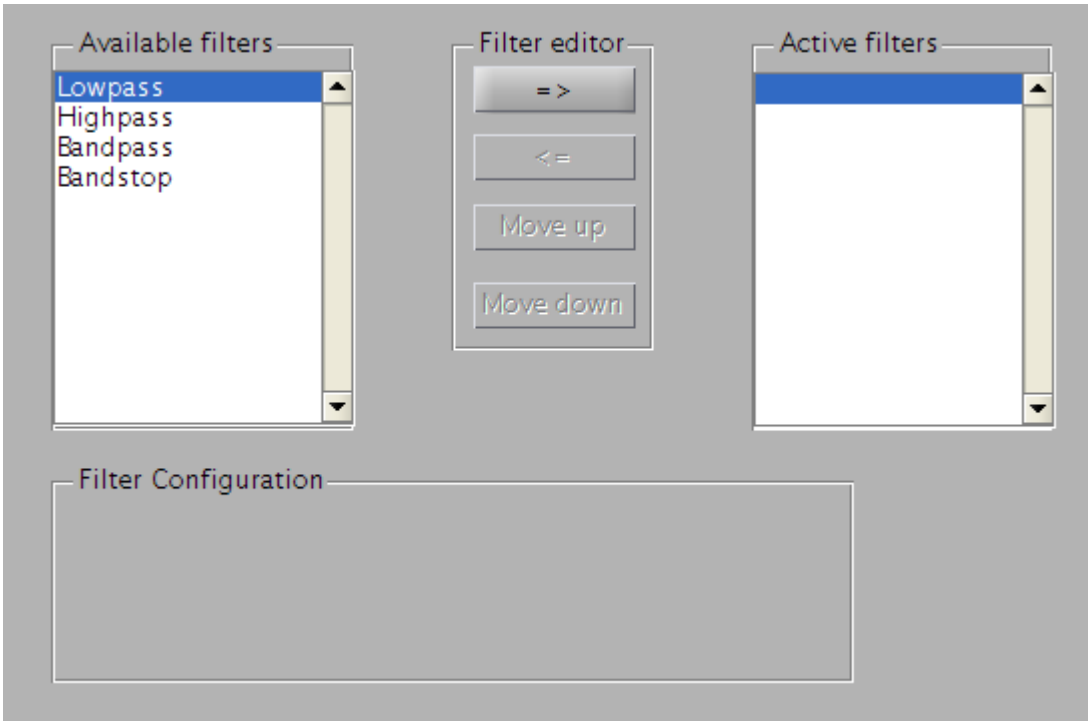

Figure 3: Controls that handles the filter configuration.

The filter editor panel (top middle of Figure 3) manages the underlying process of filter creation, deletion and structure of the filter chain. The push buttons in the panel are only active when the result of a button push is relevant, mainly for making it easy for the user to control the filter chain. This is why the bottom three push buttons are greyed out when the GUI starts. How it looks when all of the buttons are active can be seen in Figure 5 .

The panel Available filters is populated with the different filters that are currently implemented. It is also possible to define user specific filters. How this works can be read about in the chapter How to add filters to knife-alien sound processing application.

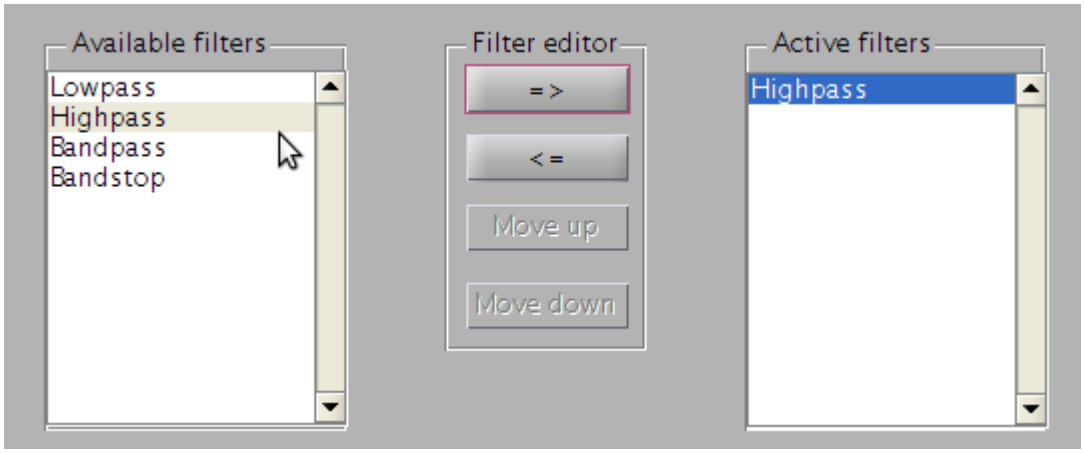

Figure 4: Active filters populated with a added filter.

The user pushes the topmost button in filter editor panel to add a filter to the filter chain. All filters that are added connects to each other in a linked list, i.e each filter knows its next and previus filter.

