

The Vamp Audio Analysis Plugin API: A Programmer's Guide

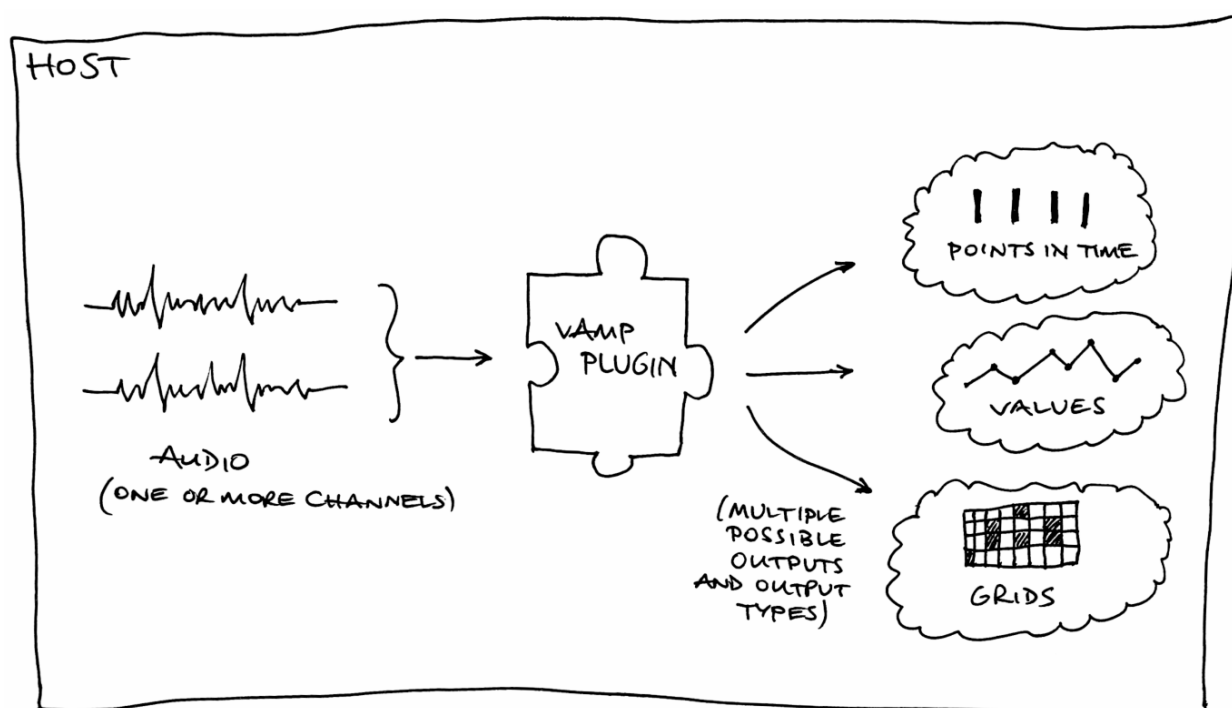
Revision 1.1, covering the Vamp plugin SDK version 1.3.

Written by Chris Cannam at the Centre for Digital Music, Queen Mary, University of London.

1. Overview

A Vamp plugin is a chunk of compiled program code that carries out analysis of a digital audio signal, returning results in some form other than an audio signal. These results are often called *audio features*.

Vamp plugins are distributed in shared library files with extension .dll, .so, or .dylib depending on the platform, with one or more plugins in each library. A plugin is usually identified to the computer using the shared library name plus a short text identifier naming the plugin within the library. The plugin cannot be used on its own, but only with a conforming “host” application which loads the plugin from its shared library and calls functions within the plugin code to configure it, supply it with data, and run it.



Plugins are not supplied with the whole of their audio input at once, but instead are fed it as a series of short blocks: in this respect they resemble real-time audio processing plugins, but in all other respects they are executed “off-line”, meaning that they are not expected to run in real-time and are not obliged to return features as they occur.

The Vamp plugin binary interface is defined in C, and the plugin SDK (software development kit) which will be described here is written in C++. Plugin authors are recommended to use the C++ interfaces in preference to plain C. This document explains the concepts and structures necessary to write Vamp plugins; it is not an API reference, which can be found at <http://www.vamp-plugins.org/code-doc/>, and it does not cover host programming.

Vamp is not an acronym. It contains letters suggestive of audio, plugins, visualisation and so on, but not in the right order.

Examples

Throughout this document we will use as examples three types of feature extractor that already exist as Vamp plugins. We will not consider the real processing work necessary to implement these techniques, only describe how they communicate with the host as plugins. These examples will appear occasionally in boxes like this one. They are:

Note onset detector. This estimates the times at which new note events begin in the signal, and returns those times as results. There are no particular values associated with the times. For the purposes of this example, our onset detector will actually have two outputs – the first will report the times as described above, and the second will report a measure of the likelihood that there is an onset within the current input block.

Chromagram. This example of a visualisation data extractor analyses audio and produces from it a grid of values, with a time step as the X coordinate and a pitch bin within the octave as the Y coordinate. Each value in the grid describes the “strength” of the corresponding pitch in the music within the given time range. Another way to describe this is to say that it returns a single column containing a fixed number of pitch strength values, from each of a series of frames or blocks of input audio.

Amplitude follower. This calculates structurally simple two-dimensional data (time and value) from the audio, to be displayed by the host as a data plot or used for some other purpose.

As these examples show, Vamp plugins do not actually do any display or user interaction: they just return data. In most cases, these data are not the final result of the work the host is doing, but are useful for something else – either for a host to display and the user to interact with in some way, or as intermediate calculations for a particular purpose such as semantically meaningful audio editing. Vamp plugins do not do swirly visualisation and are typically not useful in real-time contexts such as interacting with musical performers.

2. What does a Plugin Contain?

A Vamp plugin needs to make a certain amount of information available to the host.

Every plugin, no matter how simple, should provide the following:

1. The **identifier**, **name**, and **description** of the plugin itself.
See *Identifiers, Names and Descriptions* below.
2. The name of the **maker** of the plugin, and the plugin's **copyright** status and **version** number.
3. The **input domain** which the plugin would like its audio provided in.
See *Inputs* below.
4. The plugin's preferred **step size** and **block size** for audio input.
See *Step and Block Size* below.
5. The minimum and maximum number of **input audio channels** the plugin is capable of handling.
See *Inputs* below for discussion of these.
6. A list of **output descriptors** that contain information about the structure of the results that the plugin may produce.
See *Outputs* below.
7. Implementations of standard functions that **set up**, **reset**, and **run** the plugin.
See *Inputs* below for discussion of these.

