## Guide to BOOK Chapter 9 Examples:
*Victor Lazzarini*

# Programming the Phase Vocoder

These examples will build on most systems, and have been tested on OS X, Linux and Windows. To build the examples you will first need a C compiler. I recommend using **gcc**, if you have a choice (on OS X and Linux you'll get it by default). On Windows **gcc** can be got by either installing **MinGW/MSYS** or **Cygwin** (www.cygwin.com).

Examples are built using **scons**, which is a handy utility for code maintenance and programming projects. It looks after re-building any files that have been modified and gets all the right components together to build the whole project. See the 'scons-guide' document for more details. If you don't have **scons,** you can get it from www.scons.org, and you will also need Python, which you can get from www.python.org. Both software are of easy installation.

You will also need to install **libsndfile**, if you do not have it already. In fact, **scons** will look for it and tell you if you don't have it. Then you can get it from www.mega-nerd.com/libsndfile. It's all very simple to build (if you need to) and install, all the instructions for your system are provided.

Once your system is ready for it, all you need to do is type the following at your shell/terminal (`$` is the prompt) and all examples will be built.

```
$ scons
```

Because the example programs for this chapter require several source files, building them from the command line directly with **gcc** is very tricky and is not recommended.

### pv

This example demonstrates the use of the Phase Vocoder and its inverse operations and how they are transparent (ie. the inverse operation recuperates the original signal). It also serves as basis for some of the exercises/projects in this chapter, as the processing operations can be applied for the spectral data that is generated in between the two transforms. The main function is found in *pv_main.cpp* and the *pva()* and *pvs()* operations in *pv.cpp*. If you want to use these functions in your own projects, you will need to add that file to your project build (in fact, all further examples for this chapter will also include it).

### morph

This program implements a morphing operation. Input 1 is seamlessly morphed into input 2 throughout its duration. **Project 1** proposes some ideas of how to add more flexibility to this program.

### ifd

This example demonstrates the use of an alternative method of analysis, the instantaneous frequency distribution. To reconstruct the signal, we use an additive synthesis method. See **Project 2** for enhancements to this program.

## Projects

Here I'll propose some ideas for you to try. These will be mainly based on enhancing the examples that have been provided for this chapter. However, they'll give you some useful insight into audio programming.

### Project 1

The morphing program in the examples lacks flexibility, as it morphs from file 1 to file 2 using the duration of the shortest file. It would be nice to allow the user to define the morph trajectory. You could then:

1. Allow the user to set a breakpoint file to control the morphing.
2. Provide independent control of morphing of amplitudes and frequencies.

The breakpoint files could be just plain text files with durations and morph values, one for each parameter.

### Project 2

The **ifd** example program has a number of untapped opportunities: it allows for control of frequency and amplitude in the additive synthesis stage. Also, a change of synthesis hopsize could timescale the input (stretching or compressing it). So you could allow the user to independently:

1. Transpose the input without changing its duration.
2. Change the input duration without changing its frequency.

The latter has just a minor complication where if the output is timestretched, then more memory for the output buffer will need to be allocated (otherwise you will run out of memory and crash). Envelopes and flexible user control could be added to the program to allow for these features to be easily manipulated.