The chain is also populated with two hidden filters that merely connects the filter chain with the audio capture device and the output graphs. All the filters added by the user are added in between these dummy filters and are visible in the Active filters panel.

The user can also remove filters from the filter chain by marking the unwanted filter and press the second uppermost button. The remove button is enabled in Figure 4.

The two bottommost buttons are used to change the filter position in the filter chain. This is useful to get the desired filter output.

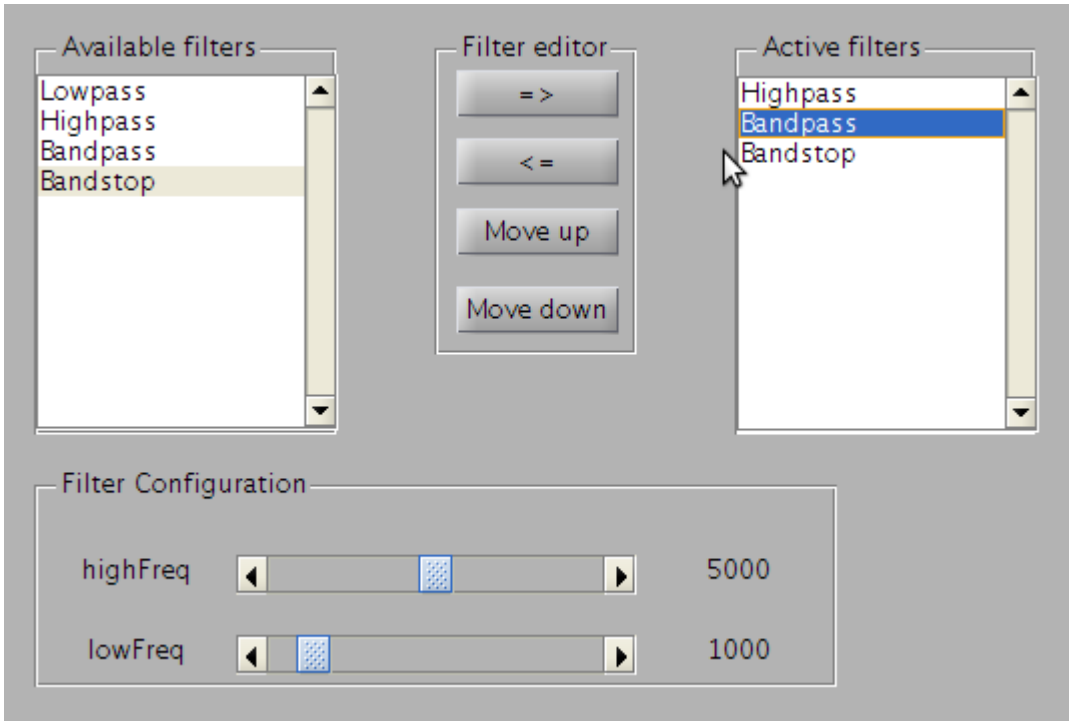

Figure 5: The middle filter is moveable both up and down in the filter list. In the bottom one can see the parameter sliders and current setup.

The filter parameters are viewable and editable when the user clicks on a filter in the Active filter panel. The Filter Configuration panel will be populated with sliders and information about the selected filter. This can be seen in Figure 5.

Editing a filters parameters can be done when executing a filtering while add and remove filters automatically will stop the filtering for a short while.

Visualization of input and output audio

The topmost graph contains the unfiltered Fourier transformed input audio data and the bottom one contains the filtered Fourier transformed data.

