Assignment 1: Fixed-point FFT
Signal Processing Technologies

21/03/2007

Instructor

Prof. R. Knopp
Institut Eurécom
knopp@eurecom.fr

1 Introduction

This assigment has the goal of understanding the limitations of fixed-point processing on an example of a common signal processing algorithm, in this case a normalized Fast Fourier Transform. Fixed-point FFTs find common application in digital communication systems for OFDM transmission and reception, and often as well for channel estimation and equalization in single-carrier systems.
The FFT is an algorithm for implementing the *discrete-Fourier Transform (DFT)* which transforms and discrete-time input waveform, $x_n, n = 0, 1, \cdots N$ into an output waveform $X_k, k = 0, 1, \cdots, N$ according to the relationship

$$X_k = \frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} x_k W_N^{kn} \tag{1}$$

where $W_N = e^{-\frac{j2\pi}{N}}$. This is written more compactly as $\mathbf{X} = \mathrm{DFT}(\mathbf{x})$. Note that this version of the DFT is energy conserving. The inverse operation (IDFT) is given by the relation

$$x_k = \frac{1}{\sqrt{N}} \sum_{k=0}^{N} X_n W_N^{-kn} \tag{2}$$

or more compactly as $\mathbf{x} = \mathrm{IDFT}(\mathbf{X})$.

2 Getting Started

You will create a fixed-point performance testbench of the above algorithm in non-optimized C (at least fill in what has not been done for you already). The starting point is in the directory `work`. The goal of the testbench is to assess the performance, in terms of distortion, of different parametrization options for the fixed-point implementation as a function of the type of signal we are trying to process. This is discussed later.
There are several files in the `work` directory:

1. `Makefile`: This is a GNU makefile for generating the testbench. You shouldn't have to modify it

2. `complex.h`: This is a header file containing complex number structures. You shouldn't have to modify it.

3. `fixed_point.c`: This contains the fixed-point emulation routines for the configurations considered in this exercise. You should not have to modify it, but you do have to look at it to answer some of the questions below.

4. `fft.c` : This is the main file you have to work with. It contains the different FFT routines and the testbench routines. You do have to modify it.

5. `taus.c` : This is a uniform random number generator. You don't have to modify it.

6. `gauss.c` : This is a floating-point Gaussian random number gnerator. You don't have to modify it.

2.1 Fixed-point arithmetic

The arithmetic in your implementation will consider two examples. First you will use 16-bit fixed-point multiplication and 16-bit saturated addition, which would be a common format for some DSP processors. Second you will assume that multiplication are 25x18 which is the configuration for DSP units in current generation Xilinx FPGAs.

2.2 The FFT routine

The classical implementation of the DFT is the Fast-Fourier-Transform (FFT), which is perhaps one of the most famous algorithms in applied signal processing. Implementing the FFT first requires a table of coefficients for the W_N^{-n}, typically one period of a sinewave (or at least one-quarter period of a sinewave if space is an issue). In our case, this sinewave must be quantized to 4096 points, since this is the largest FFT size, N, we will require. We will assume that the FFT will only be done on signals whose length is a power of two. This rules out complex "split-radix" implementations which do not involve powers of two or four. The most efficient implementation uses a combination of a radix-4 FFT and one final radix-2 stage if N is not a power of four. If you would like some information on radix-2/radix-4 FFT implementations see

1. `http://en.wikipedia.org/wiki/Fast_Fourier_transform`

2. http://www.gweep.net/ rocko/FFT
3. http://www.cmlab.csie.ntu.edu.tw/cml/dsp/training/coding/transform/fft.html
4. http://momonga.t.u-tokyo.ac.jp/%7Eooura/index.html.

The second reference has example code which is a radix-2 FFT *in floating-point arithmetic*. It nevertheless gives you an idea of the algorithm. The third reference (see fft4g_h.c) has a nice radix-4/radix-2 implementation, again in floating-point. We have taken the second example as a basis for the study on distortion in the fixed-point implementation since the distortion of a floating-point optimization is neglible.
The main issue which arises in a fixed-point implementation is the scaling procedure. To guarantee that you don't overflow the computations proper re-scaling must be performed between each stage of the FFT. This can be parametrized, in the sense that you choose the number of bits to shift at the input of each FFT stage and at the final output. We know that in our configuration (energy conserving), we have to shift by $.5 \log_2(N)$ bits in total since we scale by $\sqrt{N}$. The performance of the algorithm depends heavily on the relationship between the rescaling policy and the statistics of the input signal. Therefore, it is non-trivial to choose the scaling policy. Typically it depends on the type of application and we will explicitly treat this issue once the implementation is complete.
The twiddle factor lookup tables for the coefficients are generated for you assuming that they are full-scale.

2.3 Distortion tests

We would like to characterize the distortion as a function of the signal amplitude and type of signal. We can compute the empirical distortion as

$$D = 20 * \log_{10} \left(\frac{1}{N} \sum_{n=0}^{N-1} |X_{n,\text{fixed}} - X_{n,\text{float}}|^2 \right) \tag{3}$$

Three signal types are of interest. The basic test is with a sinusoidal input. The waveform resulting from fixed-point processing is compared to the same waveform using the floating point C routines that are supplied. The distortion due to the fixed-point implementation is computed as a function of the rescaling parameters.
The second is the performance of this processor as an OFDM modulator. To test its performance you generate a random 16-QAM signal as input and generate the output waveform. The third is for a white-noise signal, which is important when used in a receiver configuration. The same procedure should be followed for the white-noise case. You will find some Gaussian random noise generators.

3 Report

Your report should include all the code that you developped (not much!) as well as some plots of distortion. You can use MATLAB for this.
You must specifically address the following questions.

1. Explain precisely the routines for simulation of fixed-point arithmetic.
2. Explain the choice for the twiddle factor quantization. How else could his have been done?
3. Explain the distortion test.
4. For the types of signals (and especially the noisy case) determine the dynamic range (in dB) of the algorithm, in both fixed-point configurations by varying the input signal strength over several tens of dB. The dynamic range becomes limited when the signal-to-distortion ratio starts to degrade significantly. You can plot the input signal strength versus the signal-to-distortion ratio to find the practical dynamic range.