

Writing Python Bindings with SWIG

Audio Programming and Signal Processing 4

Sarah Clark

1 Introduction

Python and C++ are two very different programming languages. Python is very versatile- it's good for testing and its very easy to use. However, relatively, it's slow and therefore for time dependant applications its not a great choice. C++ on the other hand takes more time to develop and understand, but it's much more efficient as C code compiles to the native machine code of the processor rather than being interpreted by the Python interpreter. Therefore it's implementation is much faster. Ideally, we'd like to use the best parts of each language.

Python is written in C, and therefore we can 'extend' it by writing new code in C/C++. This is what the 'Simplified Wrapper and Interface Generator' (SWIG) does. SWIG is a piece of open-source software which creates 'wrappers' for C/C++ programmes in a number of different languages, such as python which we are using in this project. It allows us to use and implement C++ modules within a python program, so that faster code can be written.

2 Building C++ files

2.1 File Types

To build a C++ programme you generally need several different files. In this project we use the following:

Source Code Files: These files contain the main code needed for the programme to work. They often end in something such like '.cc', '.cpp', or '.cxx'.

Header Files: These files contain class and function declarations and preprocessor statements (e.g. #include, #define, etc.). This allows the source code to access external classes and functions. These files end in '.h'.

Object Files: These files are produced when the header and source files are compiled. They are binary outputs containing the function definitions but can't be executed on their own. They generally end in '.o'[2].

Shared Object Files: These files are created by compiling and linking object files to create a 'library'. This can be used by other programmes to access the source code functions[3].

2.2 Compiling C++ Programmes

In order to link the source and header files together to create objects and libraries, the code needs to be compiled. This can be done from the command line using a C++ compiler such as `g++` or `gcc`, e.g:

```
g++ -fPIC -c simple_hello.cxx
```

Here, `g++` is the compiler, `-fPIC` tells the computer to create an output file of 'Position Independent Code', i.e. an object file, and `-c` tells us to compile the named file- here 'simple_hello.cxx'.

Another example would be:

```
g++ -shared simple_hello.o simple_hello_wrap.o -o _simple_hello.so
```

Here, `-shared` tells the computer to link the named object files, here 'simple_hello.o' and 'simple_hello_wrap.o', and create a shared object. The command `-o` tells us to create the new shared file with the name specified, here '_simple_hello.so'.

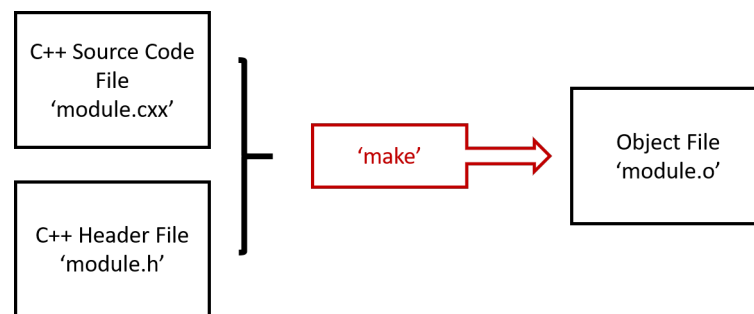


Figure 1: Compiling C++ source and header code files

2.3 MakeFiles

For longer C++ programmes with many files, it can be quite a lengthy process to compile everything from the command line. Therefore there are several different ways of building these projects, for an example a 'Makefile' which we are going to consider for this project.

To write a Makefile, you have to write a set of instructions to tell the computer how to compile the files you have, but telling it what you want to make, what files are needed to make this, and how to make this. E.g:

```
example: example.cxx examplefunc.cxx
g++ -o example example.cxx example func.cxx -I
```

Here the output is 'example' and the colon tells us that it depends on the files 'example.cxx' and 'examplefunc.cxx'. The next line needs to start with a tab (needed for Makefile syntax) and tells

us how to compile these files[4].

For each line in the Makefile we would need to carefully consider the sources and dependencies for each output file. To make this easier to edit, we can group all the source files, e.g: `SRCS := example.cxx examplefunc.cxx`. Going further we can create other constants and flags, such as the compiler to use (`CC := g++`), the build command (`CXXFLAGS += -fPIC`), the list of flags to pass to the compilation command (`CFLAGS := -I`) and dependencies (`DEPS := example.h`).

We can also introduce rules such as:

```
%.o: %.cxx $(DEPS)
      $(CC) -c -o $@ $(CFLAGS)
```

The `%o` tells us that this rule apply to all object files, and that it depends on `%.cxx` files and all the `.h` files that are listed in the `'DEPS'` constant. The second lines tells us to use the compiler defined in `'CC'`, generate the object file (`-c`) and output this as the type of file named to the left of the colon (`$@`), i.e. an object file, using the flags specified in `'CFLAGS'`.

