Software-Defined Radio
Lab 3: Frequency shift compensation

Version 0.1

Anton Blad

May 16, 2011

1 Introduction

In lab 1, you wrote a very basic system for transmission and reception of an analog signal. The source signal is frequency shifted to an intermediate frequency by the transmitter, but since the used LF daughterboards do not contain any mixers the resulting signal still has an I and a Q component and requires two separate cables. On the receiver side, the signal is frequency shifted down to baseband again. However, in this process, a small frequency offset is introduced. The offset normally comes from the different clocks used for TX and RX, but in the single-USRP setup the TX and RX clocks are the same and the offset stems from differences in the implementations of the frequency shifts in the AD9862 chip (for TX) and the FPGA (for RX).

The source audio signal is purely real (no imaginary part), but due to the frequency mismatch the downshifted received signal will have both a real and an imaginary part. If this data is plotted in the complex plane it will show as a rotating line. In lab 1, you reconstructed the audio signal without consideration of the frequency shift by simply taking the real part of the received signal, and the frequency shift could then be heard as a slow modulation of the amplitude of the signal, as the power alternated between the real and the imaginary part. In this lab, you will solve the problem by doing a manual frequency and phase compensation of the received signal. While this could be done using a number of standard GNU Radio building blocks, you will instead do it by writing your own frequency compensation block in C++.

2 Hardware requirements

The following hardware is needed for this lab:

- A computer with USB 2.0
- A USRP with an LFTX/LFRX daughterboard pair at the A side

2.1 Initialization

If you are doing this lab in CommSys' research lab:
Log in to one of the lab computers as the user *SDR lab user*. Your lab teacher will give you

the password. Your lab computer is prepared with all needed files in the directory

```
/home/sdrlabuser/labfolder/lab3
```

Open a terminal and cd to that directory. You do not need to worry about making changes to files there. That directory will be updated for the next occation.

If you are not following these instructions in CommSys' research lab:
Open a terminal. Create a directory `labfolder` somewhere and cd into it. Download and extract the accompanying lab files by executing the following commands:

```
$ wget http://www.commsys.isy.liu.se/SDR/labs/gr/lab3.tar.gz
$ tar zxf lab3.tar.gz
$ cd lab3
$ wget http://www.commsys.isy.liu.se/SDR/labs/gr/test.wav
```

All files needed in this lab are now in the directory `lab3` that you just created. Note: Compiling C++ blocks for GNU Radio has a rather extensive list of prerequisite libraries and tools. Refer to the build instructions for GNU Radio 3.3.0 and make sure that all needed libraries are present before continuing.

3 Build system overview

This section contains a brief description of the configuration files and tools used to build processing blocks for GNU Radio. If this is your first encounter with the `autotools` suite, you will likely feel overwhelmed with the complexity of it. However, most of the configuration is done for you already.

3.1 A short history on UNIX build systems

UNIX systems have for a long time used the `make` tool to automate compiling and linking programs and libraries. `make` uses files called `Makefile`, which contains general rules describing how to invoke tools for compiling source files and linking object files. These rules depend on a large number of factors, including the architecture of the system, the used compiler, the location of libraries, and more. Also, different flavors of UNIX have different function calls, requiring the code to be different depending on the system it is compiled for. After a while, developers got bored of updating the `Makefile`s and adapting their code for every system, and thus `autoconf` was born. `autoconf` uses a little-known language called M4 to generate a shell script called `configure`. The M4 scripts define different kinds of system tests to perform, that are then included in the `configure` shell script. `configure` in turn performs the system tests which result in a set of variables describing library versions, compiler and linker options and so on. `configure` then creates a system-specific `Makefile` from a template `Makefile.in`, and also creates a file `config.h` that can be included by the source files to selectively determine code that is suitable for the target system. Thus the burden of maintaining a project was reduced to writing platform-independent `Makefile.in` files, and writing system tests in the M4 script language.

With time, the `Makefile.in` files grew in size, and rules were added for dependency generation, cleaning of the build directory, redistribution of the source code, building of technical documentation, installing, and so on. Maintaining these files also became boring, and thus `automake` was written to automate it. `automake` uses a file `Makefile.am` that contains a number of "magic" variable names describing target executables and libraries, and the source files that are needed to build them. `automake` then generates all the standard build, install and clean targets in a `Makefile.in` file.

In this project, `libtool` is also used to automate building, configuring and installation of shared libraries. However, this is mostly transparent for the developer.

3.2 The directory tree structure

The use of both Python and C++ in GNU Radio complicates things a bit more. The source tree has the following structure:

- `.` is the top level of the source tree.
- `config/` contains the M4 scripts used by `autoconf` and `automake`.
- `lib/` contains the C++ code of the processing blocks.
- `python/` contains the Python code in the package. In this lab, there is a unit test for the block, and applications for transmitting and receiving test signals.
- `swig/` contains the SWIG interface files that defines the connections between the C++ code and the Python code.

