# fir1

Generated by Doxygen 1.8.17

# 1 FIR1

An efficient finite impulse response (FIR) filter class in C++, JAVA wrapper for Android and Python wrapper.

The floating point class offers also adaptive filtering using the least mean square (LMS) or normalised least mean square (NLMS) algorithm.

## 1.1 Installation

### 1.1.1 Ubuntu packages for xenial, bionic and focal

Add this repository to your package manager:

```
sudo add-apt-repository ppa:berndporr/dsp
sudo apt-get update
sudo apt install fir1
sudo apt install fir1-dev
```

This adds `fir1-dev` and `fir1` to your package list. The demo files are in `/usr/share/doc/fir1-dev`. Copy them into a working directory, type `gunzip *.gz`, `cmake .` and `make`.

### 1.1.2 MacOS packages (homebrew)

Make sure you have the homebrew package manager installed: https://brew.sh/

Add the homebrew tap:
```
brew tap berndporr/dsp
```

and then install the fir filter package with:
```
brew install fir
```

### 1.1.3 Linux / Unix / MACOSX: compilation from source

The build system is `cmake`. Install the library with the standard sequence:
```
cmake .
make
sudo make install
sudo ldconfig
```

or for debugging run cmake with:
```
By default optimised release libraries are generated.
### Windows
Under windows only the static library is generated which
should be used for your code development.
```

cmake -G "Visual Studio 15 2017 Win64" . ``` and then start Visual C++ and compile it. Usually you want to compile both the release and debug libraries because they are not compatible to each other under Windows.

### 1.1.4 Android / JAVA

The subdirectory `firj` contains an Android project. Load it into Android studio and build it either as a release or debug binary. This generates an Android aar which you import into your project. See the `InstrumentedTest.java` for an instructional example.

### 1.1.5 Python

#### 1.1.5.1 Installation from the python package index (PyPi)

Windows / Linux / Mac
```
pip3 install fir1
```

under Windows it might be just `pip` for python3.

#### 1.1.5.2 Installation from source

Windows / Linux / Mac: make sure that you have swig and a C++ compiler installed. Then type:
```
python3 setup.py install
```

## 1.2 How to use it

### 1.2.1 cmake

Add to your `CMakeLists.txt` either
```
target_link_libraries(myexecutable fir)
```

for the dynamic library or
```
target_link_libraries(myexecutable fir_static)
```

for the statically linked library.

You can also use `find_package(fir)`.

### 1.2.2 Generating the FIR filter coefficients

Set the coefficients either with a C floating point array or with a text file containing the coefficients. The text file or the floating point array with the coefficients can easily be generated by Python or OCTAVE/MATLAB:

#### 1.2.2.1 Python

Use the `firwin` command to generate the coefficients:
```
# Sampling rate
fs = 1000
# bandstop between 45 and 55 Hz:
f1 = 45
f2 = 55
b = signal.firwin(999,[f1/fs*2,f2/fs*2])
```

For fixed point you need to scale up the coefficients, for example by 15 bits: b∗32768.

#### 1.2.2.2 octave/MATLAB:

```
octave:1> h=fir1(100,0.1);
```

which creates the coefficients of a lowpass filter with 100 taps and normalised cutoff 0.1 to Nyquist.

Again, for fixed point "h" needs to be scaled.

### 1.2.3 Initialisation

#### 1.2.3.1 C++ floating point FIR filter:

```
Fir1 fir("h.dat");
```

or import the coefficients as a const double array:
```
Fir1 fir(coefficients)
```

there is also an option to import a non-const array (for example generated with the ifft).

#### 1.2.3.2 C++ integer FIR filter:

```
Fir1fixed fir("h_fixed.dat",12);
```

where the coefficients have been scaled up by 2^12 and the filter will scale them down by this amount (with the help of a bitshift operation).

#### 1.2.3.3 JAVA:

```
Fir1 fir = new Fir1(coeff);
```

where `coeff` is an array of double precision coefficients and returns the fir filter class.