Objects, libraries and dependencies can also be built in their own directory so that the source directory is kept uncluttered, e.g.:

Object directory: `ODIR := obj`

Library directory: `LIBDIR := ../lib`

Include directory: `IDIR := ../include`

The command `'@echo'` can be used which prints out something to indicate the progress of the `'make'` command so we can see what is happening.

Another important thing to note about Makefiles is that when the command `'make'` is issued, only files that have had their dependencies edited since the last `'make'` command are remade, so not everything is remade everytime.

Generating Prerequisites Automatically:

One of the rules that needs to be written for an object file is the dependencies, e.g. to say which header files an object file depends on:

```
example.o: example.h
```

For a large programme this can get very confusing, and so compiler flags can be used to tell the compiler to find these dependencies by looking at the `'#include'` lines in the source files. This information is generated into a `'.d'` file which are then included by the Makefile so `'make'` is aware of that information. These flags might include:

- M** : Creates makefile dependencies rules based on the source filename
- MT** : Renames the target of the makefile dependency rule (can only be used in conjunction with on these other rules: -M, -MM, -MD, -MMD)
- MD** : Generates dependency information during compilation. Lists both system header files and user header files.
- MMD** : Generates dependency information during compilation. Lists only user header files.
- MP** : Adds a target for each prerequisite in the generated list
- MF** : Specifies the filename for the makefile dependency rules (can only be used in conjunction with on these other rules: -M, -MM, -MD, -MMD)

3 Using SWIG

To import a C++ programme into python using SWIG, you need to write a SWIG interface file to produce a SWIG wrapper file.

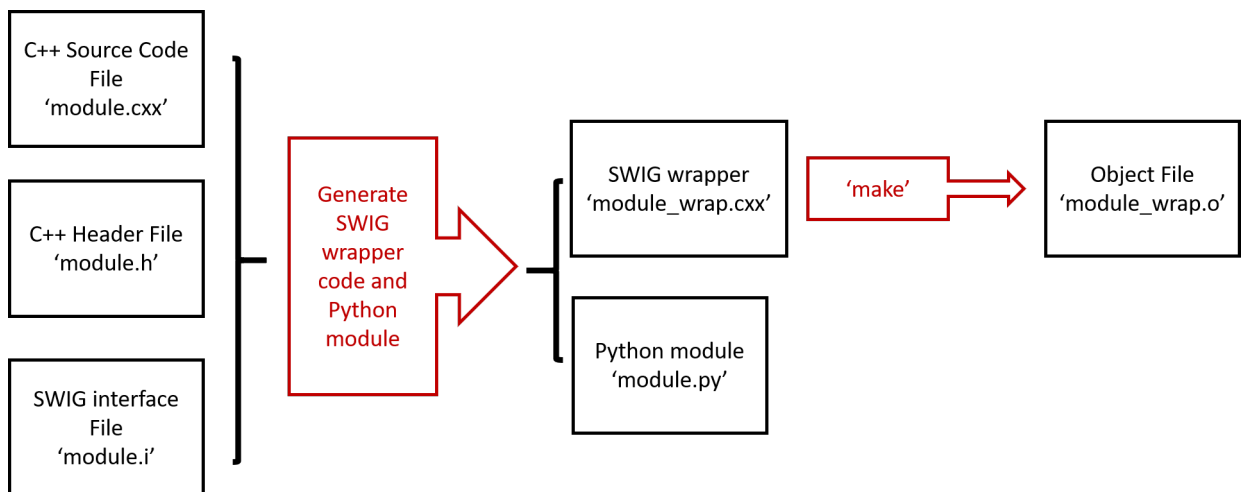


Figure 2: Building a SWIG wrapper for a C++ programme

3.1 Writing a SWIG interface file

To create a SWIG wrapper for a C/C++ programme we first need to write a interface specification file which explains to SWIG what the C++ code does. These files normally end '.i'.

For example, an interface file for a module named 'example' might include:

Module name: `%module example`

C++ standard libraries: `%include <std_string.i>, %include <std_complex.i>`

Code that inserts a macro to specify that the C++ file should be built as a python extension inserting the module init code:

```
#define SWIG_FILE_WITH_INIT
#include example.h>
```

A line to include the C++ header file so that the code can be parsed and output:

```
%include example.h
```

3.2 NumPy Arrays with SWIG

'NumPy' is a Python library that adds many functions for arrays and matrixes. For many audio programming applications, the NumPy library is very helpful, but as it is a Python module it is not available in C++. In many cases, you may want to input or output a NumPy array to or from a SWIG wrapped C/C++ function. Luckily software has been written so that this is possible. In order to use NumPy arrays with SWIG we need to add a few more lines to the interface files:

Line to include the Numpy interface file to access NumPy functions:

```
%include numpy.i
```

Import array:

```
%init %{
import_array();
%}
```

Line to set up array input/outputs, e.g.

```
%apply (double* IN_ARRAY1, int DIM1) {(double* array, int n)};
```

This line tells SWIG that a one dimensional array will be input to the programme under the argument 'double* array', and it's size will be the value of the argument 'int n'. Instead of 'IN_ARRAY' we could also use 'INPLACE_ARRAY' (arrays that are modified in place), 'ARGOUT_ARRAY' (output arrays), and several other more complicated options which are mentioned in the 'numpy.i' documentation[6]. For arrays with more dimensions, the number simply increases, e.g. 'IN_ARRAY2' would be a two dimensional array. Both dimensions would then have to be specified as input 'int' arguments.