Some plugins have parameters that can be set to adjust the way they do their processing. These plugins will also need to provide:

8. A list of **parameter descriptors** that contain information about the editable parameters of the plugin. The host may use these descriptors to show the user a control window for the plugin, for example.
See *Parameters* below.
9. Implementations of standard functions that **retrieve** and **set** the **values of parameters**.

A plugin may also have a set of pre-defined configurations that can be offered to the user by name. These are known as *programs* and a plugin that supports them will also need to provide:

10. A list of **program names**.
See *Programs* below.
11. Implementations of standard functions that **retrieve** the **current program name** and **select a new program**. The C++ base class provides virtual methods to override for these.

Base classes

The Vamp SDK contains one class from which plugin implementation classes should be derived. This class, `Plugin`, exposes pure virtual methods for most of the accessor and action functions that a plugin class needs to implement. Those that are not directly defined in `Plugin` are themselves inherited from a further class called `PluginBase`, which contains virtual methods for things that are not specific to the output structures used in Vamp – plugin name, maker, parameters, program names, etc. These classes, like everything in the SDK, are found in the Vamp namespace.

The `Plugin` and `PluginBase` classes also contain a number of data classes that are used when returning bundles of information about features (`Feature`, `FeatureList`, `FeatureSet`), outputs (`OutputDescriptor`), parameters (`ParameterDescriptor`) and so on. These will be referred to in the appropriate sections of this document.

3. Identifiers, Names and Descriptions

Vamp uses a combination of “identifier”, “name” and “description” strings to describe several sorts of object. Most obviously, the plugin itself must implement `getIdentifier`, `getName` and `getDescription` methods (inheriting from pure virtual methods in `PluginBase`) that return textual information about the plugin. Similar data are included as public data members in the `ParameterDescriptor` and `OutputDescriptor` classes.

In all of these cases, the purposes of the three strings are:

- **identifier** – This should contain a short string that the host can use to refer to the object, within the immediate surrounding scope. That is, the plugin identifier needs to be unique within the plugin's library; an output descriptor's identifier needs to be unique among output descriptors for the plugin; similarly for parameter descriptors. Identifiers are very limited in the characters they may include: upper and lower case ASCII alphabetical characters, digits 0 to 9, and the hyphen (minus sign, “-”) and underscore (“_”) characters only.
- **name** – This is a text that may be shown to the user by the host as the normal label for the object.
- **description** – This is optional, and may contain extra text to describe the purpose of the object in a way that adds to the information in the name. Hosts that show the description to the user will normally do so in addition to the name, so it should not duplicate information already in the name.

Categories

The Vamp API itself does not provide any way for the host to categorise plugins by type or purpose. However, the host extension classes in the SDK do include a method to load plugin categories from category files (with `.cat` extension) that may be found in the Vamp plugin load path alongside the plugin libraries. These are text files which contain lines of the form

```
vamp:libraryname:pluginidentifier::category
```

The category string is a series of category names separated by “>”, which describe a possibly multi-level path into a category tree. For example,

```
vamp:vamp-example-plugins:percussiononsets::Time > Onsets
```

4. Inputs

The input to a Vamp plugin is audio data, with one or more channels. The audio is non-interleaved, so the plugin receives a set of pointers to data, one per channel. The plugin can specify how many channels it will accept using its `getMinChannelCount` and `getMaxChannelCount` methods.

The number of channels, as well as the *block size* and *step size* that will be used when running the plugin, are fixed when the plugin's `initialise` method is called.

```
bool initialise(size_t inputChannels, size_t stepSize, size_t blockSize);
```

If the plugin finds the values supplied to `initialise` unacceptable, it should return `false` to indicate that initialisation has failed.

After initialisation, to supply audio data and run the plugin, the host calls the plugin's `process` method repeatedly. The process method receives a set of input pointers, and a timestamp.

```
FeatureSet process(const float *const *inputBuffers, RealTime timestamp);
```

Each time `process` is called, it is passed a single block of audio of size in samples equal to the block size that was passed to `initialise`. The difference in sample count between the input to one process call and that to the next is equal to the step size.

As with channel count, the plugin can influence the step and block size by returning its desired values through its `getPreferredStepSize` and `getPreferredBlockSize` methods; unlike channel count, the preferred step and block size are only hints, so you should always check the actual values used in `initialise` if they are important to your code. You don't have to specify a preference for these if you don't want to: return zero for the host to use its defaults, and see *Default Step and Block Sizes* below.

The audio may be provided in either *time domain* or *frequency domain* form. Time domain audio input is conventional PCM sampled digital audio with a floating-point sample type; frequency domain input is the result of applying a windowed short-time Fourier transform to each input block. The input domain is specified by the plugin using its `getInputDomain` method.

Examples

Note onset detector. The input domain and preferred step and block sizes are likely to depend on the method used for onset detection. The example plugin in the SDK, which is also annotated in the Appendix of this guide, requires frequency-domain input and can in theory handle any input step or block size. In practice it declares a preference for block size and expects the host to set the step size to something sensible accordingly.

Chromagram. The constant Q transform used for a chromagram needs, as input, the result of a short-time Fourier transform whose size depends on the sample rate, Q factor, and minimum output frequency of the constant Q transform. The chromagram plugin can therefore ask for a frequency-domain input, and make its preferred block size depend on the sample rate it was constructed with and on its bins-per-octave parameter. (See also *What Can Depend on a Parameter?* below.) It can not accept a different block size, and its `initialise` function will fail if provided with one. It may reasonably choose to leave the preferred step size unspecified.