The lab is based on the official GNU Radio processing block tutorial (`gr-howto-write-a-block`). However, the official tutorial has a catch-22, where SWIG rules are built from a template makefile by make itself. However, the generated makefile is needed by `automake` to build the `Makefile` that defines the rules to build the generated makefile, making the code impossible to build from a clean state. In this tree, the problem has been solved by moving the SWIG rule generation to a separate shell script which is run before `automake`.

4 Configuring the build system

In this lab, most of the build system is configured for you already, and your part will consist of defining the name of the software package in all relevant places, and adding the set of source files where required.

4.1 Edit build rules

A brief description of all the files and their purposes is included here. Look through them and do the requested changes. You need to change all occurences of `<package>` to `sdrlab`, which will be the package name through which the block is exposed to Python. The C++ code is contained in two files: the header file `lib/sdrlab_freqcomp_cc.h` and the source code file `lib/sdrlab_freqcomp_cc.cc`.

- `AUTHORS` – contains the list of authors of the module
- `configure.ac` – this file contains the configuration for `autoconf`
- `Makefile.common`, `Makefile.am`, `lib/Makefile.am`, `python/Makefile.am`, `swig/Makefile.am` – these files are the input to `automake`

- `Makefile.swig`, `Makefile.swig.gen.t` – these files contain SWIG invocation rules (because `automake` has no native SWIG support), no changes are required in these files
- `python/__init__.py` – this file contains Python package initialization, it imports the compiled SWIG wrapper code to make the C++ block available in Python.
- `swig/sdrlab.i` – this file defines the C++ classes and functions that are exposed to Python

The build rules are set up so that when the compiled files are installed, they will be accessible through a Python package `sdrlab`. However, it is preferable to be able to run test code without having to do a system installation. Therefore, we will cheat the Python interpreter that the `python/` directory contains the `sdrlab` package. We do this by creating a symbolic link in the `python/` directory. Execute the following command in the top directory for lab 3:

```
$ ln -s . python/sdrlab
```

Then, when the Python interpreter is run in the `python/` directory, it will be able to find the `sdrlab` package in the same directory.

4.2 Testing the build system

The lab includes skeletons for the C++ source code in the `lib` directory, and a unit test in the file `python/qa_sdrlab_freqcomp_cc.py`. Thus, at this point, you should be able to compile the project and perform the unit test. However, the unit test will fail because the C++ source code does not actually do anything useful yet.

Execute the following commands in the top directory for lab 3 to do the build:

```
$ ./bootstrap
$ ./configure
$ make
$ make check
```

Ask your lab instructor in case you are unsure whether the commands succeed or not.

5 Writing the signal processing code

When the build system is successfully configured, you are ready to start writing the code. The class that implements the processing block is defined in the C++ header file `lib/sdrlab_freqcomp_cc.h`. The file requires no changes, but you should look through it to get an idea of the behavior of the class.

The definitions of the functions are in the C++ source file `lib/sdrlab_freqcomp_cc.cc`. You will need to implement three functions:

- `set_frequency`: This function sets the parameter for the frequency correction.
- `add_phase`: This function adds an offset to the currently accumulated phase.

- `work`: This function does the actual signal processing. The implementation should read complex samples from the `input_items` argument, multiply it by the complex exponential of the accumulated phase and write the results to the `output_items` argument. For each sample, the accumulated phase must also be updated. Be sure to get the correct scaling of the phase. At some point you will need to multiply by π, which is available through the constant `M_PI`. Also make sure to renormalize the accumulated phase around zero, or you will get accuracy problems when the accumulated phase grows too big. You may also look in the unit test file to get an idea of the expected behavior of the code.

When you have implemented the code, recompile and run the unit test by executing the command

```
$ make check
```

which will reconfigure and recompile as needed. If all parts of the unit test do not succeed, go back and correct the problem before continuing.

6 Using the frequency correction block

There are three applications in the `python/` directory. `usrp_tx.py` and `wav_tx.py` are the same as in lab 1, and use the USRP to transmit a sinus signal and the `test.wav` sound file, respectively. `wav_rx.py` is the receiver from lab 1, extended with an instance of the `sdrlab.freqcomp_cc` block, and controls to set the frequency and add a phase offset. It will also show the frequency corrected samples in the complex plane, which will show as a slowly rotating line. Run one of the transmitters and the receiver using the command

```
$ ./pyrun <file.py>
```

in two terminals. Fiddle with the frequency and phase controls and notice how they change the plotted samples and the sound from the speakers.

7 Resources

- C++ reference documentation: `http://www.cplusplus.com`