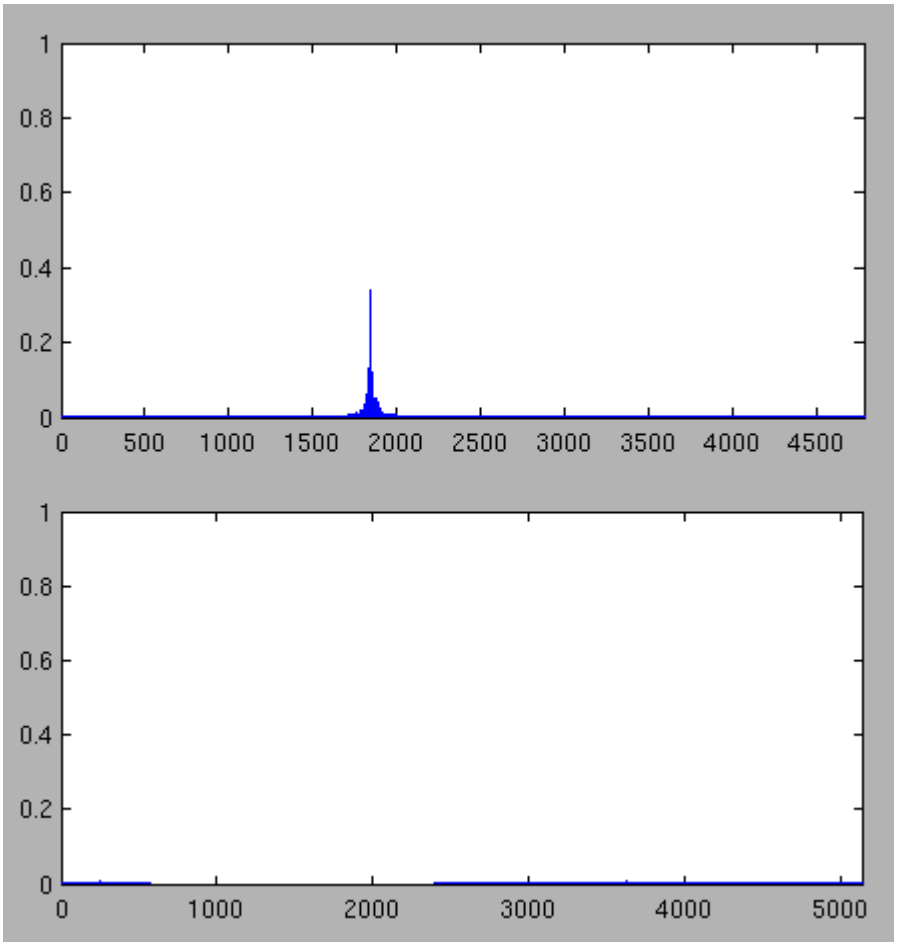

Figure 6: A whistle blow as input and the output of a band stop filter.

How to add filters to knife-alien sound processing application

Filters in knife-alien works by letting Fourier transformed data progress through a linked list of filters, starting and ending with a dummy filter. Thus, when no filters are selected in the GUI there are still two dummy filters present which the data moves through. The first and last dummy filters are available in the *handles* structure, present in all GUI callbacks, as *handles.firstDummy* and *handles.dummy*.

When either an audio recorder or audio player notifies the event *'NewAudioData'* the audio data is passed to *firstDummy* and then progress through the link of filters, and eventually the last filter (*dummy)* notifies *'FilteringComplete'*. It is also possible to listen to events by filters in between these two dummy filters.

To get a better understanding of how the filters work, consider the filter superclass file *+Filters/@FilterClass/FilterClass.m*:

```
classdef (Hidden=true) FilterClass < handle
    properties (Hidden=true)
        Fs
        userData
        Next
        Prev
    end
    properties (SetAccess = protected, Hidden = true)
        Data
        Name
    end
    methods (Abstract=true)
        filteredData = filter(obj,data)
    end
    methods
        function eventHandler(obj,src,eventData)
            obj.filter(src.Data);
        end
    end
    events
        FilteringComplete
    end
end
```

The different properties are quite self-explanatory:

- Fs – Sample rate
- userData – Could be anything
- Next – A handle to the next filter in a chain of filters
- Prev – A handle to the previous filter in a chain of filters
- Data – The actual data
- Name – A textual representation of the filter

The above properties are hidden so that they don't show up in the Filter configuration box showned in Figure 5.

Each filter has a mandatory function called *filter()* which is responsible for filtering the incoming data. The superclass filter also defines an eventHandler function that passes received event data to the *filter()* function.

All filters have the ability to notify *'FilteringComplete'*.

When creating a custom filter, just subclass *FilterClass* and create the *filter()* function. Consider *+Filters/@LowpassFilter/LowpassFilter.m* and *+Filters/@LowpassFilter/filter.m*:

```
classdef LowpassFilter < Filters.FilterClass & handle & hgsetget
    properties
        CutOffFreq = 1200;
    end
    methods
        function obj = LowpassFilter(obj)
            obj.Name = 'Lowpass';
        end
    end
end
```

```
% Filter function for LowpassFilter
function filteredData = filter (obj, data)

    %% Do some filtering.

    obj.Data = data .* filteringMask;
    filteredData = obj.Data;
    % Notify world
    notify(obj,'FilteringComplete');
    if ~isempty(obj.Next)
        obj.Next.filter(obj.Data);
    end
end
```

The class definition of LowpassFilter defines it as a subclass of Filters.FilterClass, handle and hgsetget. It defines an additional property (that is not hidden) called CutOffFreq with an initial value of 1200, as well as setting the name of the filter upon construction.

In the *filter()* function it is important to notice that obj.Data and filteredData is updated when data is filtered, and it is crucial that the last three lines is present in every custom made filter or the filtering chain would stop.

The last thing that is needed to be able to use a custom-made filter is to add it to the available filters listbox shown in Figure 3. This is done in the file *main_GUI.m*:

```
%% Code snipped
    % Add some filters to listbox
    handles.availableFilters = cell(4,1);
    handles.availableFilters{4} = Filters.BandstopFilter;
    handles.availableFilters{3} = Filters.BandpassFilter;
    handles.availableFilters{2} = Filters.HighpassFilter;
    handles.availableFilters{1} = Filters.LowpassFilter;
    update_listbox(hObject, handles)
%% Code snipped
```