SOFTWARE REQUIREMENTS SPECIFICATION

for

ADC++ Test Program

Version 0.1

Prepared by Attila Sárhegyi

Contents

1 Introduction

In the age of big data it is a crucial part of the design process to refine or readjust the models of IPs, sub-blocks, or even transistors to be able to anticipate the behavior of the circuit. Feedbacks from characterization is extremely important when it comes to failure analysis, debugging and CPK calculations in SPC.
The IEEE Standard 1241 has been developed to standardize the procedure of testing Analog to Digital Converters (ADCs). Even today, after more than 15 years there are still many inconsistencies between ways ADCs are specified and tested by the manufacturers. The first version of the proposed algorithms was implemented in Labview by a workgroup of the IEEE Instrumentation and Measurement Society in 1999. A free MATLAB version of the test program was developed in 2000 [1] which has been maintained and updated regularly. An article was also published to compare the two implementations [3].
The goal of this project is to develop a platform independent C++ library for testing Analog to Digital Converters (ADCs). The first step is to port the algorithms from MATLAB to C++ and fix the shortcomings of the MATLAB implementation. The next step is to add new features and algorithms to the library to be able to compare the different methods. The tool should be platform independent to be able to bridge the gap between Design, Test, Software, and Application engineers.

Planned fixes and features:

- Make it open source to eliminate licensing issues and make it available to a wider range of people.
- Support cross-compilation on various platforms (Linux, Windows) and embedded processors (SoCs).
- Develop algorithms to process long records (>1e6 samples) and be able to run them even on older machines without crashing it.

This project intends to accommodate these requirements and also provide an environment to test the algorithms on a broader spectrum of applications such as

- Mixed-Signal simulations during the design phase of an ADC
- Bench tests during bring-ups and characterization
- Built-In Self-Testing (BIST) of wafer and package units (class and sort BIST)
- High level modeling of ADCs in IDE tools

- Basic crosstalk analysis

This work provides the code base and the simulation environment for my PhD thesis [4].

2 How to get started

The ADC++ application was written in C++ which utilizes algorithms from the GNU Scientific Library (GSL). The base class of the program implements and extends algorithms developed in MATLAB [1]. A Python testbench was also developed to be able to benchmark and compare the algorithms in C++ vs. MATLAB by running Monte Carlo Simulations. Some of the Unit tests were also written in Python utilizing the Python testbench module.
There are two ways to use the ADC++ library. For everyday users there is no need to install MATLAB and the related python modules (python-matlab-bridge). Actually, the licensing issue was one of the reasons the program was implemented in C++. For developers MATLAB or OCTAVE is needed to be able to test the C++ algorithms against existing MATLAB/OCTAVE code. This feature will be maintained but it is not the focal point of this project.
ADC++ has been developed on the following platform and packages:

- Fedora 22 (kernel 4.4.13-200.fc22.i686+PAE)
  `https://getfedora.org/`
- g++ (GCC) 5.3.1 20160406 (Red Hat 5.3.1-6)
- GNU Scientific Library GSL-1.9
  `https://www.gnu.org/software/gsl/`
- Python (2.7.10 and 3.4.2)
- python-matlab-bridge (pymatbridge-0.5.2)
  `https://github.com/arokem/python-matlab-bridge`
- MATLAB (R2010a)

The application is intended to be platform independent and should be able to cross-compiled on embedded System-on-Chip processors. The following platforms are already in the pipeline but work still in progress:

- Windows 7 with Cygwin
- Discovery kit for STM32F7 MCU (STM32F746G-DISCO) with AC6 System Workbench for STM32
  `https://community.arm.com/docs/DOC-11293`

If you have found a bug, please report it on github
`https://github.com/asarhegyi/adcpp/issues`

2.1 Command Line Interface (CLI)

The current version of the application has to be compiled and invoked in a command line interface (CLI). The `examples/` folder contains four C++ source files that shows how to use the methods of the sine-fit (sfit) class. Note that none of the source files are compiled by default. There is a make file in the top directory that can be used to compile the source files manually.

```
999->[ati@asa02 adcpp] $cd examples/
1000->[ati@asa02 examples] $ls -la
total 76K
drwxrwxr-x.  2 jgipsz jgipsz 4.0K Jul 22 21:03 .
drwxrwxr-x. 13 jgipsz jgipsz 4.0K Jul 18 23:31 ..
-rw-rw-r--.  1 jgipsz jgipsz 5.0K Jul 22 15:36 debug_resolution.cpp
-rw-rw-r--.  1 jgipsz jgipsz 4.1K Jul 22 15:36 gen_rand_sine.cpp
-rwxrwxr-x.  1 jgipsz jgipsz  350 Jul 22 21:03 m_X.sed
-rw-rw-r--.  1 jgipsz jgipsz 1.9K Jul 22 21:02 plot_sfit.py
-rwxrwxr-x.  1 jgipsz jgipsz  469 Jul 22 21:01 run_sfit
-rw-rw-r--.  1 jgipsz jgipsz 6.7K Jul 22 20:10 sfit3_large.cpp
-rw-rw-r--.  1 jgipsz jgipsz 6.4K Jul 22 20:06 sfit3_linear.cpp
-rw-rw-r--.  1 jgipsz jgipsz 6.7K Jul 22 20:10 sfit4_large.cpp
-rw-rw-r--.  1 jgipsz jgipsz 6.4K Jul 22 20:06 sfit4_linear.cpp
-rwxrwxr-x.  1 jgipsz jgipsz  416 Jul 22 21:01 sfit_regression
1001->[ati@asa02 examples] $
```

The `examples/` folder contains four different flavours of the sine-wave fitting algorithms.

- sfit3_linear.cpp: 3-parameter sine-wave fit on a small record (less than 32768 samples).
- sfit3_large.cpp: 3-parameter sine-wave fit on a large record. It can process a million samples in a couple seconds.
- sfit4_linear.cpp: 4-parameter sine-wave fit on a small record (less than 32768 samples). It estimates the fundamental frequency as well. This is an iterative algorithm. Hence, it is slower than 3 parameter sine-wave fit.
- sfit4_large.cpp: 4-parameter sine-wave fit on a large record. Due to the iterative nature of the algorithm and the number of sample to process this is the slowest sine-fit method.

More details about the algorithms can be found in [4]. The following sections describe how you can compile and run `sfit3_linear`. The other three algorithms follow the same scheme.

2.1.1 Compile and run sfit3_linear

1. **Compile the C++ source files**

```
1001->[ati@asa02 examples] $make -f ../Makefile TARGET=sfit3_linear
g++ -Wall -DHAVE_INLINE -DGSL_C99_INLINE -I/usr/local/include
-L/usr/local/lib -osfit3_linear sfit3_linear.cpp -lgsl -lgslcblas -lm
1002->[ati@asa02 examples] $
```

2. **Print out the help message and get familiar with the syntax**

```
1002->[ati@asa02 examples] $./sfit3_linear -help
Usage is ./sfit3_linear [options] [file-name]
Options
  -v          Verbose mode. Overwrites verbosity level if used with
 debug mode!
  -d          Debug mode. It also set the verbosity level to max
  -t          Test time mode for the Monte Carlo simulations
  -f<number>  Fundamental frequency of the sine-wave
  -s<number>  Sampling frequency
  -r<number>  Number of harmonics
  -h[elp]     or
  -u[sage]    Print this help message
Example: ./sfit3_linear -f1000.0445712 -s500000 -r2
../charData/test.int
1003->[ati@asa02 examples] $
```

3. **Run example**
   At this point the example command at the bottom of the help message should work out of the box

```
1003->[ati@asa02 examples] $./sfit3_linear -f1000.0445712 -s500000 -r2
../charData/test.int
Data dumped into file residuals.dat

Best fit results:
C : 2.04701840120704946457e+03
A0: -1.18840899201339516367e+03
B0: 1.66200337650006486001e+03
A1: -6.05443115933445019650e-01
B1: -7.30346452901601717045e-01
A2: -7.71317038760628154170e-02
B2: -2.08841666839815406909e-02

Parameters of the sine-wave:
```

```
Amp0  : 2.04317673141505565582e+03
Phase0: 4.09163619294976044216e+00
Amp1  : 9.48666067642956467587e-01
Phase1: 2.26296033262118534424e+00
Amp2  : 7.99089992485763067620e-02
Phase2: 2.87717275673806138769e+00
Freq  : 1.00004457119999995030e+03
dc    : 2.04701840120704946457e+03

fs: 500000.00000000000000000000
Execution time: 0.186486 seconds

Data dumped into file fitted.dat

tss: 6.83726302273764114380e+10
Misc. parameters:
chisq: 1.47766440937575280259e+04
dof: 32761
chisq/dof: 4.51043743895410020883e-01
Rsq: 9.99999783880712977968e-01
erms: 6.71526165206215330805e-01
1004->[ati@asa02 examples]
```

2.1.2 Results

The results are printed on the CLI. In this example above the input parameters of the sine-fit algorithm are

1. Fundamental frequency of the sine-wave (known for 3-parameter sine-fit)
   `-f1000.0445712`
2. Sampling frequency
   `-s500000`
3. Number of harmonics to estimate
   `-r2`

Assuming the data record contains the sequence of M samples $y_1, y_2 \cdots, y_M$, taken at times $t_1, t_2 \cdots, t_M$, the sfit3 algorithm finds the values of A_0, B_0, and C that minimizes the following sum of squared differences [2]:

$$\chi^2 = \sum_{n=1}^{M} [y_n - A_0 \cos(\omega_0 t_n) - B_0 \sin(\omega_0 t_n) - C]^2 \tag{2.1}$$

The fundamental component of the fitted sine-wave is given by Eq. 2.2

$$\hat{y}_n = A_0 \cos(\omega_0 t_n) + B_0 \sin(\omega_0 t_n) + C, \tag{2.2}$$

which can be converted into Eq. 2.3

$$\hat{y}_n = \mathrm{Amp}_0 \cos(\omega_0 t_n + \mathrm{Phase}_0) + \mathrm{dc}, \tag{2.3}$$

where

$$\omega_0 = 2\pi \cdot \mathrm{Freq}.$$

These equations expend if the parameters of the harmonics are estimated as well. In this example, the first and second harmonics calculated at the frequencies of $2 \cdot \mathrm{Freq}$ and $3 \cdot \mathrm{Freq}$, respectively. Hence, Eq. 2.2 and 2.3 needs to be updated as follows

$$\hat{y}_n = \sum_{i=0}^{2} A_i \cos((i+1)\omega_0 t_n) + B_i \sin((i+1)\omega_0 t_n) + C \tag{2.4}$$

$$\hat{y}_n = \sum_{i=0}^{2} \mathrm{Amp_i} \cos((i+1)\omega_0 t_n + \mathrm{Phase_i}) + \mathrm{dc} \tag{2.5}$$