#### 1.2.3.4 Python

```
f = fir1.Fir1(coeff)
```

### 1.2.4 Realtime filtering

#### 1.2.4.1 C++ double: `double b = fir.filter(a);`

#### 1.2.4.2 C++ integer: `int b = fir.filter(a);`

#### 1.2.4.3 JAVA: `double b = fir.filter(a)`

#### 1.2.4.4 Python `b = f.filter(a)`

### 1.2.5 Destructor

#### 1.2.5.1 C++ `delete fir;`

#### 1.2.5.2 JAVA `fir.release();`

to release the underlying C++ class.

## 1.3 LMS algorithm

The least mean square algorithm adjusts the FIR coefficients w_k with the help of an error signal

```
w_k(t+1) = w_k(t) + learning_rate * buffer_k(t) * error(t)
```

using the function `lms_update(error)` while performing the filtering with `filter()`.

### 1.3.1 How to use the filter

- Construct the Fir filter with all coefficients set to zero: `Fir1(nCoeff)`
- Set the learning_rate with the method `setLearningRate(learning_rate)`.
- Define the signal1 to the FIR filter and use its standard `filter` method to filter it.
- Define your error which needs to be minimised: `error = signal2 - fir_filter_output`
- Feed the error back into the filter with the method `lms_update(error)`.

The `lmsdemo` in the demo directory makes this concept much clearer how to remove artefacts with this method.

The above plot shows the filter in action which removes 50Hz noise with the adaptive filter. Learning is very fast and the learning rate here is deliberately kept low to show how it works.

### 1.3.2 Stability

The FIR filter itself is stable but the error signal changes the filter coefficients which in turn change the error and so on. There is a rule of thumb that the learning rate should be less than the "tap power" of the input signal which is just the sum of all squared values held in the different taps:

```
learning_rate < 1/getTapInputPower()
```

That allows an adaptive learning rate which is called "normalised LMS". From my experiments that works in theory but in practise the realtime value of getTapInputPower() can make the algorithm easily unstable because it might suggest infinite learning rates and can fluctuate wildly. A better approach is to keep the learning rate constant and rather control the power of the input signal by, for example, normalising the input signal or limiting it.

See the demo below which removes 50Hz from an ECG which uses a normalised 50Hz signal which guarantees stability by design.

#### 1.3.3 JAVA/Python

The commands under JAVA and Python are identical to C++.

### 1.4 Demos

Demo programs are in the "demo" directory which show how to use the filters for both floating point and fixed point.

1. `firdemo` sends an impulse into the filter and you should see the impulse response at its output.
2. `fixeddemo` filters an example ECG with 50Hz noise. The coefficients are 12 bit and you can generate them either with OCTAVE/MATLAB or Python. The scripts are also provided.
3. `lmsdemo` filters out 50Hz noise from an ECG with the help of adaptive filtering by using the 50Hz powerline frequency as the input to the filter. This can be replaced by any reference artefact signal or signal which is correlated with the artefact.
4. JAVA has an `InstrumentedTest` which filters both a delta pulse and a step function.
5. filter_ecg.py performs the filtering of an ECG in python using the fir1 python module which in turn calls internally the C++ functions.

### 1.5 Unit tests

Under C++ just run `make test` or `ctest`.

The JAVA wrapper contains an instrumented test which you can run on your Android device.

### 1.6 Credits

This library has been adapted form Graeme Hattan's original C code.

Enjoy!

Bernd Porr & Graeme Hattan

## 2 Class Index

### 2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:



## 3 Class Documentation

### 3.1 Fir1 Class Reference

```
#include <Fir1.h>
```

**Public Member Functions**

- template<unsigned nTaps>
  Fir1 (const double(&_coefficients)[nTaps])
- Fir1 (std::vector< double > _coefficients)
- Fir1 (double ∗coefficients, unsigned number_of_taps)
- Fir1 (const char ∗coeffFile, unsigned number_of_taps=0)
- Fir1 (unsigned number_of_taps)
- ∼Fir1 ()
- double filter (double input)
- void lms_update (double error)
- void setLearningRate (double _mu)
- double getLearningRate ()
- void reset ()
- void zeroCoeff ()
- unsigned getTaps ()
- double getTapInputPower ()

### 3.1.1 Detailed Description

Finite impulse response filter. The precision is double. It takes as an input a file with coefficients or an double array.