3.3 Generating a SWIG wrapper

A SWIG wrapper for a C++ file can also be generated from the terminal, e.g:

```
swig -c++ -python simple_hello.i
```

Here, we tell 'SWIG' to create a Python language wrapper for the programmes and actions described by the interface file, here 'simple_hello.i'. SWIG defaults to C processing, but by including -c++ we have told it to enable C++ processing instead. This line will create a file called 'simple_hello_wrap.cxx' which should be compiled along with the source code files to create objects, and then shared objects. This also creates a python file, here 'simple_hello.py'

which, when the shared object library has been created, can be imported to python just like any other python module, as shown in Figure 1 below which uses the example 'simple_hello' code to call a function to print 'Hello World':

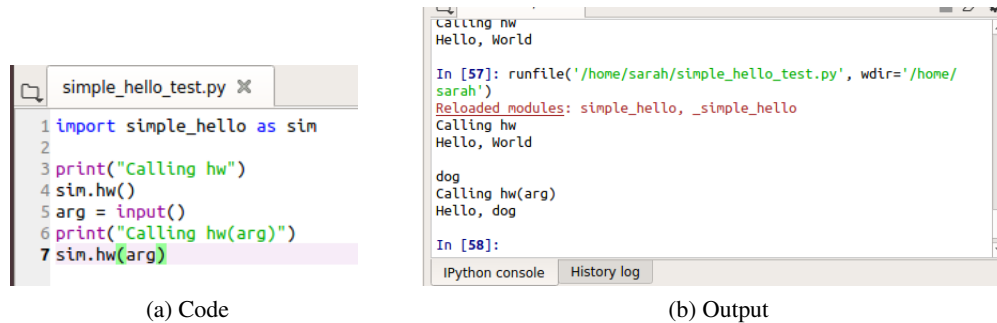


Figure 3: Example Python Code

Of course, the line to generate a SWIG wrapper can also be included in the Makefile.

3.4 Common Errors

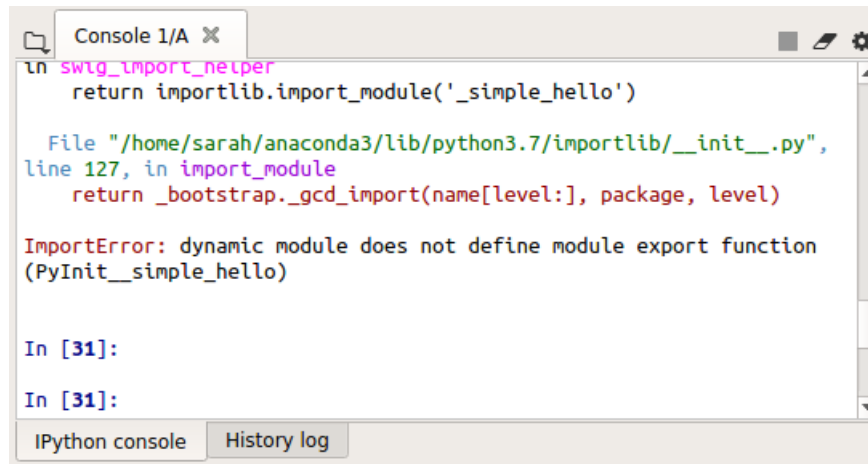
Most SWIG errors are due to build issues. Some of the most common SWIG issues were experienced here as I was learning how to use the software.

- 1) The error shown below usually means that either the wrapper code wasn't compiled in the module, or that the module hasn't been given the right name, e.g. the module (shared object file) doesn't begin with an underscore[1].