The residuals, r_n of the fit are given by Eq. 2.6

$$r_n = y_n - \left(\sum_{i=0}^{2} A_i \cos((i+1)\omega_0 t_n) + B_i \sin((i+1)\omega_0 t_n) + C \right) \tag{2.6}$$

The total sum of squares (TSS) of data about the mean

$$\mathrm{TSS} = \sum (y_i - \hat{y})^2, \tag{2.7}$$

where the mean value is estimated by

$$\hat{\hat{y}} = \frac{1}{M} \sum y_i. \tag{2.8}$$

The coefficient of determination R^2 is a statistical measure of the goodness of the least squares estimation. R^2 is expressed as a ratio of the variance of the fitted model's prediction to the total sample variance. It ranges from 0 to 1 where $R^2 = 1$ indicates that the fitted models explains all samples, while $R^2 = 0$ indicates that there is no correlation between the model and the samples.

$$R^2 = 1 - \frac{\chi^2}{\mathrm{TSS}} \tag{2.9}$$

And the root mean square (rms) error is given by Eq. 2.10

$$e_{rms} = \sqrt{\left(\frac{1}{M} \sum_{n=1}^{M} r_n^2 \right)} \tag{2.10}$$

Further analysis of the algorithms and the formulas can be found in [2] and [4].

2.1.3 Waveforms

As a result of the sine-fit two .dat files get generated to store the fitted sine-wave (fitted.dat) and the residuals (residuals.dat), which is the delta between the observations and the fitted sin-wave. The data vectors can be plotted by running the `plot_sfit.py` script. Note that fitted.dat and residuals.dat have to be located in the current directory while the path to the observation vector has to be passed into the script.

```
1006->[ati@asa02 examples] $python ./plot_sfit.py ../charData/test.int
Loading ../charData/test.int
Loading ./fitted.dat
Loading ./residuals.dat
1007->[ati@asa02 examples]
```

The waveforms are depicted in Figure 2.1.

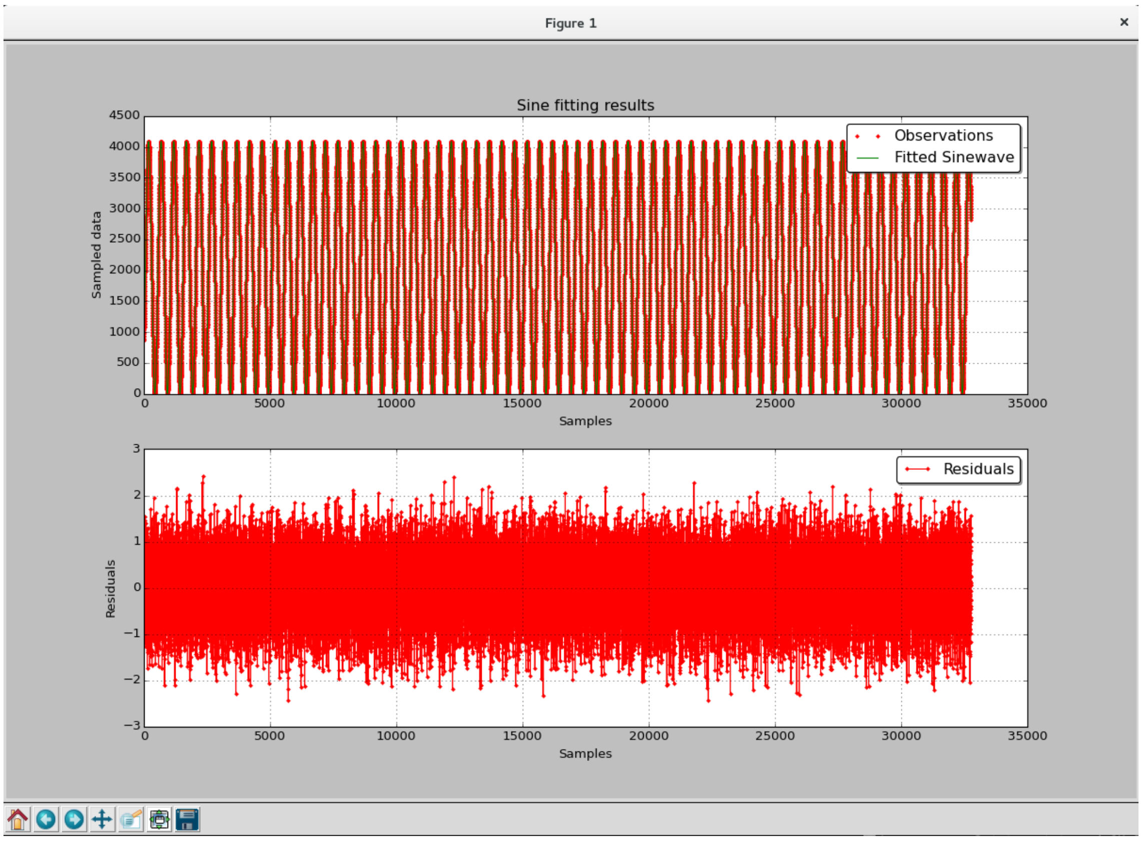

Figure 2.1: Run plot_sfit.py to plot the results of the sine-wave fitting

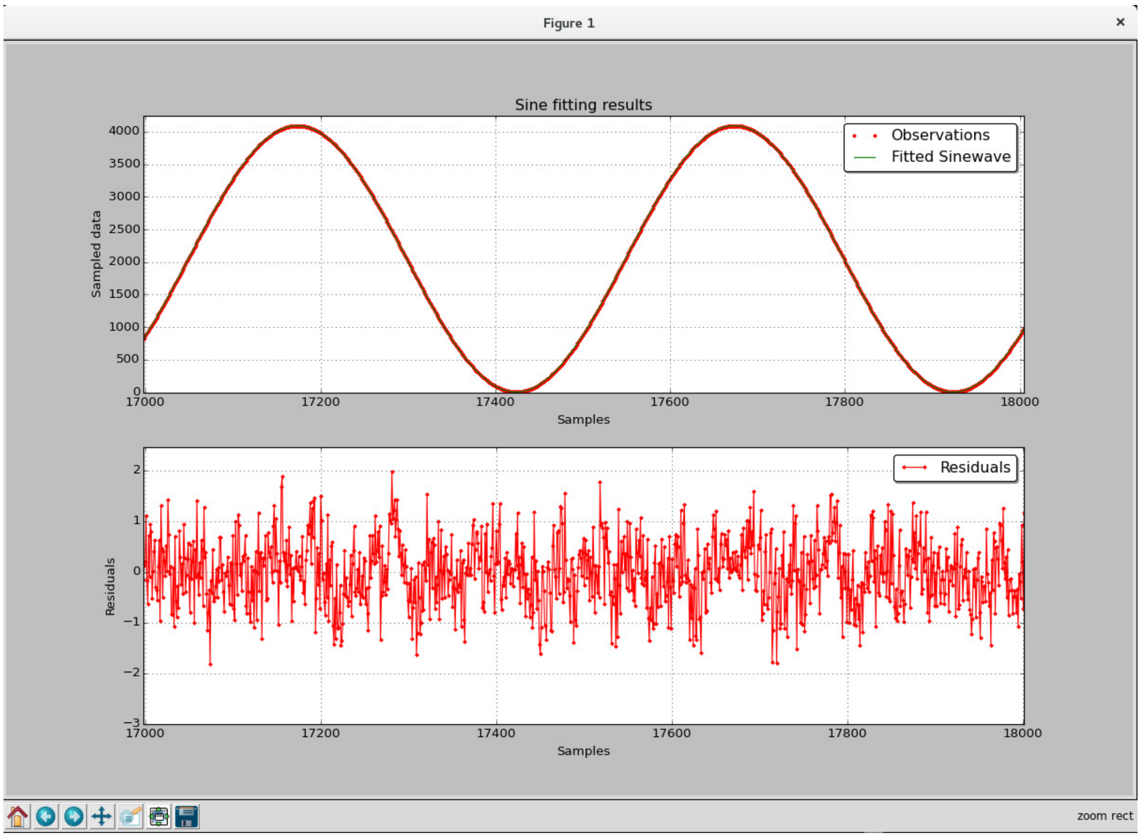

Figure 2.2: Zoomed in version of the waveforms depicted above

3 Test Plan and Specification

This chapter summarizes the test environment in which the Least-Squares sine-wave fitting (sfit) and Maximum Likelihood Estimation (MLE) algorithms were simulated and verified. The discussion is broken down to two section such as Unit tests, Monte Carlo simulations.