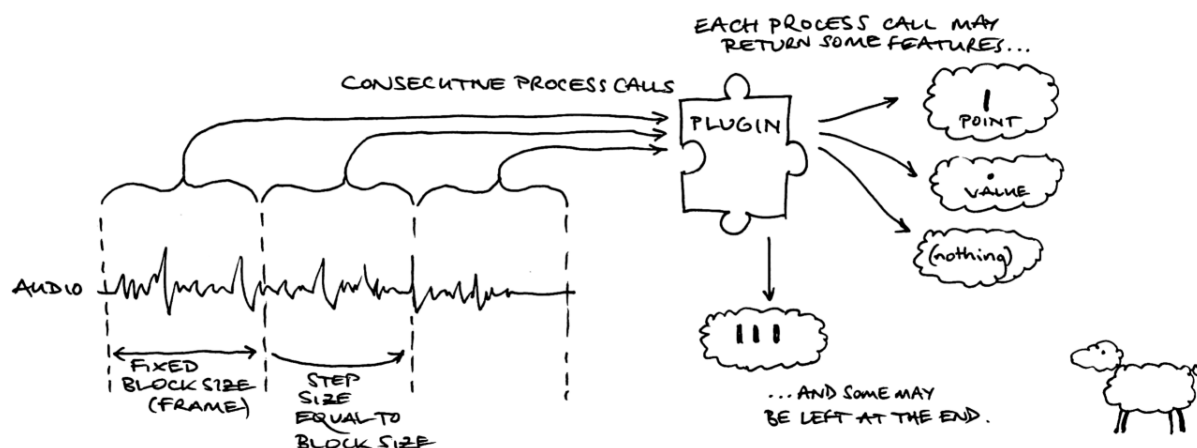
Amplitude follower. This time-domain method is likely to work with any input step and block size, and so will probably leave them unspecified.

Time Domain Input

When a plugin requests time domain input, the host divides the audio input stream up into a series of blocks of equal size, and feeds one to each successive call to `process`. The process call may then return features derived from that audio input block, according to its whim. The `inputBuffers` argument to `process` will point to one array of floats for each input channel. For example, `inputBuffers[0][blockSize-1]` will be the last audio sample in the current block for the first input channel.

When all of the audio data has been supplied, the host calls `getRemainingFeatures`, and the plugin

returns any features that are now known and not yet been returned from earlier process calls.

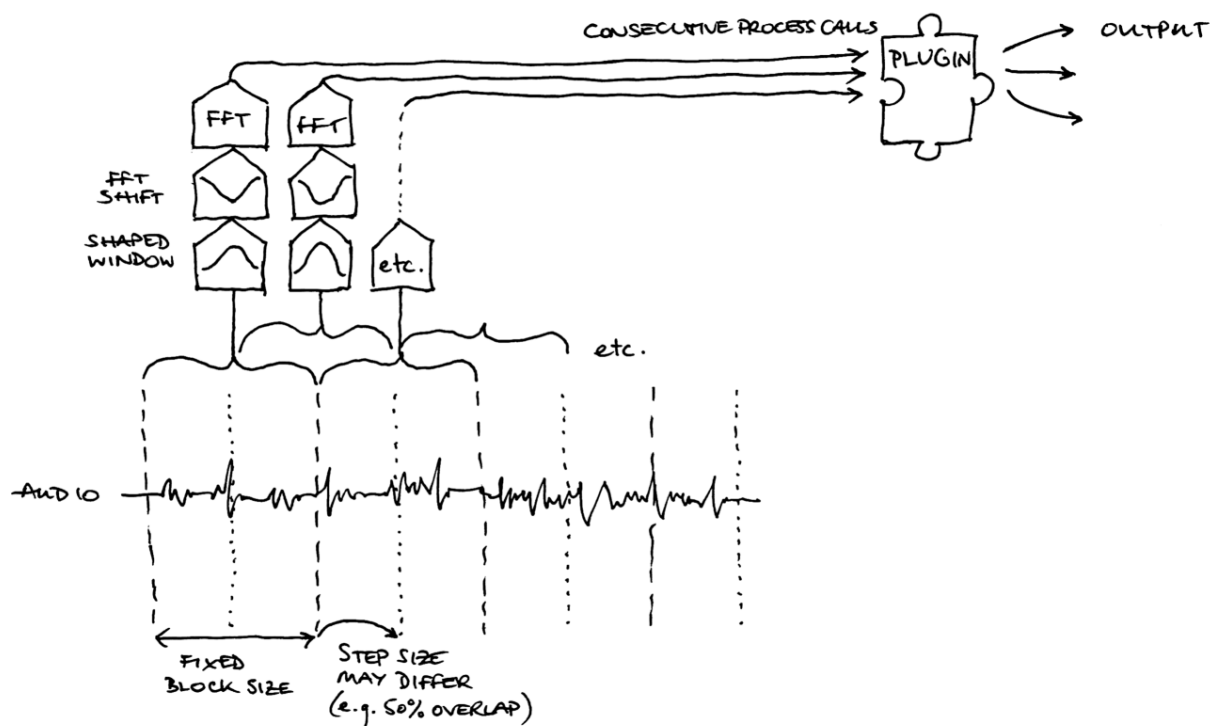


When supplying time domain input, it is most usual for the step size to be equal to the block size as shown above. This means that the plugin is receiving every sample in the audio input exactly once, in a series of contiguous blocks of data.

This does not have to be the case – the plugin can return a different value for `getPreferredStepSize` to that returned from `getPreferredBlockSize` if for any reason it would prefer to receive overlapping or non-contiguous blocks.

Vamp currently makes no provision for partial input blocks. If the audio input ends in the middle of a block, the host will fill the block with zero values up to the block size.

Frequency Domain Input



If the plugin requests frequency domain input, each block of audio input is processed by the host using a windowed short time Fourier transform before being supplied to the process function.

In this situation, it is most usual for the input blocks to overlap – that is, for the step size to be half of the

block size, or less. This is because the original time-domain data needs to be shaped using a cosine or similar window before the Fourier transform is applied, in order to avoid generating spectral noise because of the discontinuities at the edges of the input blocks. This windowing is the host's responsibility, but the plugin needs to be aware that it will happen so that it can choose sensible preferred step and block sizes.

The plugin does not get any control over what shape of window is used, or any other details of the time- to frequency-domain processing apart from the step and block size. If more control is needed, you will need to ask for time domain input and carry out the processing in the plugin instead.