Figure 4: Error 1

- 2) This error, again, is often caused by not giving the right name to the object file. It can also happen due to linking errors[1]. Here the error was solved when the files were moved to the home directory so I assume it was a problem with the file path.



```
Console 1/A X
In swig_import_helper
    return importlib.import_module('_simple_hello')

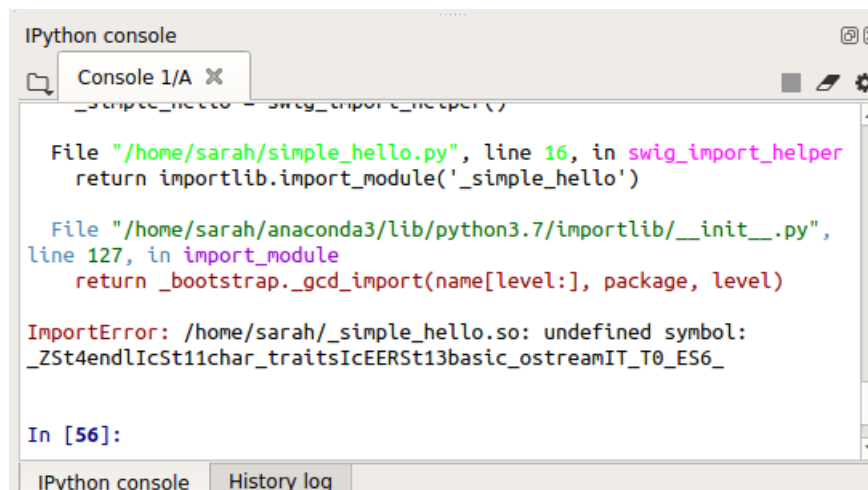
File "/home/sarah/anaconda3/lib/python3.7/importlib/__init__.py",
line 127, in import_module
    return _bootstrap.gcd_import(name[level:], package, level)

ImportError: dynamic module does not define module export function
(PyInit_simple_hello)

In [31]:
In [31]:
IPython console History log
```

Figure 5: Error 2

- 3) This error usually means that a file has been forgotten when linking the shared library file[1]. Here this error occurred because the files were moved to a different directory. As soon as the object files were rebuilt, the error disappeared.



```
IPython console
Console 1/A X
..._simple_hello = swig_import_helper()

File "/home/sarah/simple_hello.py", line 16, in swig_import_helper
    return importlib.import_module('_simple_hello')

File "/home/sarah/anaconda3/lib/python3.7/importlib/__init__.py",
line 127, in import_module
    return _bootstrap.gcd_import(name[level:], package, level)

ImportError: /home/sarah/_simple_hello.so: undefined symbol:
_ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_0_ES6_

In [56]:
IPython console History log
```

Figure 6: Error 3

4 Example for Tone Generation

To use SWIG to create a wrapper for a C++ program to use in python, we have to do the following steps:

- 1) Write a programme in C++ that you want to implement in python
- 2) Write a SWIG interface file and generate the wrapper code
- 3) Compile and link the C++ programme with the wrapper code to create a shared library
- 4) Import the shared object file (shared library) in python as a python module and use the C++ functions in the python programme