3.1 Unit tests

3.1.1 File interface

The samples of the observations are stored in text files. The fileio class in `/src/utils.h` contains all of the functions that load, stream, decimate, and save data samples from and to text files. The interface is tested with `/tests/fileio.cpp` by loading/streaming data from `/charData/test.int` into a vector array and save that array in a separate file. The two files can be compared by `diff` or similar programs. Note that some files might contain multiple columns (time and data), a header section, and comments. The fileio class should take care of them. An example can be found in `/charData/dacout_sine_short.txt`

3.1.2 Python-Matlab-bridge

The `/tests/mypymatbridge.py` script tests the python-matlab-bridge setup. First it searches for a matlab executable. If matlab is on the path it creates a matlab session, and connects the python interpreter. After that, it calls `/matlab/adder.m` 800 times, which emulates a counter. Once it is done counting it shuts down the matlab server.

```
1042->[ati@asa02 adcpp] $
1042->[ati@asa02 adcpp] $cd tests/
1043->[ati@asa02 tests] $
1043->[ati@asa02 tests] $python mypymatbridge.py
Found MATLAB in /home/jgipsz/R2010a/bin/matlab
Starting MATLAB on ZMQ socket ipc:///tmp/pymatbridge-edbac200-81ca-4dc1
-8907-858864d6eed7
Send 'exit' command to kill the server
................MATLAB started and connected!
1
2
3
4
5
```

```
.
.
.
795
796
797
798
799
800
MATLAB closed
1044->[ati@asa02 tests] $c
```

3.1.3 Interpolated FFT

The 4-parameter sine-wave fitting methods (sfit4) are iterative algorithms. An initial estimate of the frequency is required to make sure the algorithm converges. In addition to using FFT for the initial estimate of the frequency an Interpolated FFT (IpFFT) method was added to the spectrum class `/src/spectrum_class.h`. The IpFFT algorithm can be tested by compiling and running `/tests/spectrum.cpp`.

```
1067->[ati@asa02 tests] $make -f ../Makefile TARGET=spectrum
g++ -Wall -DHAVE_INLINE -DGSL_C99_INLINE -I/usr/local/include
-L/usr/local/lib -ospectrum spectrum.cpp -lgsl -lgslcblas -lm
1068->[ati@asa02 tests] $./spectrum
File ../charData/test.int opened successfully. Reading data from file...

Calculating FFT...

Data dumped into file magnitude.dat

Post-processing FFT results...

maxIndex           : 66
magnitude[maxIndex]: 1397.39
maxFrequency       : 1007.08
magnitude[0]       : 2055.64
Ip. Frequency      : 1000.04
1069->[ati@asa02 tests] $g magnitude.dat
1070->[ati@asa02 tests] $
```

In the example above, the IpFFT method improved the frequency estimation of the fundamental frequency of the sine-wave from 1007.08Hz to 1000.04Hz.

3.1.4 Stimulus generation

In order to be able to run Monte Carlo simulations, known stimuli have to be applied as an input to the sine-fit algorithms. In order to do so, a zero order model of the ADC was created as shown in Figure 3.1. This is a black-box behavioral model of the ADC which can induce noise and distortion to the sine-wave. The output of the Quantizer represents the digital output of the ADC. The Quantizer itself has a static model, which means hysteresis, and frequency dependent behaviors were not modelled yet. The Quantizer block has two inputs:

1. The first one is the signal path input which can be either a sine-wave or a ramp signal. The ramp waveform is used only for debug purposes right now. Gaussian noise and distortion can be added to the ideal excitation signal to model a non-ideal signal generator and the channel between the generator and the input of the ADC. In the zero order model the Least-Squares (LS) sine-fit methods estimates the parameters of the signal path (sine-wave, noise, and distortion).

2. The second input to the Quantizer is an abstract input to the black-box model, which represents the parameter vector of the Code Transition Levels (CTL) $T[1]$,$T[2]$, $\cdots$, $T[2^N - 1]$, where N is the number of bits. The ideal Quantizer has uniformly distributed CTLs. A non-ideal Quantizer can be modelled by deviating from the ideal CTLs, which can be done by adding independent identically distributed (IID) noise and distortion to the CTL vector. The Maximum Likelihood Estimation (MLE) method estimates the CTL vector and refines the results of the sine-fit method.

The first set of Monte Carlo simulations intend to verify the performance of the sine-fit algorithms when only Gaussian noise induced in the model. Hence, both distortion blocks were bypassed. The effect of the distortions will be discussed in the next release.

A ramp signal was used to verify the distribution of the code transition levels. The two corner cases when mutually exclusive Gaussian noise was added to either the signal path or the CTLs were depicted in Figure 3.2 and 3.3, respectively. Note that, the analog input was scaled and aligned with the digital output to be able to plot them in the same figure.

The same corner cases were shown in Figure 3.4 and 3.5 when the excitation signal was a sine-wave.

3.2 Monte Carlo simulations

The sine-fit and MLE algorithms were verified by running Monte Carlo (MC) simulations and calculating the estimation errors of the ADC parameters. The MATLAB versions of the sfit algorithms were provided as references to the mean and standard deviation of the estimation errors. Further analysis of the histograms might results in updating the requirements later. The first set of MC simulations characterized the distribution,

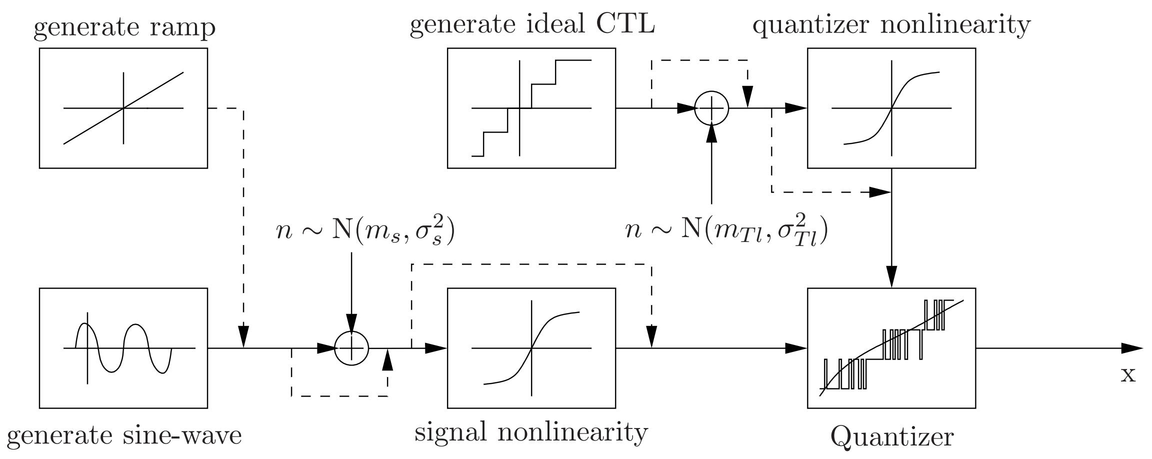

Figure 3.1: Block diagram of the stimulus generation. Noise and distortions are added to the ideal sine-wave signal and to the Code Transition Levels (CTL) which results in a non-ideal quantization. The noise sources and nonlinearities can be bypassed. A ramp signal can be generated for debug purposes.