When receiving frequency domain input, the `inputBuffers` argument to `process` will point to one array of floats for each input channel as for time domain input, but the arrays of float have a particular layout. Each channel contains `blockSize+2` floats, which are alternately real and imaginary components of the Fourier transform's complex output bins.

For example:

- `inputBuffers[0][0]` and `inputBuffers[0][1]` contain the real and imaginary components of the DC bin for this block in the first input channel. The imaginary component for this bin should be zero.
- `inputBuffers[0][2]` and `inputBuffers[0][3]` contain the real and imaginary components of the bin with frequency $\text{sampleRate} / \text{blockSize}$ for the first input channel.
- `inputBuffers[0][blockSize]` and `inputBuffers[0][blockSize+1]` contain the real and imaginary components of the bin with “Nyquist” frequency $\text{sampleRate} / 2$. Again, this imaginary component should be zero.

Default Step and Block Sizes

If the plugin does not care about either the step or block size, it should return zero as its preference.

If the plugin returns zero as its block size preference, the host will pick a block size that is practical for its own processing purposes. For time domain inputs this could reasonably be almost anything, from relatively small (e.g. 512) to huge (e.g. the length in sample frames of the entire audio file that the host is processing). For frequency domain inputs of course the host can reasonably be expected to use some block size that is generally considered appropriate for block-by-block short-time Fourier transform processing, such as 1024 or 2048.

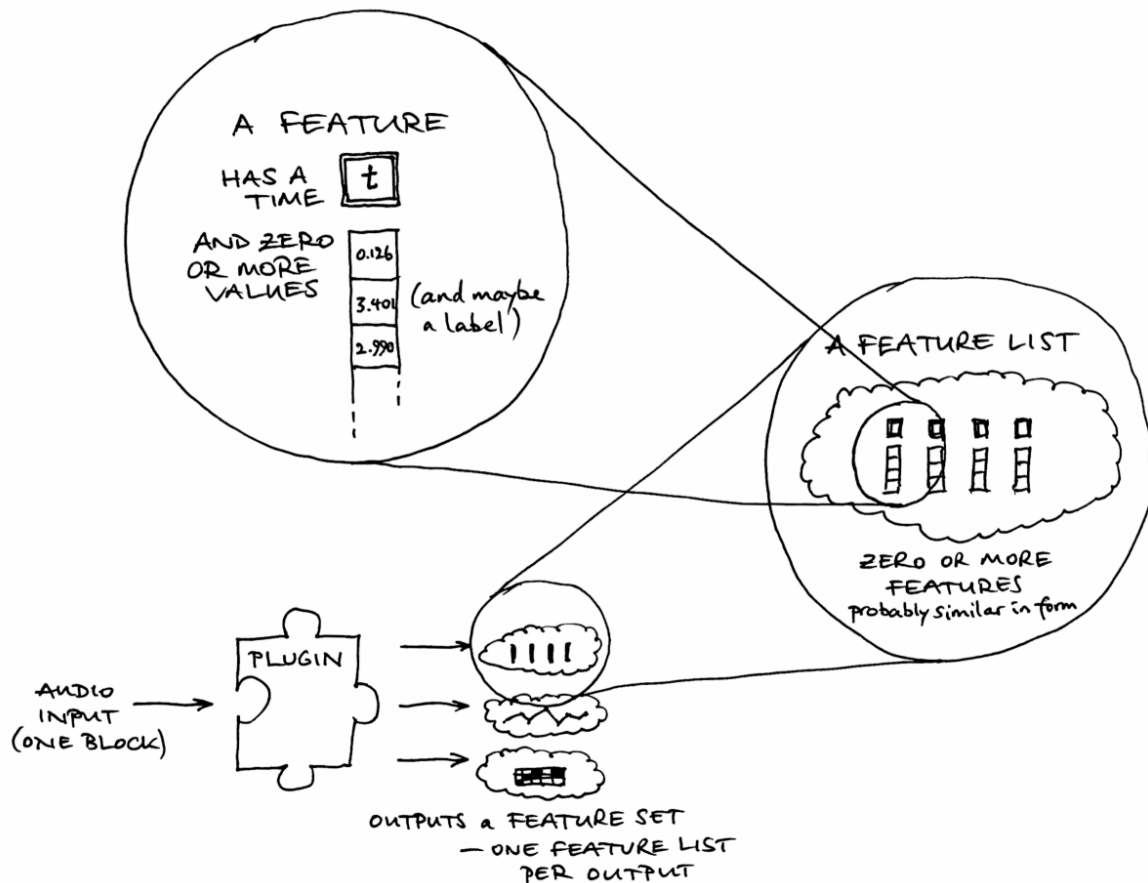
If the plugin returns zero as its step size preference, the host will typically use a step size equal to the block size for time domain inputs, or half the block size for frequency domain inputs.

In either case, the host may alternatively offer the choice of step or block size to the user.

5. Outputs

Feature Structure

A plugin may return features in two places: from the process call, and from `getRemainingFeatures`. The process call is made repeatedly to provide the plugin with input data; see “Input” above. When all of the input has been consumed, `getRemainingFeatures` is called once, and the plugin should return any other features that it has computed or can now compute but has not yet returned.



The return type from `process` and `getRemainingFeatures` is called `FeatureSet`. This is an STL map whose key is an output number and whose value is a `FeatureList`, which is an STL vector of `Feature` objects. The use of a `FeatureList` allows the plugin to return features with more than one timestamp from a single process call, or to return all features for the entire audio input in a single `FeatureSet` from `getRemainingFeatures`.

A `Feature` has an optional timestamp (see “Sample Types and Timestamps”, below), a vector of zero or more values, and an optional label. Note that even a `Feature` with zero values, no timestamp, and no label could still be a valid feature; it may indicate that “something happened in the block of audio passed to this process call”, with the interpretation of “something” depending on which output of the plugin was returning the feature.

A lot about the interpretation of returned features depends on which output the feature is associated with. A plugin has a fixed number of outputs, and it must provide a `getOutputDescriptors` method that returns data about all of them, as a vector of `OutputDescriptor` objects. The descriptor with index zero in this vector contains information about the output whose values are found with a key of zero in the feature sets returned by the plugin, and so on.

Output Descriptors

For a plugin to be of any use, it must provide at least one `OutputDescriptor` (that is, have at least one output). This contains all of the information provided by the plugin about the “meaning” of values associated with an output of the plugin. `OutputDescriptors` for all of the plugin's outputs, in order, must be returned by a call to `getOutputDescriptors`.