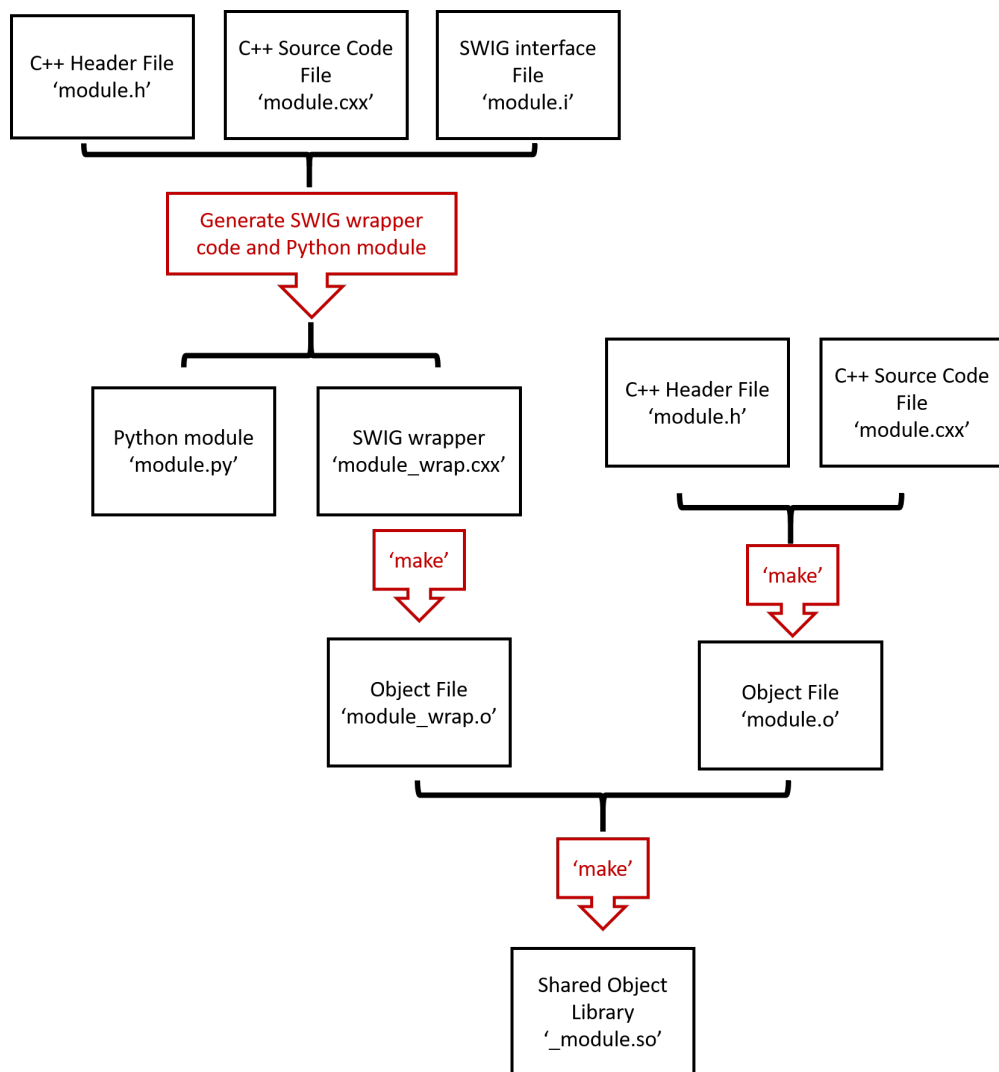


Figure 7: Using SWIG to import a C++ function into python

4.1 C++ Programme

For this assignment, I wrote a C++ function, based off a python programme written earlier in this course, to generate a sine wave. The function takes in the number of samples you wish to generate (the length of the array to return) and a value for frequency in Hz. The function will then return an array (to python) of the sine wave values generated. The arguments were: 'double *arr', 'int fs', 'float f'.

The sine wave was calculated using the formula:

$$\text{Array value} = \text{Amplitude} * \sin(2 * \pi * \text{frequency} * \frac{\text{sample no.}}{\text{sampling frequency}})$$

This produced the C++ function:

```
void wave (double *arr, int fs, float f)
{
    int i = 0;
    float b = float(fs);
    for (int i = 0; i < fs; i++)
    {
        float c = i/b;
        arr[i] = 100*sin(2*PI*f*c);
    }
}
```

The C++ files for this code are included as 'generator.cxx' and 'generator.h'.

While testing this code the main the issue I encountered was a problem with integer division. As the sine wave values were based on the decimal incrementation of *sample number/sampling frequency*, both of these values had to be floats, otherwise the array output was always zero. As the sampling frequency value was input as the size of the array, it had to be an int. Therefore I had to cast this value to float before the function code work.

4.2 SWIG Interface File

An interface file, 'generator.i', was created for this C++ function:

```
%module generator

#include <std_string.i>
#include <math.i>

%
#define SWIG_FILE_WITH_INIT
#include "generator.h"
%}

#include <numpy.i>
```

```
%init %{
import_array();
%

%apply (double* ARGOUT_ARRAY1,int DIM1) {(double *arr, int fs)};
#include "generator.h"
```

The argument 'double *arr' was assigned to be a 'double* ARGOUT_ARRAY1' (a one dimensional output array), and the argument 'int fs' was set as the length of that array 'int DIM1'. For this project a test file was not made, as due to the swig syntax, the C++ test file didn't work exactly the same as the SWIG generated python module, i.e. the C++ code saw the output array as an input argument, and did not output anything.

4.3 Compile and Link

The files were compiled and linked using a Makefile:

```
#####
#Variables to change depending on programmes:
#Module name
MODULE := generator

#Library Sources
LIBSRCS := generator.cxx

#Output file extension
LIBEXT := so

#Compiler flags
CXX = g++
CXXFLAGS += -fPIC
LDFLAGS += -shared
SWIG += swig
SWIGFLAGS = -c++ -python

#File paths for the include files provided by Python and NumPy
PYTHON_INCLUDES := $(python3-config --includes)
NUMPY_INCLUDES := -I/usr/lib/python3/dist-packages/numpy/core/include

#####

#Set up
#Specify link flags for Python extension
PYTHON_LIBS := $(python3-config --libs)
```

#Object Directory

OBJDIR := obj

#Object files for shared library and test programme

LIBOBJS := \$(LIBSRCS:%.cxx=\$(OBJDIR/%.o))

#Swig Wrapper

WRAPPER := \$(MODULE)_wrap.cxx

WRAPOBJ := \$(OBJDIR)/\$(MODULE)_wrap.o

#Sources

SRCS := \$(LIBSRCS) \$(WRAPPER)

#Output files from build

OUTPUT := _\$(MODULE).\$(LIBEXT)

PYTHON_MODULE := \$(MODULE).py

#####

#Set-up auto-dependency generation

#Place dependency files into a subdirectory of object directory named .dep

DEPDIR := \$(OBJDIR)/.deps

SWIGDEPS := \$(DEPDIR)/\$(MODULE).swigdeps.d

#Compiler flags to convince the compiler to generate the dependency file

DEPFLAGS = -MT \$@ -MMD -MP -MF \$(DEPDIR)/\$(F).d

SWIGDEPFLAGS := -MM

#Object file compile rule

COMPILE.cxx = \$(CXX) \$(DEPFLAGS) \$(CXXFLAGS) -c

#####

#Compile

#Name files to 'make'

all:: \$(OUTPUT) \$(PYTHON_MODULE)

#Create SWIG wrapper

#The &: separator means that both the C++ wrapper and Python module are generated.