### 3.1.2 Constructor & Destructor Documentation

#### 3.1.2.1 Fir1() [1/5] template<unsigned nTaps>

```
Fir1::Fir1 (
            const double(&) _coefficients[nTaps] )  [inline]
```

Coefficients as a const double array. Because the array is const the number of taps is identical to the length of the array.

**Parameters**

| | |
|---|---|
| *_coefficients* | A const double array with the impulse response. |

#### 3.1.2.2 Fir1() [2/5] Fir1::Fir1 (

```
            std::vector< double > _coefficients )  [inline]
```

Coefficients as a C++ vector

**Parameters**

| | |
|---|---|
| *_coefficients* | is a Vector of doubles. |

#### 3.1.2.3 Fir1() [3/5] 

```
Fir1::Fir1 (
    double * coefficients,
    unsigned number_of_taps )
```

Coefficients as a (non-constant-) double array where the length needs to be specified.

**Parameters**

| | |
|---|---|
| *coefficients* | Coefficients as double array. |
| *number_of_taps* | Number of taps (needs to match the number of coefficients |

#### 3.1.2.4 Fir1() [4/5]

```
Fir1::Fir1 (
    const char * coeffFile,
    unsigned number_of_taps = 0 )
```

Coefficients as a text file (for example from Python) The number of taps is automatically detected when the taps are kept zero.

**Parameters**

| | |
|---|---|
| *coeffFile* | Patht to textfile where every line contains one coefficient |
| *number_of_taps* | Number of taps (0 = autodetect) |

#### 3.1.2.5 Fir1() [5/5]

```
Fir1::Fir1 (
    unsigned number_of_taps )
```

Inits all coefficients and the buffer to zero This is useful for adaptive filters where we start with zero valued coefficients.

#### 3.1.2.6 ∼Fir1()

```
Fir1::∼Fir1 ( )
```

Releases the coefficients and buffer.

### 3.1.3 Member Function Documentation

#### 3.1.3.1 filter()

```
double Fir1::filter (
    double input ) [inline]
```

The actual filter function operation: it receives one sample and returns one sample.

**Parameters**

| | |
|---|---|
| *input* | The input sample. |

#### 3.1.3.2 getLearningRate() `double Fir1::getLearningRate ( ) [inline]`

Getting the learning rate for the adaptive filter.

#### 3.1.3.3 getTapInputPower() `double Fir1::getTapInputPower ( ) [inline]`

Returns the power of the of the buffer content: sum_k buffer[k]^2 which is needed to implement a normalised LMS algorithm.

#### 3.1.3.4 getTaps() `unsigned Fir1::getTaps ( ) [inline]`

Returns the number of taps.

#### 3.1.3.5 lms_update() `void Fir1::lms_update ( double error ) [inline]`

LMS adaptive filter weight update: Every filter coefficient is updated with: w_k(n+1) = w_k(n) + learning_rate ∗ buffer_k(n) ∗ error(n)

**Parameters**

| | |
|---|---|
| *error* | Is the term error(n), the error which adjusts the FIR conefficients. |

#### 3.1.3.6 reset() `void Fir1::reset ( )`

Resets the buffer (but not the coefficients)

#### 3.1.3.7 setLearningRate() `void Fir1::setLearningRate ( double _mu ) [inline]`

Setting the learning rate for the adaptive filter.

**Parameters**

| | |
|---|---|
| *_mu* | The learning rate (i.e. rate of the change by the error signal) |

#### 3.1.3.8 zeroCoeff() `void Fir1::zeroCoeff ( )`

Sets all coefficients to zero

The documentation for this class was generated from the following file:

- Fir1.h

# Index