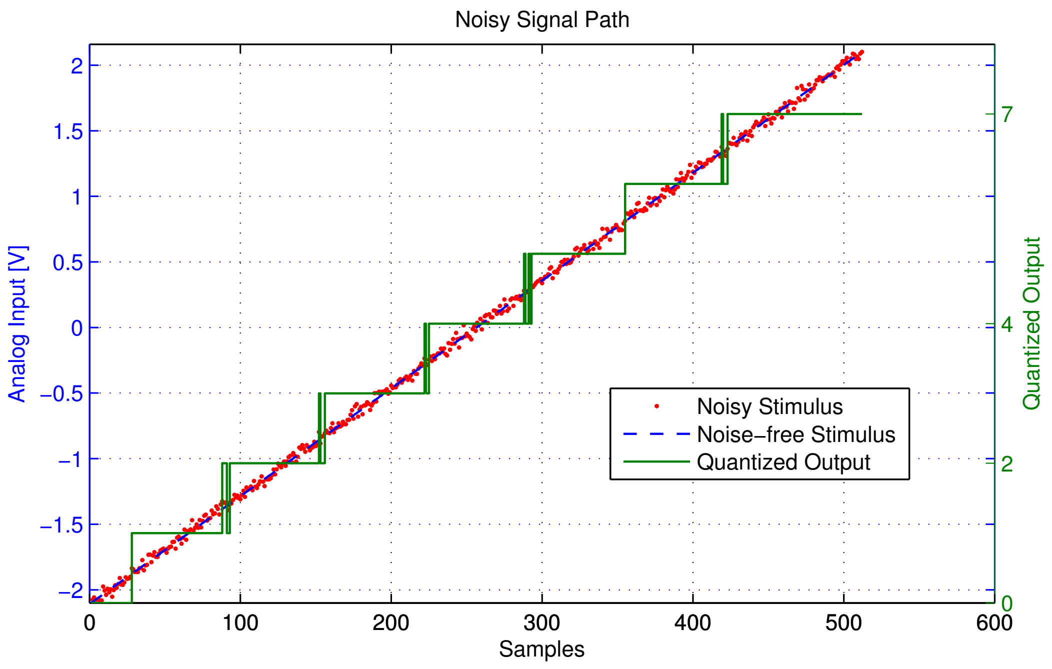

Figure 3.2: Noisy Signal Path with ramp stimulus

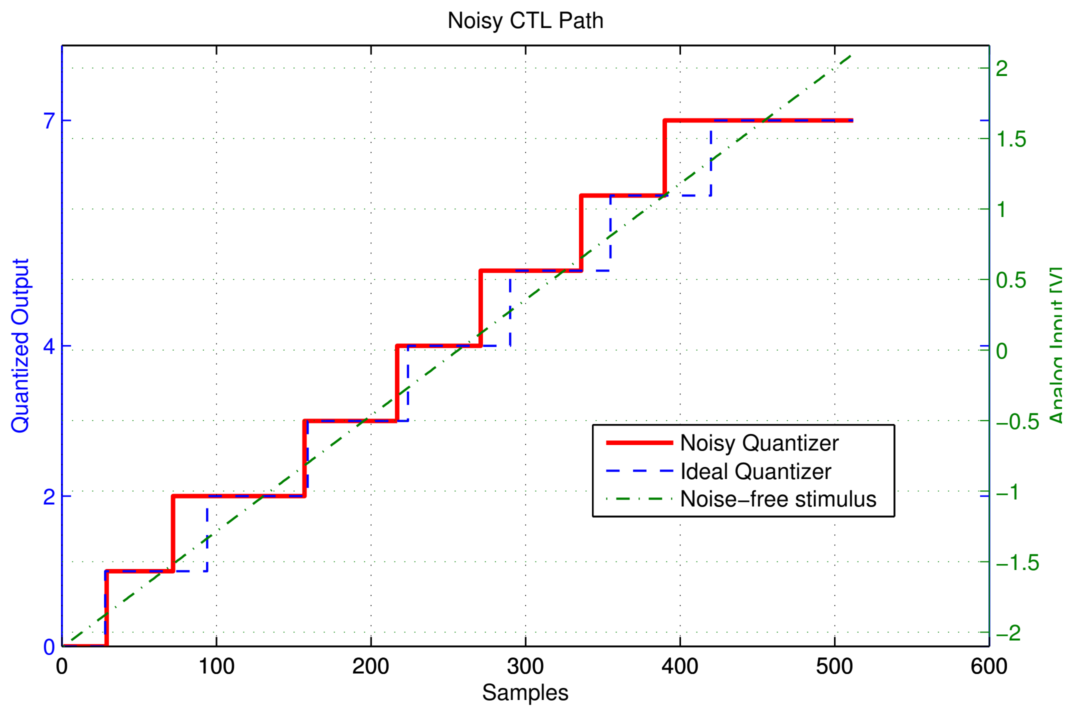

Figure 3.3: Noisy CTL Path with ramp stimulus

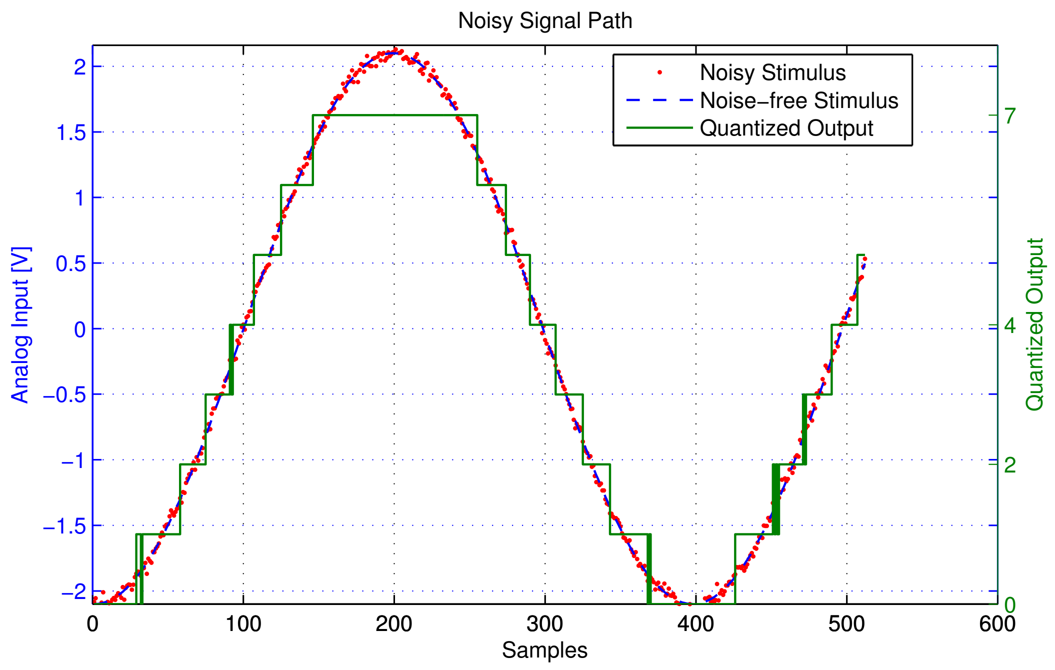

Figure 3.4: Noisy Signal Path with sine stimulus

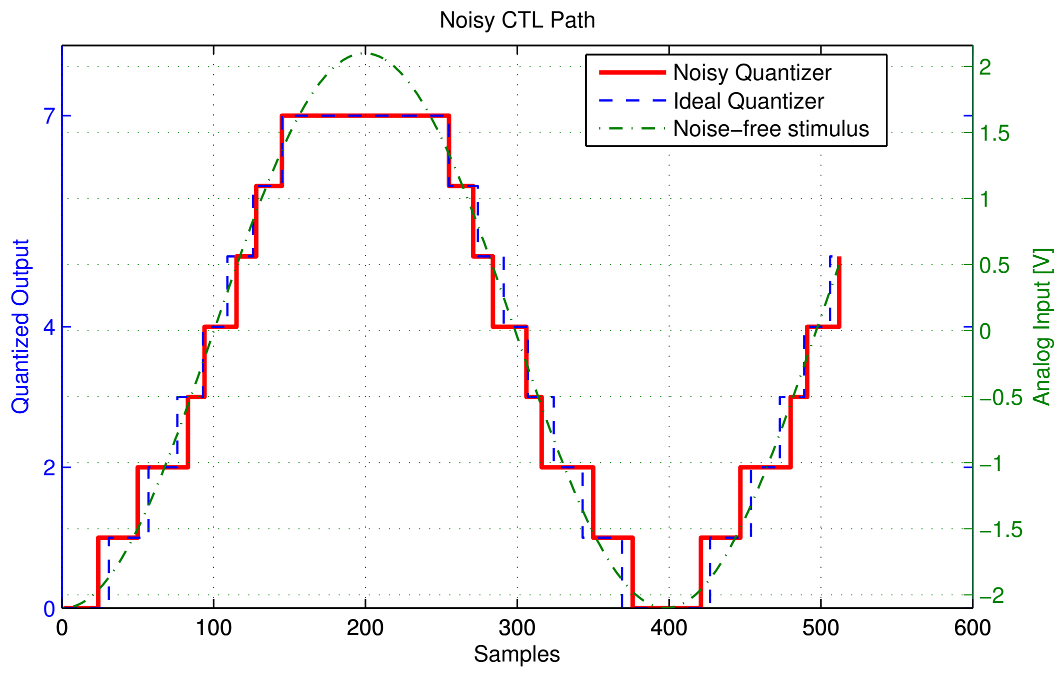

Figure 3.5: Noisy CTL Path with sine stimulus

bias, and the variance of the estimation when only Gaussian noise was added to the sine-waves and the CTLs (the distortions were bypassed in Figure 3.1).
Parameters to qualify for the sfit algorithms are:

1. Amplitude of the sine-wave
2. Frequency of the sine-wave
3. Phase of the sine-wave
4. DC voltage of the sine-wave

A testbench was developed in python to run the MATLAB and C++ code parallel, analyze the results, and plot the histograms. A flowchart of the testbench is shown in Figure 3.6. The two sine-fit algorithms under test were the 3-parameter and the 4-parameter least-squares fit to sine wave data [2] referred as sfit3 and sfit4, respectively. Each of these algorithms were implemented in MATLAB (sfit3.m and sfit4imp.m) and C++ as well. There were two versions of the C++ implementation, one for processing a small set of data (sfit3_linear, sfit4_linear), and another one for processing large data sets (sfit3_large, sfit4_large). This resulted in running MC simulations on a total of six algorithms. In Figure 3.6 `sfit_montecarlo.py` was a testbench wrapper which called `gend_rand_sine.cpp` to generate a random sine-wave and fed that waveform into the sine-fit algorithms. The results were stored in text files (*.est), which were post processed to get the histograms.

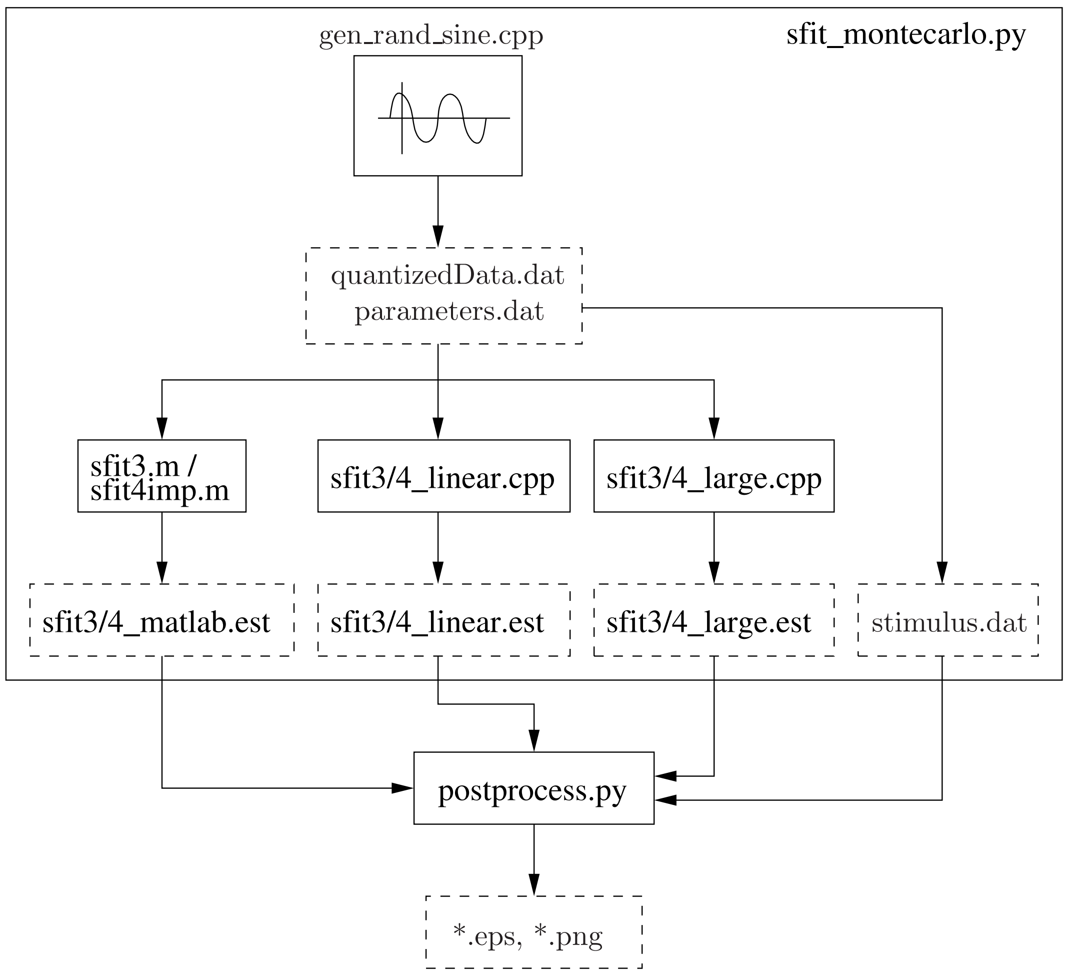

Figure 3.6: Data flow diagram of the Monte Carlo simulation. Random quantized sine-waves are generated by gen_rand_sine.cpp which is fed into the sfit3 and sfit4 algorithms (one MATLAB and two C++ versions). The results of the estimations are stored in text files (*.est) which are post-processed by post-process.py to plot the histograms of the estimated parameters.