The `OutputDescriptor` contains:

- The identifier, name and description of this output (see *Identifiers, Names and Descriptions* above).
- Optionally, the unit (as a string) of all of the values associated with this output.
- Optionally, the number of value “bins” that features associated with this output have (via `hasFixedBinCount` and `binCount`). The bin count might be zero for “time only” features like the simple onset detector, one for “time and value” features like an amplitude tracker, or many for “column” features like a chromagram that have a series of a fixed number of values in each feature. Some features might have a variable number of values, and they will need to leave `hasFixedBinCount` false.
- Optionally, if `hasFixedBinCount` is true, names for the value bins (in `binNames`). For example, a chromagram plugin might have bin names describing the frequencies whose strengths are represented in the bin values.
- Optionally, the extents for values associated with this output (i.e. their minimum and maximum values), via `hasKnownExtents`, `minValue` and `maxValue`. Like the unit, these apply to values in all bins, if the output has more than one bin.
- Optionally, the quantization of the values associated with this output, via `isQuantized` and `quantizeStep`. A feature whose values can only fall within a certain subset of real numbers (for example, a feature whose values are always integers) may wish to set these. They are analogous to the quantization for parameters described in *Parameters* below, except that it is not possible to associate names with the quantize steps as it is for parameters.
- The sample type and rate for the output, via `sampleType` and `sampleRate`. See *Sample Types and Timestamps* below for discussion of these.

Sample Types and Timestamps

Every feature that is returned from a Vamp plugin has a time associated with it. This is the time at which the feature is considered to “start”, as a fractional number of seconds since the start of the audio input.

The time may be explicit – stored in the timestamp of the feature structure itself – or implicit – omitted from the feature itself and instead deduced by the host on the basis of the sample type and rate defined for the output in which the feature is returned. Whether the time is implicit or explicit depends on the `sampleType` field of the output and the `hasTimestamp` field of the feature.

The permitted `sampleType` values for an output are:

- **OneSamplePerStep** – Implicit time. The effective time of any feature returned by process for that output is the same as the time that was passed in to the process function by the host. The plugin should not set a timestamp on the output feature, and should set the feature's `hasTimestamp` field to false. The host should not read the timestamp from the output feature, even if the feature's `hasTimestamp` field is erroneously found to be true. Any features returned from `getRemainingFeatures` will all be effectively timed to the end of the input audio.
- **FixedSampleRate** – Implicit or explicit time. If the output feature's `hasTimestamp` field is true, the host should read and use the output feature's timestamp. The host may round the timestamp according to the sample rate given in the output descriptor's `sampleRate` field (which must be non-zero). If the feature's `hasTimestamp` field is false, its time will be implicitly calculated by incrementing the time of the previous feature according to the sample rate.
- **VariableSampleRate** – Explicit time. The time of each feature is returned in the timestamp of the feature, and the feature's `hasTimestamp` field must be set to true. The feature may have a “resolution” given by the `sampleRate` value for that output, but the host may ignore it. The

sampleRate should be zero if there is no sensible resolution.

The hasTimestamp field in the feature structure is only significant when using FixedSampleRate. If the sample type is VariableSampleRate, the host must always read and use the timestamp; if the sample type is OneSamplePerStep, the host should never read the timestamp.

The difference between FixedSampleRate and VariableSampleRate is more significant than it may at first appear. An output whose sample type is FixedSampleRate can be said to be “dense” and, for example, may be plotted naturally on a grid if it has a constant number of output bins. An output whose sample type is VariableSampleRate is in principle sparse, may need a different representation, and may be treated differently by a host – even if its features are in fact evenly spaced in time.

Examples

Note onset detector. Our onset detector as described has two outputs, one for the note onsets and the other for a function describing the likelihood that there is a note onset in a given block of audio. One thing to observe here is that the note onsets should be the first output, so that a host which defaults to using the first available output will get the results it expects, given the stated purpose of the plugin.

The note onsets output is most likely to have VariableSampleRate, with each feature timestamped explicitly with the estimated note onset position. If the plugin's detection method has a limited resolution in audio samples (for example if it can only detect whether an onset happened “during this block” but not where exactly within the block), then the plugin may also wish to report that by setting the sampleRate field of its output descriptor.

The onset detection function output could use either OneSamplePerStep or FixedSampleRate. If the detection function is always calculated directly from the input data block given to the process call, and thus results in one result for each input block, then OneSamplePerStep is the natural sample type. If the input block is subdivided within the plugin for analysis purposes, or the processing block size is otherwise different from the input block size, then it may use FixedSampleRate, giving the input sample rate divided by the processing block size as the output's sampleRate.

Chromagram. If the chromagram plugin demands frequency domain input with a block size equal to the expected FFT size for its constant Q transform, then OneSamplePerStep is the natural sampleType for the chroma data output, because one column of output data is produced for every input block.

Amplitude follower. The output sampleType may depend on the processing method, but as with the onset detector's detection function output, OneSamplePerStep or FixedSampleRate is likely.

6. Configuration

Parameters

The principal method for users and hosts to adjust the working of a Vamp plugins is via *parameters*. A parameter is a named value that may be set (and retrieved) by the host at any point between instantiation and initialisation of the plugin.

The plugin must provide a `getParameterDescriptors` method, which returns a list of `ParameterDescriptor` objects describing the available parameters of the plugin. The host refers to a parameter using the `identifier` given in its descriptor, and the plugin must provide `getParameter` and `setParameter` methods to retrieve and set the current value of the parameter.

Parameter values for Vamp plugins are always floating-point numbers. The parameter descriptor must provide limits for the permissible values through its `minValue` and `maxValue` fields. It may also provide a quantization for values, through its `isQuantized` and `quantizeStep` fields; if, for example, a parameter has `isQuantized` set to true and `quantizeStep` is set to 1.0f, then the host will only provide integer values for this parameter (or, strictly, floating point values that are the closest to integers). The descriptor may also include names to be associated with the quantize steps for a quantized parameter through its `valueNames` field: a graphical host may use these names to offer the user a list of named options instead of a numeric entry field or controller for the parameter.

