Automatically Generating VST Plugins from MATLAB Code

Charlie DeVane

MathWorks, Natick, MA, USA
charlie.devane@mathworks.com

Gabriele Bunkheila

MathWorks, Cambridge, UK
gabriele.bunkheila@mathworks.co.uk

June 4, 2016

ABSTRACT

We describe the automatic generation of VST audio plugins from MATLAB code using the Audio System Toolboxfrom MathWorks. We provide MATLAB code for three complete example plugins, discuss problems that may be encountered, and describe a workflow to generate VST plugins as quickly and easily as possible.

1 Introduction

Many audio researchers and product developers design their algorithms in MATLAB® and then for various reasons re-write their prototypes in C++ as VST plugins. Re-writing algorithms is at best tedious and error prone, even when using plugin frameworks. Short-circuiting this process by generating VST plugins directly from MATLAB code using the Audio System Toolbox™ from MathWorks® can substantially accelerate algorithm development. In this paper we provide three complete example plugins to demonstrate the fundamental programming constructs required to process audio, create plugin parameters, manage internal state, and react to parameter changes. We then discuss problems that may be encountered, and a workflow to generate plugins as quickly and easily as possible.

2 Writing A Plugin

In this section we walk through three complete example plugins. If you are accustomed to writing traditional MATLAB programs that read in all the input data, process all the data, then write out results, then this style of code may look new to you. An audio plugin only does processing, while the DAW does all the work of getting data in and out. Furthermore, in normal operation audio plugins do not have access to the entire input. They must process data one frame at a time.

Plugins use MATLAB class syntax, but you do not need to understand classes and object oriented programming to successfully write powerful audio plugins in MATLAB. Follow the patterns in these examples. They contain all you need to know to get started.

2.1 The simplest plugin: Stereo Wire

We begin with the *hello, world* of audio plugins, the stereo wire, shown in Figure 1. This is the bare minimum to create a plugin in MATLAB. It just copies its input to its output. This plugin is named `stereoWire`.

```
classdef stereoWire < audioPlugin
  methods
    function out = process(plugin, in)
      out = in;
    end
  end
end
```

Fig. 1: Stereo wire plugin

This paper was presented at the 140th Convention of the Audio Engineering Society, as Engineering Brief 238. The full published version can be found at www.aes.org/e-lib/browse.cfm?elib=18142.

Every plugin must have an audio processing function named `process`. This is the heart of the plugin. You can rename arguments `plugin`, `in`, and `out` if you wish, but the function must be named `process`. By default `process` has one stereo input and one sterero output, but other arrangements are possible. Despite its simplicity, this plugin is useful if we put a more interesting algorithm inside `process`. We can make this plugin even more useful if we add plugin parameters so we can tune the algorithm we put inside `process`.

2.2 Plugin parameters: Stereo Width

In this example, shown in Figure 2, we use simple mid-side processing to tamper with a signal's stereo image. The technique is to convert the audio from left-right format to mid-side, adjust the level of the side channel, and then convert back. Attenuating the side channel narrows the stereo image, while boosting the side channel widens the image. Thus we need a plugin parameter to control the level of the side channel.

To create the plugin parameter we add a *property* named `Width` to hold the parameter value. Object properties hold their value across function calls. Then we add a plugin interface specification to tell the plugin generator, `generateAudioPlugin`, that the generated plugin should have a parameter tied to the property `Width`. Property `PluginInterface` specifies the appearance of the plugin's dialog as it will appear in a DAW.

```
classdef stereoWidth < audioPlugin
  properties
    Width = 1
  end
  properties (Constant)
    PluginInterface = ...
      audioPluginInterface( ...
      audioPluginParameter('Width'))
  end
  methods
    function out = process(plugin, in)
      mid = (in(:,1) + in(:,2)) / 2;
      side = (in(:,1) - in(:,2)) / 2;
      side = side * plugin.Width;
      out = [mid + side   mid - side];
    end
  end
end
```