The histograms were organized in two subsections, one for sfit3, and another one for sfit4. Each subsection was divided further down to histograms and pairwise delta histograms subsections. The distribution of the estimated parameters were shown in the histograms subsections. These charts looked quite similar so, the pairwise delta histograms were introduced to analyze the differences between the algorithms.

3.2.1 sfit3 histograms

Since there was no visible difference between the histograms for sfit3, only MATLAB results are show in this subsection.

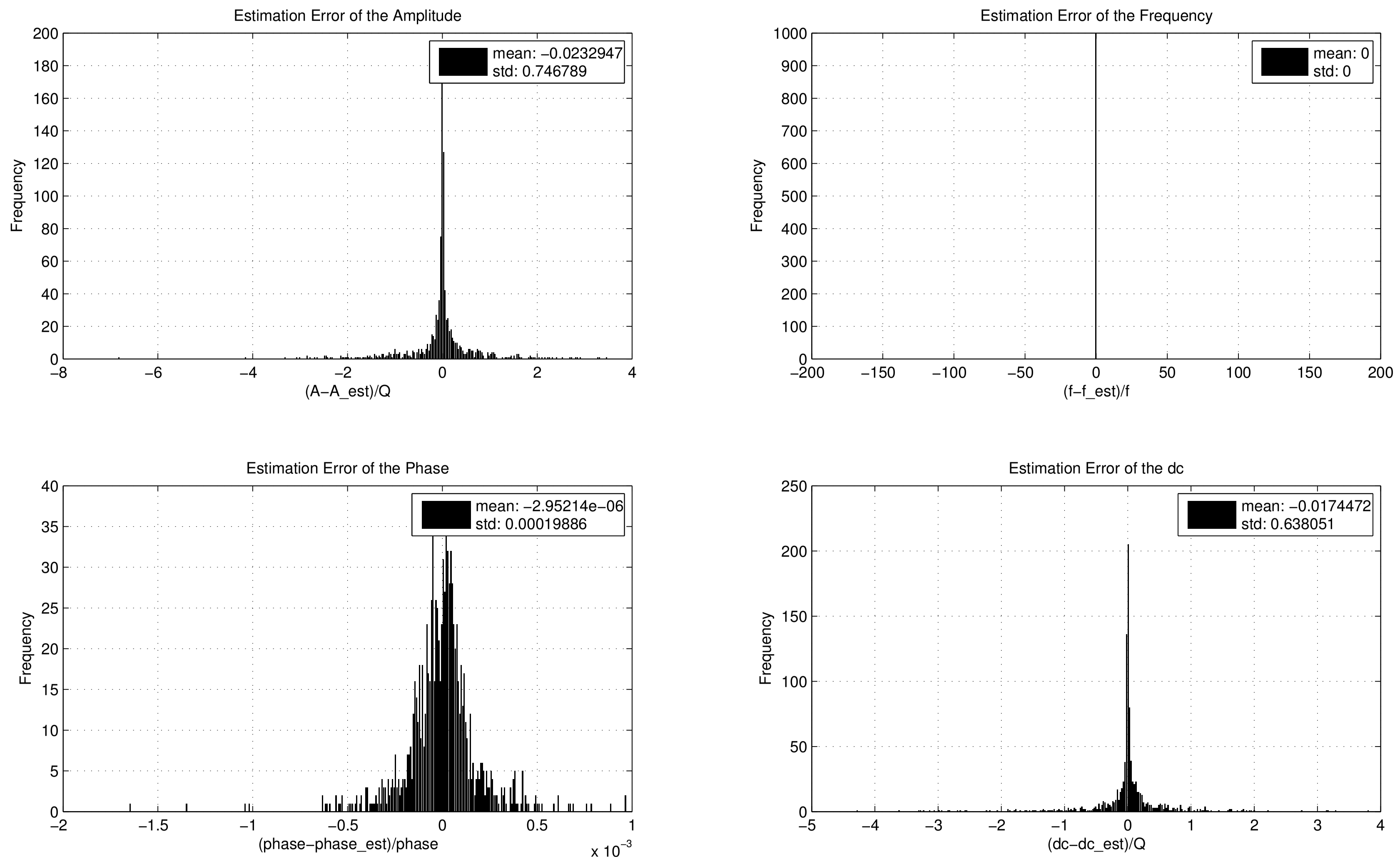

Figure 3.7: Results of a Monte Carlo simulation - sfit3 MATLAB histograms (frequency is known).

3.2.2 sfit3 pairwise delta histograms

Since there is no visible difference between the histograms for sfit3, the pairwise delta histograms of the estimations were calculated and plotted below. beginlandscape

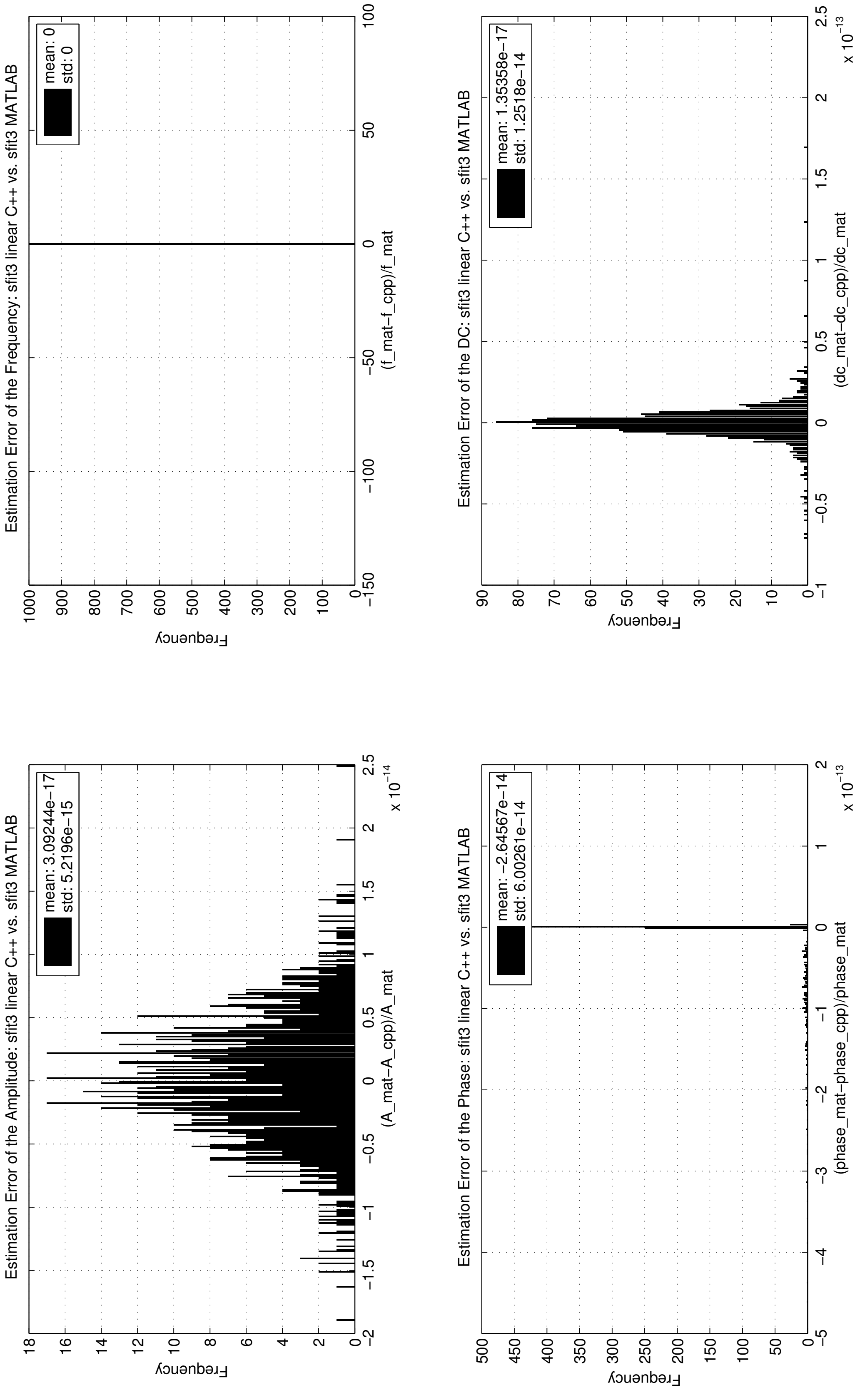

Figure 3.8: Results of a Monte Carlo simulation - sfit3 linear C++ vs. sfit3 MATLAB histograms (frequency is known).