Programs

Some plugins may have combinations of parameters that are known to be effective for particular sorts of tasks. For example, an onset detector may have certain parameter settings that work well for music with only soft onsets, other parameters that work well with percussive onsets, and so on. These things can be encapsulated to some degree using *programs*. A program is simply a name that a host may select or query, such that at most one program may be active at once.

A plugin that supports programs should provide a `getProgramNames` method that returns the names of all known programs, a `getCurrentProgram` method that returns the name of the currently selected program if any (or an empty string otherwise), and a `selectProgram` method that configures the plugin for the given program. The plugin is free to rewrite its own parameter values when a new program is selected; it's the host's responsibility to read the new parameter values afterwards if necessary.

What Can Depend on a Parameter?

One difference between Vamp plugins and most real-time audio processing plugin APIs is that the Vamp plugin may need to make significant aspects of its inputs and outputs dependent upon the parameters used to configure it.

For this reason, a Vamp plugin must be completely configured by the host, with its parameters and programs set, before it is initialised; the parameters cannot be changed afterwards. Furthermore, some properties of the plugin can depend on the values passed in to the `initialise` function, so the host must query these again after initialisation but before calling `process`.

A plugin may change the following at any time, up to and including during its `initialise` function:

- The `sampleType` and `sampleRate` for an output (in the `OutputDescriptor` as returned from `getOutputDescriptors`).
- The number of bin values a feature associated with an output may have (`hasFixedBinCount`, `binCount`, and `binNames` in the `OutputDescriptor`).
- The extents of values taken by features associated with an output (`hasKnownExtents`, `minValue`, `maxValue`, `isQuantized` and `quantizeStep` in the `OutputDescriptor`).
- The units of values taken by features associated with an output (`unit` in the `OutputDescriptor`).

In addition, the plugin may change its mind about the following at any point before `initialise` has been called:

- Its preferred input step size and block size (returned from `getPreferredStepSize` and `getPreferredBlockSize`).

Although a plugin may change these properties after construction, it should not do so unless it is necessary. The plugin should ensure that these properties have plausible default values right from the moment of construction, so that the host can make reasonable observations about the default or likely behaviour of the plugin without needing to initialise it. If the host does not in fact change any parameters or programs, and supplies the plugin with its preferred step and block size, then any values it has already queried from the plugin should remain valid after initialisation.

Example

Chromagram. The chromagram plugin produces a series of columns of output data, with each column containing a certain number of constant Q (pitch) bins spanning a range of an octave.

It is desirable to make the bins-per-octave value for the chromagram adjustable as a parameter. This means that the number of bin values declared for each output feature should be variable depending on the state of the plugin parameters.

Also, the processing block size for the chromagram depends upon factors including the bins-per-octave value, as well as on the input sample rate. This means that the preferred block size for the plugin should be variable depending on the sample rate given at construction time, as well as on the plugin parameters. Similarly if the plugin chooses to declare any `FixedSampleRate` outputs, that rate will also likely depend on the processing block size and therefore on the input sample rate and plugin parameters.

Finally, it is useful to offer column normalization as an option in the plugin, selectable through a parameter. For this to be possible, the extents of feature values should also be variable depending on plugin parameters.

The plugin may *not* change any of the following after construction:

- Its total number of outputs.
- The minimum and maximum number of audio input channels it accepts.
- The input domain it requires audio to be supplied in.
- The set of available parameters returned through `getParameterDescriptors`, nor anything in any of the `ParameterDescriptors` themselves.
- The set of available programs returned through `getPrograms`.
- Any other descriptive data, such as the identifier, name or description text for the plugin itself or for any of its parameters or outputs.

Appendix: An Annotated Example Plugin

In this section we take a look at an example plugin, with explanatory annotations. The plugin is essentially the “simple percussion onset detector” example plugin found in the Vamp plugin SDK, with a little reformatting in places. We choose to look at this plugin because its actual processing code is very simple, but it uses several features of the Vamp SDK including parameters and multiple outputs, and its input is in the frequency domain.

The header and implementation code for the plugin follow, and then the code for the plugin library entry point. Notes are in italics to the right of the code.

Header

```
#ifndef _PERCUSSION_ONSET_DETECTOR_PLUGIN_H_
#define _PERCUSSION_ONSET_DETECTOR_PLUGIN_H_

#include "vamp-sdk/Plugin.h"

class PercussionOnsetDetector : public Vamp::Plugin
{
public:
    PercussionOnsetDetector(float inputSampleRate);
    virtual ~PercussionOnsetDetector();

    bool initialise(size_t channels, size_t stepSize, size_t blockSize);
    void reset();

    std::string getIdentifier() const;
    std::string getName() const;
    std::string getDescription() const;
    std::string getMaker() const;
    std::string getCopyright() const;
    int getPluginVersion() const;

    size_t getMinChannelCount() const;
    size_t getMaxChannelCount() const;
    InputDomain getInputDomain() const;
    size_t getPreferredStepSize() const;
    size_t getPreferredBlockSize() const;

    ParameterList getParameterDescriptors() const;
    float getParameter(std::string id) const;
    void setParameter(std::string id, float value);

    OutputList getOutputDescriptors() const;

    FeatureSet process(const float *const *inputBuffers,
                      Vamp::RealTime timestamp);

    FeatureSet getRemainingFeatures();

protected:
    size_t m_stepSize;
    size_t m_blockSize;

    float m_threshold;
    float m_sensitivity;
    float *m_priorMagnitudes;
    float m_dfMinus1;
    float m_dfMinus2;
};

#endif
```

A plugin is implemented as a single C++ class.

All of the “basic data” methods in this block are inherited from pure virtual methods in Vamp::PluginBase.

These “input data preference” methods are inherited from virtual methods in Vamp::Plugin. Apart from the pure virtual getInputDomain, the base class implementations all return simple defaults (1 for the channel counts, 0 for the step and block size to indicate “no preference”).

The “parameters” methods are inherited from virtual methods in Vamp::PluginBase, with default implementations that declare no parameters.

This plugin does not support programs, so it can inherit the default implementations of getPrograms, getCurrentProgram, and selectProgram.

This method is pure virtual in Vamp::Plugin.

This method is pure virtual in Vamp::Plugin.

This method is pure virtual in Vamp::Plugin.