#DEPDIR is an order prerequisite (specified by '|'), meaning it will get built before the targets.

\$(WRAPPER) \$(PYTHON_MODULE) &: \$(MODULE).i | \$(SWIGDEPS) \$(DEPDIR)

```

@echo Target:  $@
@#echo Stem:   $*
@echo Unsatisfied Prerequisites:  $?
$(SWIG) -c++ $(SWIGDEPFLAGS) $< > $(SWIGDEPS)
$(SWIG) $(SWIGFLAGS) $<

#Create shared library
$(OUTPUT): $(WRAPOBJ) $(LIBOBSJS)
@echo Target:  $@
@#echo Stem:   $*
@echo Unsatisfied Prerequisites:  $?
$(CXX) $(LDFLAGS) $(WRAPOBJ) $(LIBOBSJS) -o $@ $(PYTHON_LIBS)

#Create swig wrapper object
$(WRAPOBJ): $(WRAPPER) $(DEPDIR)/$(WRAPPER:%.cxx=%.d) | $(DEPDIR)
@echo Wrapper.
@echo Target:  $@
@#echo Stem:   $*
@echo Unsatisfied Prerequisites:  $?
$(COMPILE.cxx) $(PYTHON_INCLUDES) $(NUMPY_INCLUDES) -o $@ $<

#####

#'%:%.o:%.cxx' = This commands is to delete the built-in rules for
building object files from .cxx files, so that this rule is used
instead
#'$$(DEPDIR)/%.d' = This declares the generated dependency file as
a prerequisite of the target, so that if it's missing the target
will be rebuilt
#'|$(DEPDIR)' = This declares the dependency directory as an order
only prerequisite of the target, so that it will be created when
needed.
$(OBJDIR)/%.o : %.cxx $(DEPDIR)/%.d | $(DEPDIR)
@echo A C++ file
@echo Target:  $@
@#echo Stem:   $*
@echo Unsatisfied Prerequisites:  $?
$(COMPILE.cxx) -o $@ $<

#Declare a rule for creating the dependency directory if it doesn't
exist
$(DEPDIR): ; @mkdir -p $@

#Generate a list of all the dependency files that could exist

```

```
DEPFILES := $(SRCS:%.cxx=$(DEPDIR)/%.d) $(SWIGDEPS)
```

#Mention each dependency file as a target, so that 'make' won't fail if the file doesn't exist

```
$(DEPFILES):
```

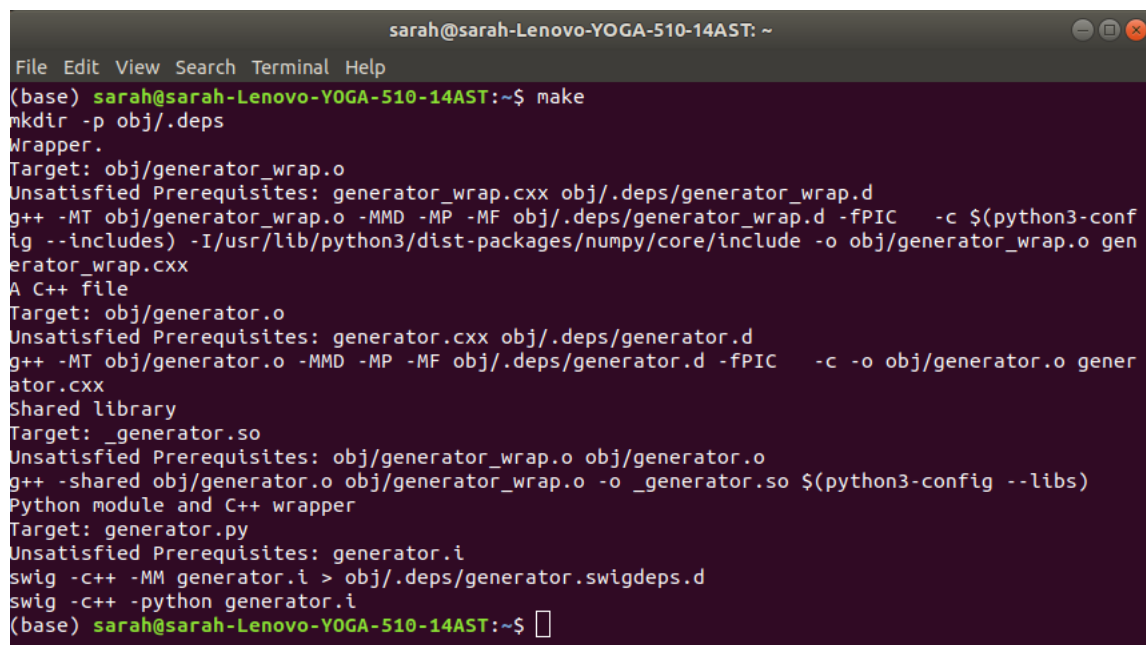
```
clean::
```

```
@printf "Removing Object and Depend files... n"
@if [ "x`realpath $(OBJDIR)`x" = "x`realpath .`x" ] ; then
    printf " n*** Don't set OBJDIR to . n n" ;
    $(RM) $(LIBOBS) $(TESTOBS) ;
    $(RM) -r $(DEPDIR) ;
else
    $(RM) -r $(OBJDIR) ;
fi
$(RM) $(SONAME) $(WRAPPER) $(PYTHON_MODULE) $(TESTPRG)
```