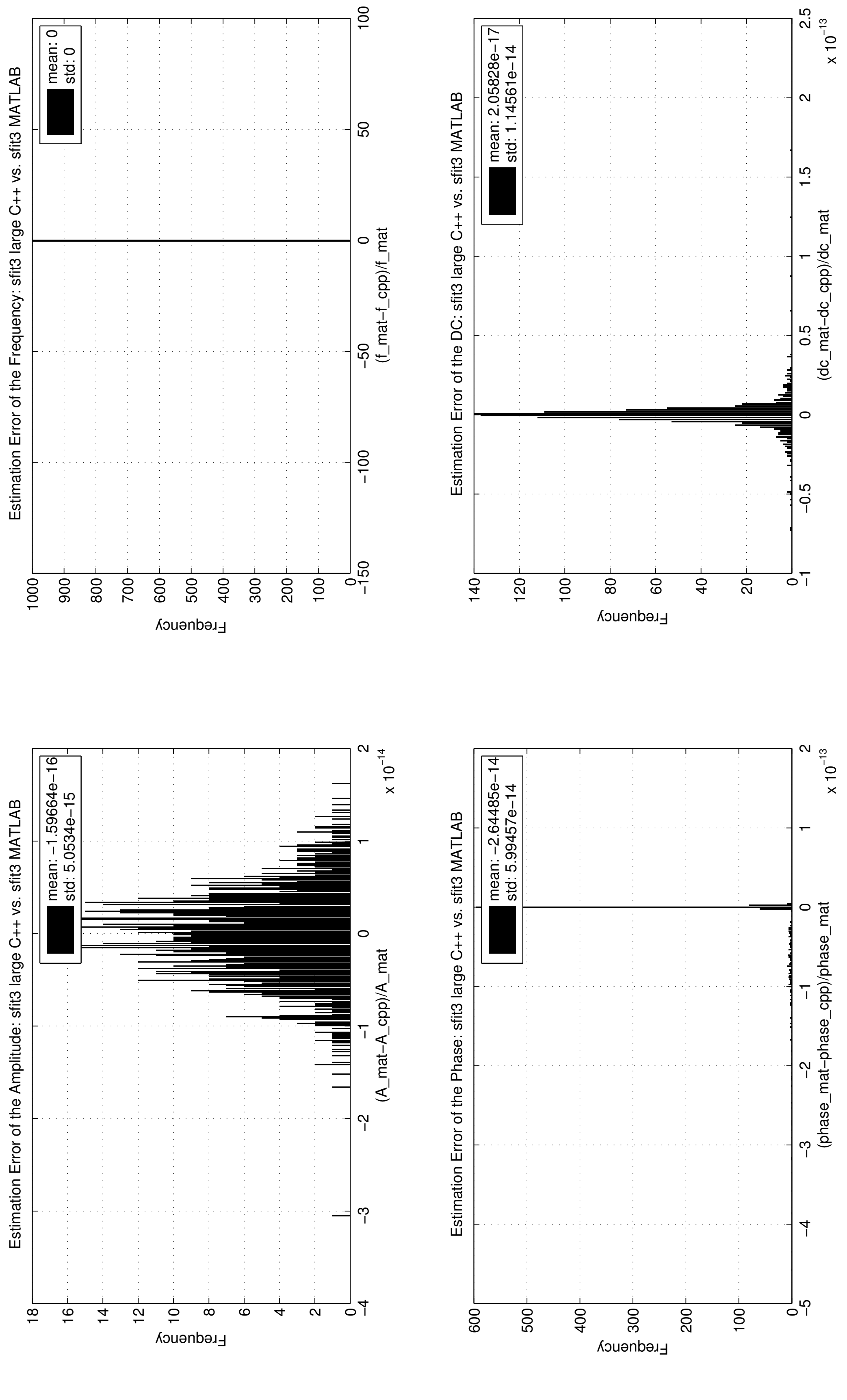

Figure 3.9: Results of a Monte Carlo simulation - sfit3 large C++ vs. sfit3 MATLAB histograms (frequency is known).

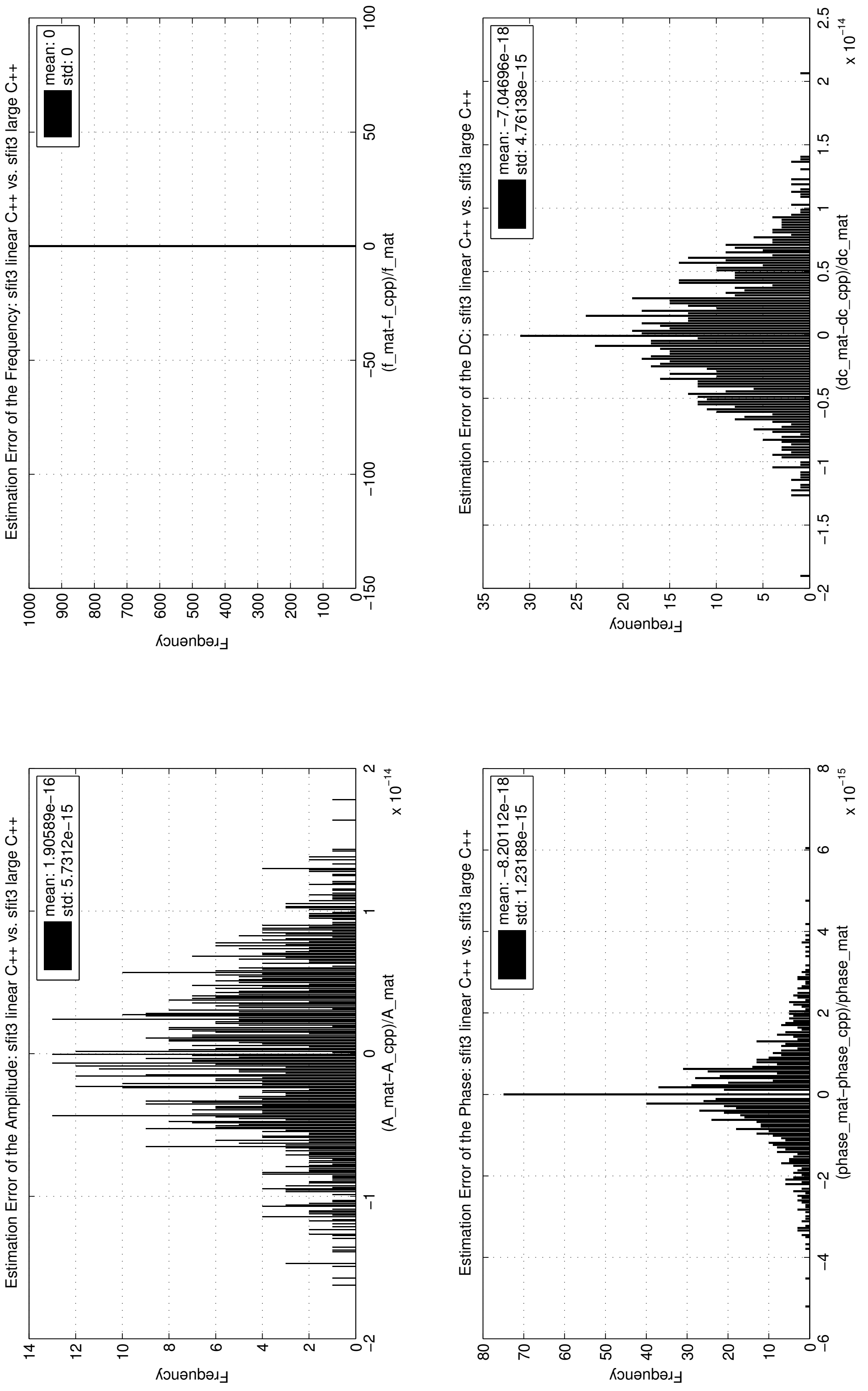

Figure 3.10: Results of a Monte Carlo simulation - sfit3 linear C++ vs. sfit3 large C++ histograms (frequency is known).

3.2.3 sfit4 histograms

There were small visible differences between the sfit4 algorithms, which is expected due to their iterative nature.

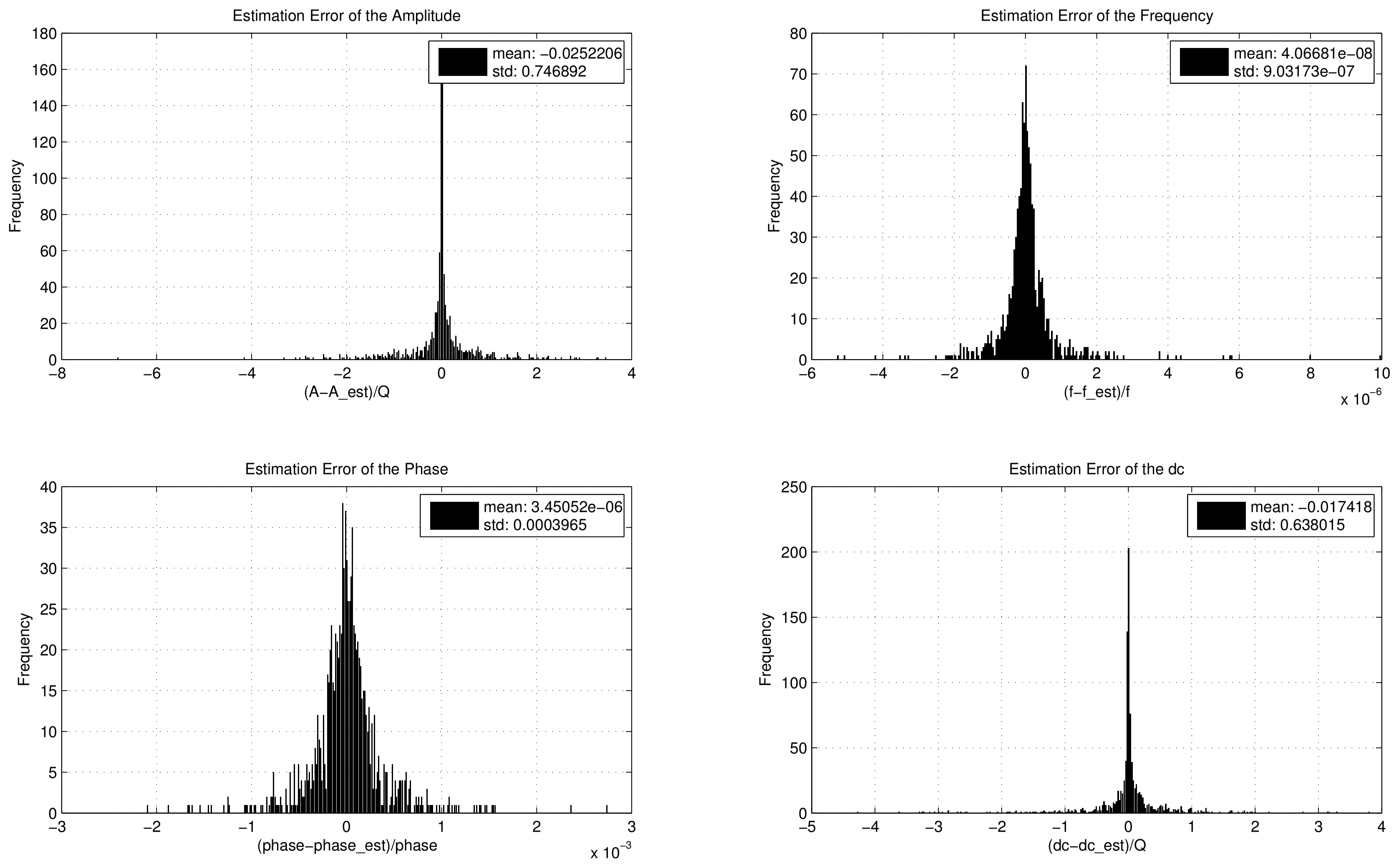

Figure 3.11: Results of a Monte Carlo simulation - sfit4 MATLAB histograms.