Implementation data members start here.

Plugin Implementation

```
#include "PercussionOnsetDetector.h"

using std::string;
```

```
using std::vector;
using std::cerr;
using std::endl;
```

```
#include <cmath>
```

```
PercussionOnsetDetector::PercussionOnsetDetector
    (float inputSampleRate) :
    Plugin(inputSampleRate),
    m_stepSize(0),
    m_blockSize(0),
    m_threshold(3),
    m_sensitivity(40),
    m_priorMagnitudes(0),
    m_dfMinus1(0),
    m_dfMinus2(0)
{
}
```

The sample rate at which a plugin runs is fixed at instantiation; the plugin has no control over this, and must accept any rate.

We are using 0 for step and block size to indicate “not yet set”.

Default values for our public parameters.

Basic initialisation of internal processing data.

A host needs to instantiate (construct) a plugin just to find out what outputs it provides, and other basic information. The plugin therefore needs to make its constructor as cheap to run as possible and should defer expensive setup work to initialise.

```
PercussionOnsetDetector::~PercussionOnsetDetector()
{
    delete[] m_priorMagnitudes;
}
```

Clean up processing data. delete[] is safe even if the data were never initialised and the pointer is still NULL.

```
string
PercussionOnsetDetector::getIdentifier() const
{
    return "percussiononsets";
}
```

Identifier for this plugin within its shared library.

```
string
PercussionOnsetDetector::getName() const
{
    return "Simple Percussion Onset Detector";
}
```

Name of this plugin, as far as the user is concerned.

```
string
PercussionOnsetDetector::getDescription() const
{
    return "Detect percussive note onsets by identifying broadband energy rises";
}
```

Description of what the plugin does, to accompany its name.

```
string
PercussionOnsetDetector::getMaker() const
{
    return "Vamp SDK Example Plugins";
}
```

Normally, the company, institution, or developer of the plugin.

```
string
PercussionOnsetDetector::getCopyright() const
{
    return "Code copyright 2006 Queen Mary, University of London, after Dan Barry et al 2005. Freely redistributable (BSD license)";
}
```

Concise summary of copyright and distribution license.

```
int
PercussionOnsetDetector::getPluginVersion() const
{
    return 2;
}
```

Newer versions of the plugin must have larger version numbers. The host may use this to warn the user the the plugin has been updated since they last used it, or that it may be out of date.

```
size_t
PercussionOnsetDetector::getMinChannelCount() const
{
    return 1;
}
```

Smallest number of input audio channels plugin can accept. The base class implementation of this method returns 1, so we don't strictly need to define it for this plugin.

```
size_t
PercussionOnsetDetector::getMaxChannelCount() const
{
    return 1;
}
```

Largest number of input audio channels plugin can accept. Again, the base class implementation also returns 1 – a plugin that doesn't declare otherwise always gets one channel.

```
PercussionOnsetDetector::InputDomain
PercussionOnsetDetector::getInputDomain() const
{
    return FrequencyDomain;
}
```

The form in which audio input must be provided (time or frequency domain).

```

size_t
PercussionOnsetDetector::getPreferredStepSize() const
{
    return 0;
}

size_t
PercussionOnsetDetector::getPreferredBlockSize() const
{
    return 1024;
}

bool
PercussionOnsetDetector::initialise(size_t channels, size_t stepSize, size_t blockSize)
{
    if (channels < getMinChannelCount() ||
        channels > getMaxChannelCount()) {
        return false;
    }

    m_stepSize = stepSize;
    m_blockSize = blockSize;

    m_priorMagnitudes = new float[m_blockSize/2];

    for (size_t i = 0; i < m_blockSize/2; ++i) {
        m_priorMagnitudes[i] = 0.f;
    }

    m_dfMinus1 = 0.f;
    m_dfMinus2 = 0.f;

    return true;
}

void
PercussionOnsetDetector::reset()
{
    for (size_t i = 0; i < m_blockSize/2; ++i) {
        m_priorMagnitudes[i] = 0.f;
    }

    m_dfMinus1 = 0.f;
    m_dfMinus2 = 0.f;
}

PercussionOnsetDetector::ParameterList
PercussionOnsetDetector::getParameterDescriptors() const
{
    ParameterList list;

    ParameterDescriptor d;
    d.identifier = "threshold";
    d.name = "Energy rise threshold";
    d.description = "Energy rise within a frequency bin necessary to count toward broadband total";
    d.unit = "dB";
    d.minValue = 0;
    d.maxValue = 20;
    d.defaultValue = 3;
    d.isQuantized = false;
    list.push_back(d);

    d.identifier = "sensitivity";
    d.name = "Sensitivity";
    d.description = "Sensitivity of peak detector applied to broadband detection function";
    d.unit = "%";
    d.minValue = 0;
    d.maxValue = 100;
    d.defaultValue = 40;
    d.isQuantized = false;
    list.push_back(d);

    return list;
}

float
PercussionOnsetDetector::getParameter(std::string id) const
{
    if (id == "threshold") return m_threshold;
    if (id == "sensitivity") return m_sensitivity;
    return 0.f;
}

```

This plugin works with any step size. Since the plugin demands frequency domain input, most hosts will use a step size equal to half the block size.

Preferred block size (i.e. FFT size, for frequency domain input).

This function does the real work of setting up the plugin's internal processing. If any of the values provided are unacceptable, it should return false to indicate failure.

Although we had a preferred block size, we can accept anything.

Properly initialise our processing data...

True indicates success.

After reset is called, the plugin should behave as if it has just been initialised.

Host calls this to find out what adjustable parameters the plugin has. These are returned as a ParameterList, which is an STL vector of ParameterDescriptors.