#Include the dependency files that exist. Use 'wildcard' to avoid failing on non-existent files

```
include $(wildcard $(DEPFILES))
```

The SWIG wrapper 'generator_wrap.cxx' was produced, along with object file 'generator_wrap.o', and the shared library '_generator.so'.



```
sarah@sarah-Lenovo-YOGA-510-14AST: ~
File Edit View Search Terminal Help
(base) sarah@sarah-Lenovo-YOGA-510-14AST:~$ make
mkdir -p obj/.deps
Wrapper.
Target: obj/generator_wrap.o
Unsatisfied Prerequisites: generator.cxx obj/.deps/generator_wrap.d
g++ -MT obj/generator_wrap.o -MMD -MP -MF obj/.deps/generator_wrap.d -fPIC -c $(python3-config --includes) -I/usr/lib/python3/dist-packages/numpy/core/include -o obj/generator_wrap.o generator_wrap.cxx
A C++ file
Target: obj/generator.o
Unsatisfied Prerequisites: generator.cxx obj/.deps/generator.d
g++ -MT obj/generator.o -MMD -MP -MF obj/.deps/generator.d -fPIC -c -o obj/generator.o generator.cxx
Shared library
Target: _generator.so
Unsatisfied Prerequisites: obj/generator_wrap.o obj/generator.o
g++ -shared obj/generator.o obj/generator_wrap.o -o _generator.so $(python3-config --libs)
Python module and C++ wrapper
Target: generator.py
Unsatisfied Prerequisites: generator.i
swig -c++ -MM generator.i > obj/.deps/generator.swigdeps.d
swig -c++ -python generator.i
(base) sarah@sarah-Lenovo-YOGA-510-14AST:~$
```

Figure 8: Build output from make command

4.4 Import into Python

The '`_generator.so`' shared object file was imported into python like a module in the '`generator-test.py`' code:

```
import generator as g
import scipy.io.wavfile as wavfile
import numpy as np
import matplotlib.pyplot as plt

fs = 220500
f = 1000

print("Calling wave")
a = g.wave(fs,f)

#plot waveform
time = np.linspace(0, len(a)/fs, len(a))
plt.figure(1)
plt.plot(time,a)
plt.xlabel("Time (s)")
plt.ylabel("Amplitude")
plt.title("Waveform Generated")

#create new wav file
x = np.array(a, dtype = 'int16')
name = str(f)+"Hz.wav"
wavfile.write(name, fs, a)
```

The output array produced by the C++ function was examined for several values of frequency, and graphs were plotted to confirm that a sine wave was generated. These are shown below. Wav files were also created from these arrays which are included with the files in this project.