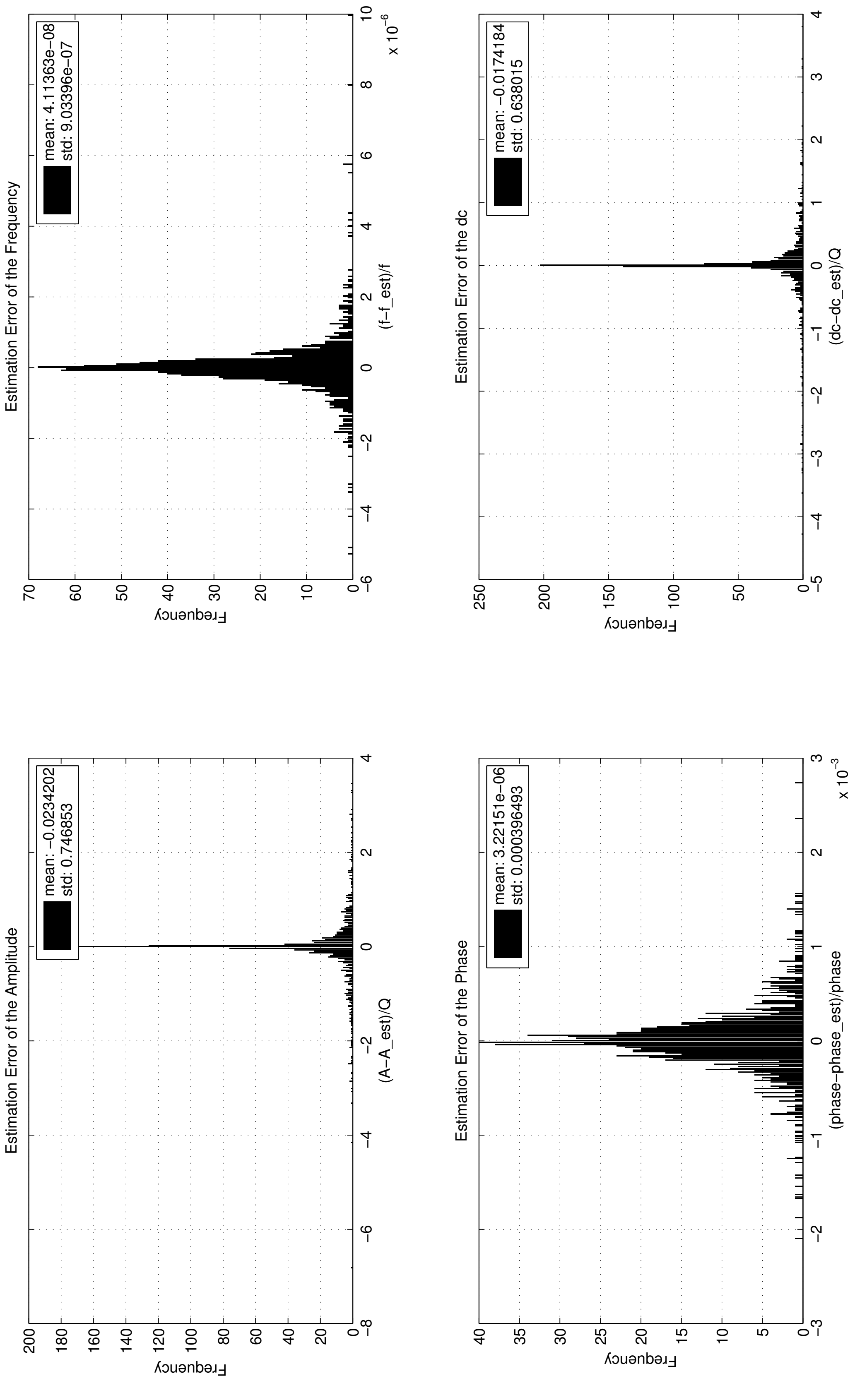

Figure 3.12: Results of a Monte Carlo simulation - sfit4 linear C++ histograms.

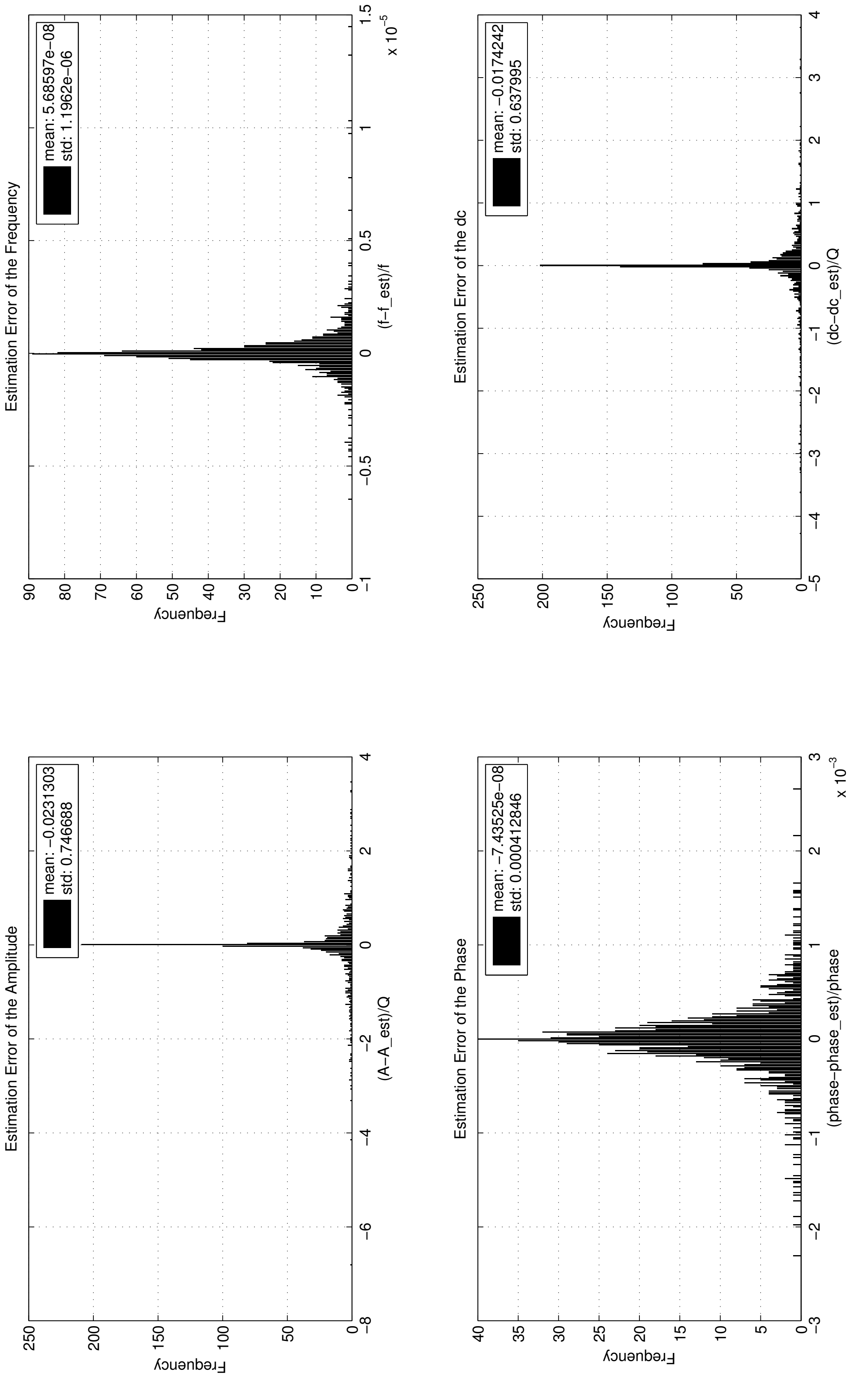

Figure 3.13: Results of a Monte Carlo simulation - sfit4 large C++ histograms.

3.2.4 sfit4 pairwise delta histograms

The histogram of the pairwise differences of the estimations are calculated and plotted below. Due to the iterative nature of the sfit4 algorithms the requirements should be less stringent than the ones for sfit3. beginlandscape