Return the current value of the parameter whose identifier is given in the id argument.

```

void
PercussionOnsetDetector::setParameter(std::string id, float value)
{
    if (id == "threshold") {
        if (value < 0) value = 0;
        if (value > 20) value = 20;
        m_threshold = value;
    } else if (id == "sensitivity") {
        if (value < 0) value = 0;
        if (value > 100) value = 100;
        m_sensitivity = value;
    }
}

PercussionOnsetDetector::OutputList
PercussionOnsetDetector::getOutputDescriptors() const
{
    OutputList list;

    OutputDescriptor d;
    d.identifier = "onsets";
    d.name = "Onsets";
    d.description = "Percussive note onset locations";
    d.unit = "";
    d.hasFixedBinCount = true;
    d.binCount = 0;
    d.hasKnownExtents = false;
    d.isQuantized = false;
    d.sampleType = OutputDescriptor::VariableSampleRate;
    d.sampleRate = m_inputSampleRate;
    list.push_back(d);

    d.identifier = "detectionfunction";
    d.name = "Detection Function";
    d.description = "Broadband energy rise detection function";
    d.binCount = 1;
    d.isQuantized = true;
    d.quantizeStep = 1.0;
    d.sampleType = OutputDescriptor::OneSamplePerStep;
    list.push_back(d);

    return list;
}

PercussionOnsetDetector::FeatureSet
PercussionOnsetDetector::process(const float *const *inputBuffers,
                                Vamp::RealTime ts)
{
    if (m_stepSize == 0) {
        cerr << "ERROR: PercussionOnsetDetector::process: "
              << "PercussionOnsetDetector has not been initialised"
              << endl;
        return FeatureSet();
    }

    int count = 0;

    for (size_t i = 1; i < m_blockSize/2; ++i) {
        float real = inputBuffers[0][i*2];
        float imag = inputBuffers[0][i*2 + 1];

        float sqrmag = real * real + imag * imag;

        if (m_priorMagnitudes[i] > 0.f) {
            float diff = 10.f * log10f(sqrmag / m_priorMagnitudes[i]);
            if (diff >= m_threshold) ++count;
        }
    }
}

```

Set the value of the parameter whose identifier is given in the id argument.

Host calls this to find out what outputs a plugin has. A single plugin may have any number of outputs. The host does not specify which outputs it is going to use; the plugin always computes and returns value from all of them.

*This output has a fixed number of values per feature ...
... and that is zero: only the time of the feature is relevant here.*

*Each feature will have its own timestamp ...
... which is accurate to 1/ the input sample rate, in seconds.
The first descriptor in the list corresponds to output 0 in the features returned from the process call below.*

*This output has one value per feature ...
... and that value is quantized ...
... to 1: that is, the output values are converted from integers.
Each process call returns a single implicitly-timed feature.
The second descriptor in the list corresponds to output 1 in the features returned from the process call.*

The function that does the bulk of the work. This function receives non-interleaved floating-point audio data through inputBuffers and the timestamp of the first audio frame for this process call in ts. The input audio has the same number of channels as reported to the initialise function earlier: this number will be within the range the plugin returns from its getMinChannelCount and getMaxChannelCount methods. In the case of this plugin, that means it will always have exactly one channel.

This part is our actual algorithm; see the referenced paper for details. Essentially, because we asked for frequency-domain input through getInputDomain earlier, the input to process will be a series of m_blockSize/2 + 1 real+imaginary floating point number pairs containing DFT bin values for the input block. We count the number of bins whose power exceeds that of the corresponding bin in the previous process block by more than m_threshold dB, and if this number exceeds a quantity calculated from the sensitivity value, we declare an onset to have been detected. This loop does the counting.


```

        m_priorMagnitudes[i] = sqrmag;
    }

    FeatureSet returnFeatures;

    Feature detectionFunction;
    detectionFunction.hasTimestamp = false;
    detectionFunction.values.push_back(count);
    returnFeatures[1].push_back(detectionFunction);

    if (m_dfMinus2 < m_dfMinus1 &&
        m_dfMinus1 >= count &&
        m_dfMinus1 >
            ((100 - m_sensitivity) * m_blockSize) / 200) {

        Feature onset;
        onset.hasTimestamp = true;
        onset.timestamp = ts - Vamp::RealTime::frame2RealTime
            (m_stepSize, lrintf(m_inputSampleRate));
        returnFeatures[0].push_back(onset);
    }

    m_dfMinus2 = m_dfMinus1;
    m_dfMinus1 = count;

    return returnFeatures;
}

PercussionOnsetDetector::FeatureSet
PercussionOnsetDetector::getRemainingFeatures()
{
    return FeatureSet();
}

```

Save the bin power for comparison in the following call.

The FeatureSet maps from output number to list of features.

Feature for onset detection probability function. Its time is implicit, based on the timestamp of the input block, as the corresponding OutputDescriptor says OneSamplePerStep. This is the only feature returned for output 1.

Now use the detection function to guess whether an onset is happening. We actually examine the prior input block, and call it an onset if that block's detection function value was both greater than a certain quantity, and greater than the values appearing before and after it.

This feature is explicitly timed, to correspond to the prior block. Time is the input timestamp for this block, minus the step size. If this feature exists, it is the only feature returned for output 0. So if no onset detected, this feature will not be returned at all.

Save previous detection function values for comparison.

Return any features that could not be returned from process calls, after all data has been received. Some plugins may do all of their real work from this function. This one has nothing to do here; all results were returned from process.

Plugin library entry point

This is how the examples/plugins.cpp file from the Vamp plugin SDK might appear, if the library in question contained only one plugin (the percussion onsets one just described). There is not much scope for creativity in this file – cut-and-paste is the recommended way to write it!

```

#include "vamp/vamp.h"
#include "vamp-sdk/PluginAdapter.h"
#include "PercussionOnsetDetector.h"

static Vamp::PluginAdapter<PercussionOnsetDetector>
    percussionOnsetAdapter;

const VampPluginDescriptor *vampGetPluginDescriptor(unsigned int version,
                                                    unsigned int index)
{
    if (version < 1) return 0;

    switch (index) {
    case 0:
        return percussionOnsetAdapter.getDescriptor();
    default:
        return 0;
    }
}

```

The PluginAdapter class takes a C++ plugin implementation and provides a descriptor structure for it that conforms with the official Vamp plugin API, which uses C linkage. You should never need to know anything about how it works!

This function should fail if the library was built using a version of the Vamp API that is both incompatible with and more recent than the version number provided to this function. Since the only existing Vamp API at the moment is version 1, the correct incantation is as shown here.

This function is called by the host repeatedly with increasing index values starting at zero, until it returns a null result, in order to discover how many plugins are in the library and to retrieve their descriptors. We only have one plugin, so we return its descriptor for index 0 and null otherwise.