Fig. 2: Stereo width plugin

Now we can use `Width` inside `process` to control the algorithm. By default, the `Width` parameter varies linearly between 0 and 1; so this plugin can only adjust the image between mono and the original stereo. Adding a *mapping* to the parameter will improve this range:

```
audioPluginParameter('Width', ...
'Mapping', {'lin', 0, 4}))
```

This mapping enables `Width` to vary linearly between 0 and 4, so the control can adjust the image between mono and hyper-wide, with the original stereo when the control is at the 25% position. The control is more intuitive if the original stereo is at the 50% position, which we can do with a different mapping:

```
audioPluginParameter('Width', ...
'Mapping', {'pow', 2, 0, 4}))
```

Now `Width` will still vary between 0 and 4, but it will follow a square law so that `Width` is 1 at the control's midpoint. Figure 3 shows what the final stereo width plugin dialog looks like in Reaper.

You can add any number of plugin parameters in this fashion.

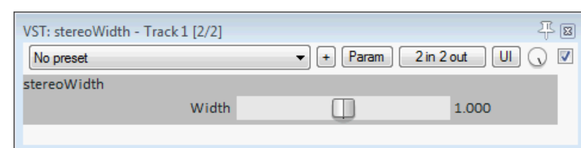

Fig. 3: Stereo width plugin dialog in Reaper

2.3 Internal state: High Pass Filter

Most algorithms require internal state to be carried over from one frame to the next. Efficiency considerations may also require internal state, such as caching results to avoid redundant computation.

To illustrate correct handling of internal state we create a high pass filter with a cutoff frequency that varies over a four octave range from 20 Hz to 320 Hz. We will use a second-order biquad filter, which gives us a slope of 12 dB/octave. The completed plugin is shown in Figure 4.

To control the cutoff frequency, we create a plugin parameter much like the stereo width control, using a

```
classdef highPass < audioPlugin
  properties
    % public interface
    Fc = 20
  end
  properties (Constant)
    PluginInterface = ...
      audioPluginInterface( ...
      audioPluginParameter('Fc', ...
      'DisplayName', 'High Pass', ...
      'Label', 'Hz', ...
      'Mapping', { 'log', 20, 320}))
  end
  properties
    % internal state
    z = zeros(2)
    b = zeros(1,3)
    a = zeros(1,3)
  end
  methods
    function out = process(p, in)
      [out,p.z] = filter(p.b, p.a, in, p.z);
    end
    function reset(p)
      % initialize internal state
      p.z = zeros(2);
      Fs = getSampleRate(p);
      [p.b, p.a] = highPassCoeffs(p.Fc, Fs);
    end
    function set.Fc(p, Fc)
      p.Fc = Fc;
      Fs = getSampleRate(p);
      [p.b, p.a] = highPassCoeffs(Fc, Fs);
    end
  end
end
```

Fig. 4: High pass filter plugin

```
% Butterworth high pass filter coefficients
function [b, a] = highPassCoeffs(Fc, Fs)
  w0 = 2*pi*Fc/Fs;
  alpha = sin(w0)/sqrt(2);
  cosw0 = cos(w0);
  norm = 1/(1+alpha);
  b = (1 + cosw0)*norm * [.5  -1  .5];
  a = [1  -2*cosw0*norm  (1 - alpha)*norm];
end
```

Fig. 5: Computing high pass filter coefficients based on [1]

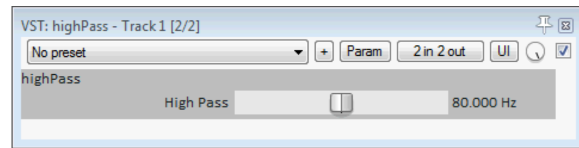

Fig. 6: High pass filter plugin dialog in Reaper

property named `Fc`. To create a more usable dialog we have added `DisplayName` and `Label` strings to the plugin interface specification and used a different mapping, `log`, which gives a more natural control for frequencies. Figure 6 shows what the high pass filter plugin dialog looks like in Reaper with the slider moved to the center.

To do the actual filtering, `process` uses the standard MATLAB `filter` function:

```
[y, z] = filter(b, a, x, z)
```

where `b` and `a` are filter coefficients, `x` and `y` are audio input and output, and `z` is the state of the internal filter delays.