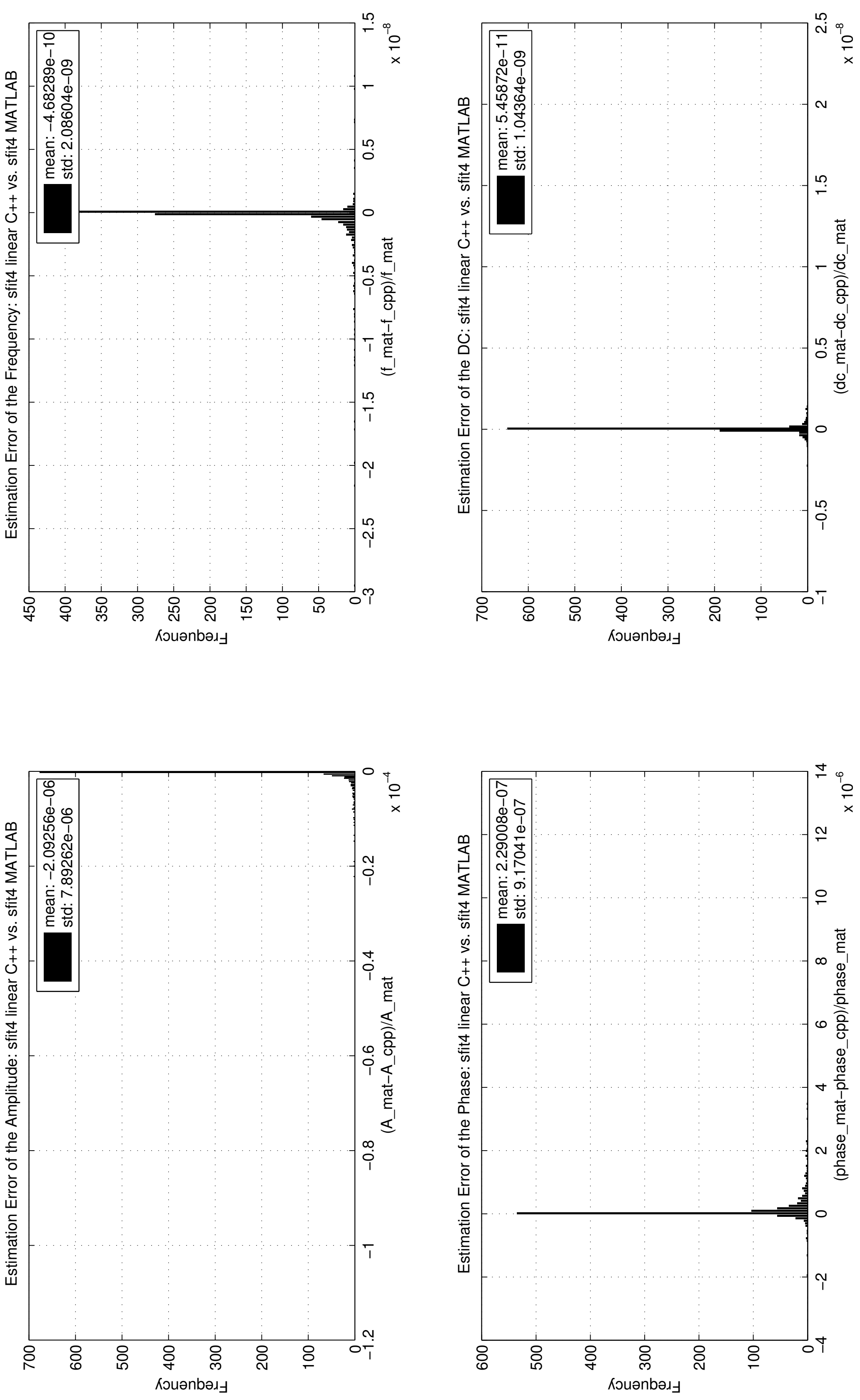

Figure 3.14: Results of a Monte Carlo simulation - sfit4 linear C++ vs. sfit4 MATLAB histograms.

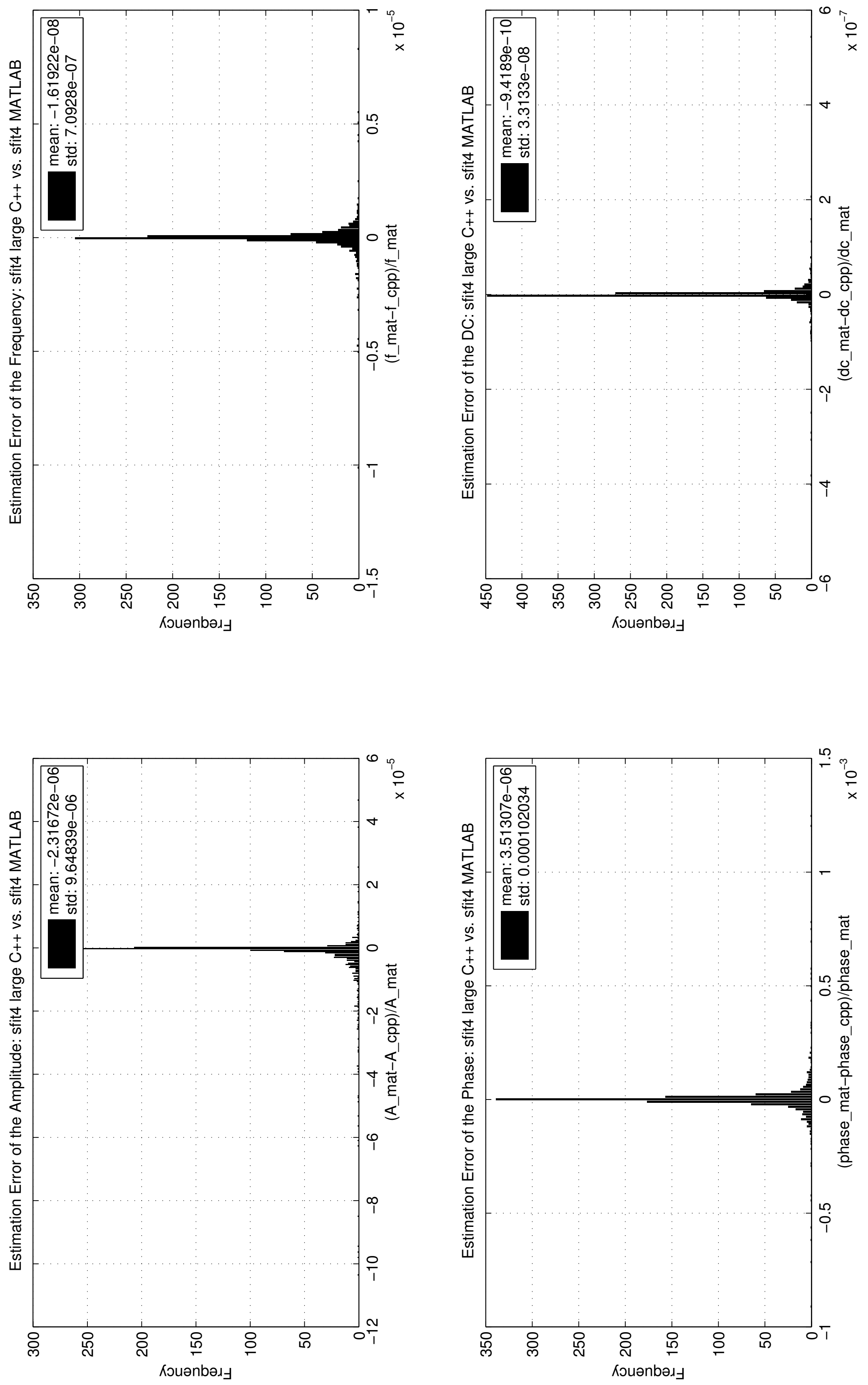

Figure 3.15: Results of a Monte Carlo simulation - sfit4 large C++ vs. sfit4 MATLAB histograms.

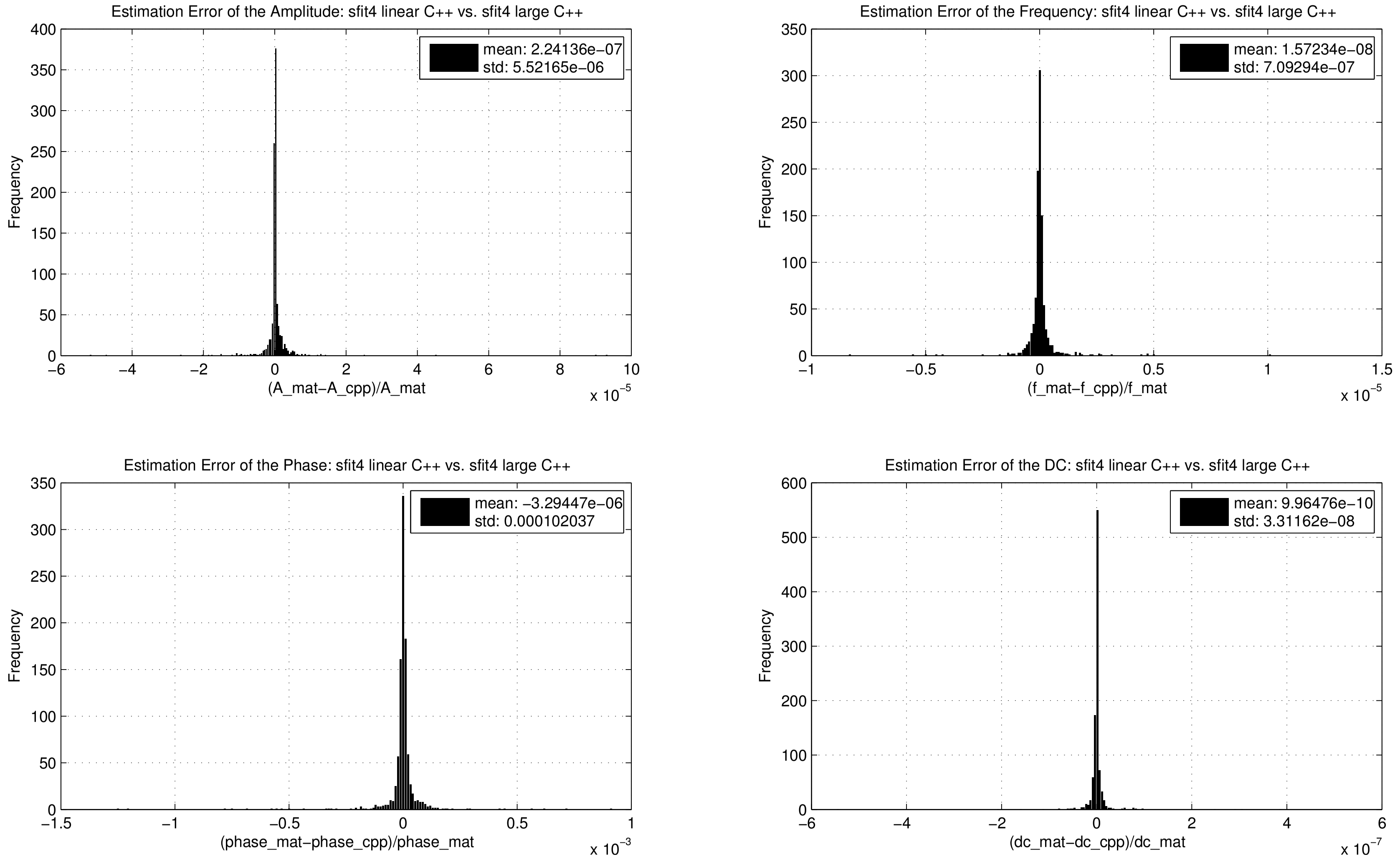

Figure 3.16: Results of a Monte Carlo simulation - sfit4 linear C++ vs. sfit4 large C++ histograms.

Bibliography

[1] ADC testing toolbox for MATLAB. `http://www.mit.bme.hu/projects/adctest/`.

[2] IEEE standard for terminology and test methods for analog-to-digital converters. *IEEE Std 1241-2000*, page 92, 2001.

[3] István Kollár and János Márkus. Sine wave test of ADC's: Means for international comparison. In *Proceedings of the IMEKO TC4 5th European Workshop on ADC Modelling and Testing (EWADC)*, pages 211–16, Vienna, Austria, 25–28 September 2000.

[4] Attila Sárhegyi. *Maximum Likelihood Estimation of ADC Parameters.* PhD thesis, Budapest University of Technology and Economics, Department of Measurement and Information Systems, Magyar tudósok körútja 2. H-1117 Budapest, Hungary, June 2017.