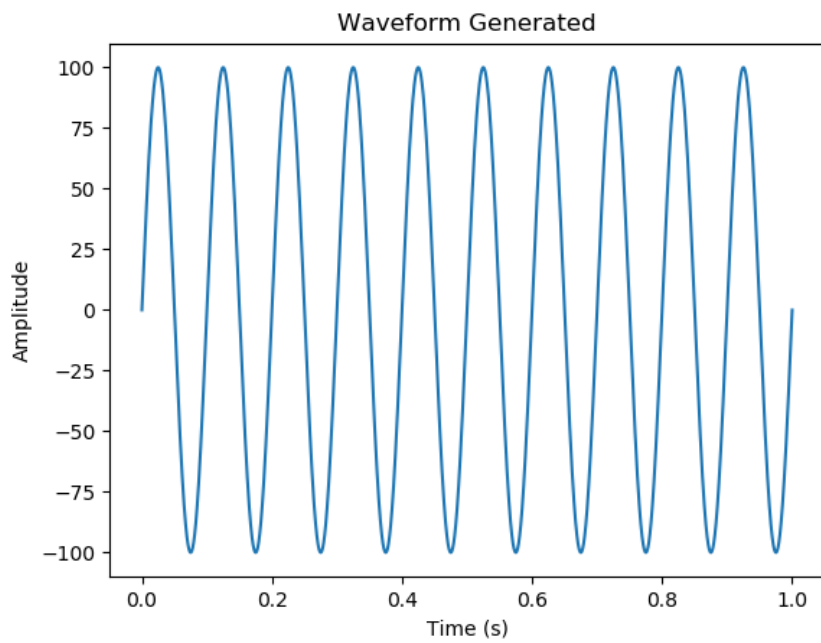


Figure 9: Python output waveform with frequency = 10Hz

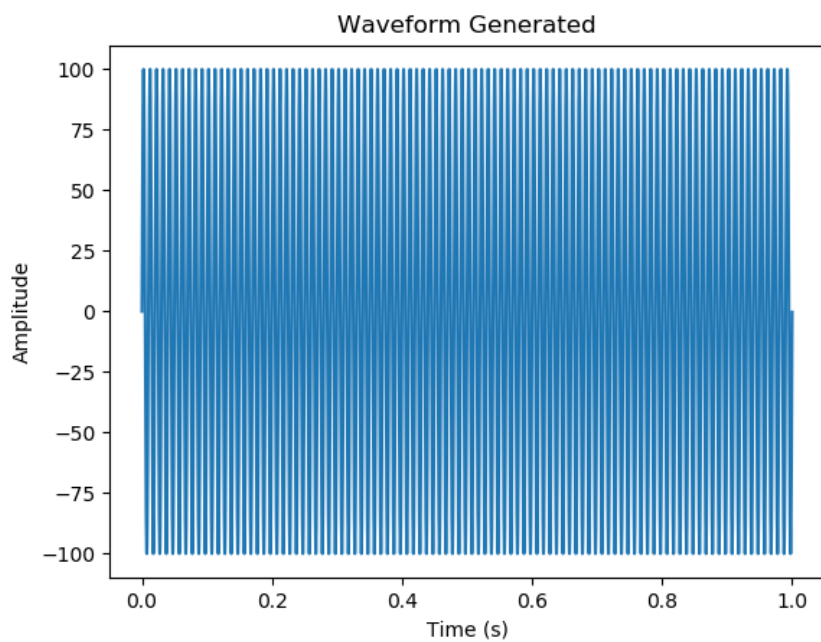


Figure 10: Python output waveform with frequency = 100Hz

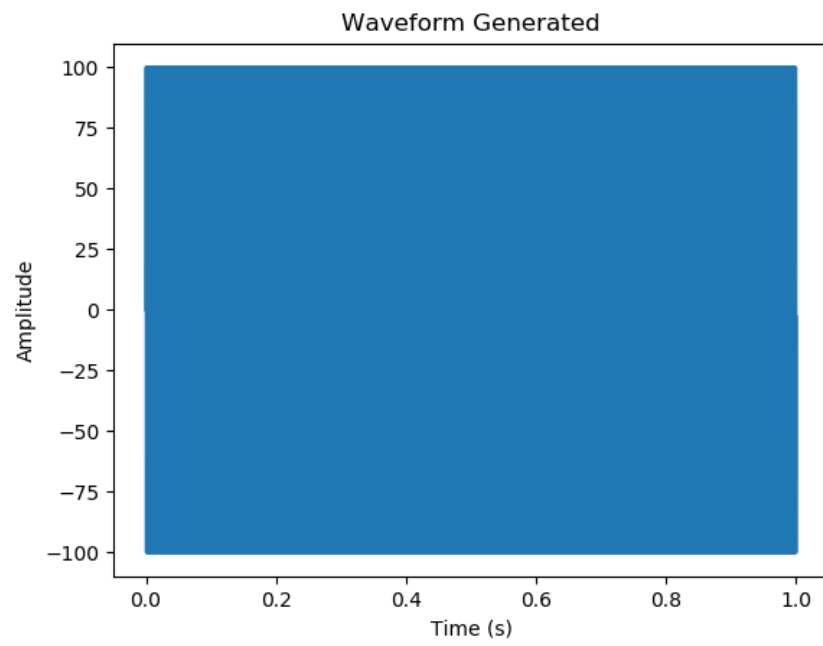


Figure 11: Python output waveform with frequency = 1000Hz

References

- [1] SWIG Documentation. *SWIG and Python*. Available online: http://www.swig.org/Doc3.0/Python.html#Python_nn3. (Accessed April 2020)
- [2] Caltech Computing and Mathematical Sciences. *C++ track: compiling C++ programs*. Available online: http://courses.cms.caltech.edu/cs11/material/cpp/mike/misc/compiling_c++.html. (Accessed April 2020)
- [3] YoLinux. *Static, Shared Dynamic and Loadable Linux Libraries*. Available online: <http://www.yolinux.com/TUTORIALS/LibraryArchives-StaticAndDynamic.html>. (Accessed April 2020)
- [4] Colby Computer Science. *A Simple Makefile Tutorial*. Available online: <http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/>. (Accessed April 2020)
- [5] GNU make (2019). *Auto-Dependency Generation*. Available online: <http://make.mad-scientist.net/papers/advanced-auto-dependency-generation/>. (Accessed April 2020)
- [6] SciPy.org. *numpy.i: a SWIG Interface File for NumPy*. Available online: <https://docs.scipy.org/doc/numpy/reference/swig.interface-file.html>. (Accessed April 2020)