We create a property `z` to carry the filter state from one `process` call to the next. Property `z` is a 2 x 2 matrix because our second-order filter requires 2 delay elements per channel.

The filter coefficients are computed by a separate function, `highPassCoeffs` (Figure 5), using equations adapted from [1]. A naive implementation might call `highPassCoeffs` directly in the `process` function, but this is unnecessary and may cause excessive CPU loading or dropouts if the computation is expensive. Instead we will cache the coefficients in two new properties `a` and `b`, and only recompute them when the cutoff frequency or sample rate change.

We have declared properties `z`, `a`, and `b` in a separate `properties` section[1], to remind ourselves that these are internal plugin states, conceptually very different from `Fc` which is part of the plugin's interface.

Internal states like properties `z`, `a`, and `b` must be initialized before processing begins, and again if the sample rate changes. This is performed by the function `reset`. In general, every plugin with internal state should have a `reset` function, and anything that depends on the sample rate should be re-computed in `reset`.

[1] Good software engineering practice recommends making this separate section private, but debugging is easier if we leave it public.

To recompute the coefficients when `Fc` changes we use the *property access* function `set.Fc`, which is called any time `Fc` is modified.

3 Workflow

In this section we discuss several kinds of errors you may encounter when developing an audio plugin in MATLAB, and a workflow designed to create a working plugin as quickly and painlessly as possible.

The kinds of errors you may encounter while creating a VST plugin include:

MATLAB programming errors These include syntax errors and runtime errors thrown by MATLAB when your code runs.

The solution to these errors is testing your code thoroughly in MATLAB. Use the many testing and debugging tools available in the MATLAB environment. Exercise all code paths and data conditions that might throw an error.

Algorithm errors Your algorithm doesn't produce the results you expect.

The solution here is also testing your code thoroughly in MATLAB. Use the many tools available in MATLAB for generating, visualizing, and analyzing data. Use the Audio Test Bench app to interactively explore your algorithm's behavior by listening to the output and viewing analyses while tuning parameters in real-time with a MIDI control surface.

Audio plugin constraint errors To work correctly in a DAW, MATLAB plugins must obey constraints imposed by the VST plugin API. For example, the plugin must produce the number of output columns it declared it would, and the number of output rows must match the number of input rows. A plugin that fails to follow these constraints may crash the DAW.

Run `validateAudioPlugin` to diagnose audio plugin constraint errors. It also checks for many code generation errors.

Code generation errors These errors are thrown by `generateAudioPlugin` while generating the VST plugin. MATLAB Coder technology, which is used to generate plugins, supports most of the MATLAB language, but some constructs that cannot be translated into efficient, embeddable code are not supported. These limitations are described in the product documentation.

There is no single solution for all of these errors. Multiple approaches are required. We recommend the following workflow:

1. Write the plugin in MATLAB, following the coding patterns in the example plugins.
2. Test the plugin MATLAB code in MATLAB for programming and algorithm errors.
3. Run `validateAudioPlugin` to find plugin constraint errors and some code generation errors.
4. Run `generateAudioPlugin` to generate the VST plugin.
5. Test the VST plugin in a DAW.

This workflow is designed to find problems as early and painlessly as possible. For example, many MATLAB programming errors can be found by running `generateAudioPlugin`, but they are often easier to detect and debug by testing in MATLAB. So the order of steps in the workflow is important.

The fastest path to success is usually to write a plugin in small increments, running through all (or at least most) of the above process at each increment. Problems are much easier to debug if you only change a small amount of code at a time.

4 Summary

Using three complete plugin examples, we demonstrated the programming constructs required to process audio, create parameters, manage internal plugin state, and react efficiently to parameter changes. We noted that by following the patterns in these examples users can create powerful plugins even if they are unfamiliar with object oriented programming. We described several kinds of problems that may be encountered, and a workflow to generate plugins as quickly and easily as possible.

References

[1] Bristow-Johnson, R., "Cookbook formulae for audio EQ biquad filter coefficients," 2004, http://www.musicdsp.org/files/Audio-EQ-Cookbook.txt.