User Manual to the FFAST Code Package

BASiCS

June 8, 2016

1 Installation

1.1 Required software and packages

- C++ compiler supporting ISO C++ 2011 standard
- FFTW 3 library for computing FFT, website: http://fftw.org.

If the required software listed above are not installed, they can be obtained as follows.

1.1.1 Installing required software on Ubuntu

Linux distributions usually come with an installed C++ compiler. If FFTW 3 is not installed, it can be obtained by running the command line command:

```
sudo apt-get install libfftw3-dev libfftw3-doc
```

1.1.2 Installing required software on Mac OS X

Apple has their distribution of developer tools under the software bundle called XCode which is available on AppStore. These tools come with a C++ compiler. If you do not have have FFTW 3 we suggest installing it through a package manager such as Homebrew. The following steps can be taken to install the required software.

1. Download XCode, available on AppStore
2. Install Homebrew, website: http://brew.sh/
3. Install FFTW 3 through brew, command line: `brew install fftw`

1.2 Building FFAST

Makefile is provided with the distribution and FFAST should compile without a problem by calling `make` from the command line in the same directory of the Makefile.

2 Using the code

The FFAST engine takes an input object and an output object that describes the input/output operations. The implementation of these two can be tailored to application without changing the FFAST engine.

The Makefile compiles an example program `ffast`. It implements an *experiment* and a *customized* mode.

The experiment mode generates random input signals according to the specifications given from the command line and test the performance of FFAST algorithm.

Customized mode takes a text file from the command line that stores the signal and outputs the frequency contents to a desired file. The text file should have the values of the signal at each time index in tuple form (a, b), where a denotes the real part and b denotes the complex part. For example, the text file with content

```
(1,0)
(0,1)
(-1,0)
(0,-1)
```

defines a length 4 input x where $x[0] = 1$, $x[1] = i$, $x[2] = -1$, and $x[3] = -i$.

2.1 Arguments

-a

Run experiment

-c

Do not count the number of samples used by FFAST to speed up the algorithm

-n NUM

Signal length

-i NUM

Number of iterations

-f FNAME
Input file name

-z FNAME
Output file name

-g NUM
Minimum magnitude of frequency wanted to be recovered

-k NUM
Number of non-zero frequencies, sparsity

-s NUM
SNR in dB

-d NUM
Number of delays per chain

-e NUM
Number of chains

-l
Use ML decoding, slow search

2.2 Examples

Below we list some example calls for the executable file.

./ffast –help
Display help.

./ffast -a -n 124950 -k 10 -i 30
Run 30 FFAST experiments on randomly generated signals of length 123950 having 10 sparse Fourier spectrum.

./ffast -a -n 124950 -s 10 -k 10 -i 30
Run 30 FFAST experiments on randomly generated signals of length 123950 having an SNR of 10 dB and 10 sparse Fourier spectrum.

./ffast -f inFile.txt -k 40 -z outFile.txt
Run FFAST on input data given in `inFile.txt` to recover 40 sparse spectrum and write the recovered signal in `outFile.txt`.

./ffast -a -d 2 -e 9 -n 166155 -k 50 -s 5 -b “53 55 57” -i 30

Run 30 FFAST experiments on randomly generated signals of length 166155 having an SNR of 5 dB and 50 sparse Fourier spectrum. For the algorithm, use 3 stages with bin sizes 53, 55 and 57, and for the delays use 9 clusters with 2 delays in each (cf. [1]).

3 Code structure

config.cpp

This class gets input arguments from the command line and stores the parameters of the algorithm.

input.cpp

This is a template class. Instances of this class should implement a function that will return the value of the signal at a chosen time index. Two such instances of this class are the `experimentinput.cpp` and `customizedinput.cpp`.

frontend.cpp

This class is where sub-sampling and short discrete time Fourier transforms are done.

backend.cpp

The peeling engine is implemented here.

output.cpp

This is a template class. Instances of this class should implement a function that will get the results from the backend and output. Two instances are `experimentoutput.cpp` and `customizedoutput.cpp`.

ffast.cpp

The class that runs the FFAST engine.

main.cpp

The implementation for the executable file with the experiment and customized modes.

References

[1] S. Pawar and K. Ramchandran, “A robust sub-linear time R-FFAST algorithm for computing a sparse DFT,” *ArXiv*, vol. abs/1501.00320, 2015. [Online]. Available: http://arxiv.org/abs/1501.00320