# fir1

Generated by Doxygen 1.9.1

# 1 FIR1

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

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

## 1.1 Installation

### 1.1.1 Packages for Ubuntu LTS

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 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.
For example for Visual Studio 2019 you write:
```

cmake -G "Visual Studio 16 2019" -A x64 .
```
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.
### Python
#### Installation from the python package index (PyPi)
Windows / Linux / Mac
```

pip install fir1
```
under Windows it might be just 'pip' for python3.
#### Installation from source
Windows / Linux / Mac: make sure that you have swig and a C++ compiler installed. Then type:
```

python setup.py install
```
## How to use it
### 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)'.
### 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:
#### Python
Use the 'firwin' command to generate the coefficients:
```

## 1.2 Sampling rate

fs = 1000

## 1.3 bandstop between 45 and 55 Hz:

f1 = 45 f2 = 55 b = signal.firwin(999,[f1/fs∗2,f2/fs∗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.
### Initialisation
#### 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) and using std::vector. You can also
create a moving average filter by initialising all coefficients
with a constant value:
```

Fir1 moving_average(100,1.0/100);

```
#### Python
```

f = fir1.Fir1(coeff)

```
### Realtime filtering
#### C++ double:
```

double b = fir.filter(a);

```
#### Python
```

b = f.filter(a)

```
### Utility methods
These functions are the same in C++ and Python:
+ `getTaps()` returns the length of the FIR filter kernel.
+ `reset()` sets all delay lines to zero.
+ `zeroCoeff()` sets all coefficients to zero.
Retreiving the coefficients/kernel from the FIR filter is different depending on the
language used:
#### C++
+ `void getCoeff(double* target, unsigned length) const` copies the FIR kernel into the
given C array of `double`s with length `length`.
   If `length` exceeds the length of the filter kernel, the result is zero-padded to fill
the given array.
   If `length` is smaller than the filter kernel, a `std::out_of_range` exception is thrown.

+ `std::vector<double> getCoeffVector() const` returns a copy of the filter kernel.
#### Python
+ `getCoeff(n : int) -> numpy.array` as per the C++ method, following the zero-padding
and exception-throwing behaviour of the C++. The returned array will have `n` elements.
+ `getCoeff() -> numpy.array` additional to the C++ methods, this returns an numpy array
which is a copy of the filter kernel. This is probably the default use case in Python.
## LMS algorithm
![alt tag](fir_lms.png)
The least mean square algorithm adjusts the FIR coefficients h_m
with the help of an error signal e(n):
```

h_m(n+1) = h_m(n) + learning_rate ∗ h_m(n) ∗ e(n)

```
using the function `lms_update(e)` while performing
the filtering with `filter()`.
### How to use the LMS filter
- Construct the Fir filter with all coefficients set to zero: `Fir1(nCoeff)`
- Set the learning_rate with the method `setLearningRate(learning_rate)`.
- Provide the input signal `x` to the FIR filter and use its standard `filter` method to filter it.
- Define your error which needs to be minimised: `e = d - y`
- Feed the error back into the filter with the method `lms_update(e)`.
The `lmsdemo` in the demo directory makes this concept much clearer how to remove
artefacts with this method.
![alt tag](lms.png)
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.
### 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.

#### 1.3.0.1 Python

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

### 1.3.1 Demos

Demo programs are in the "demo" directory which show how to use the filter.

1. `firdemo` sends an impulse into the filter and you should see the impulse response at its output.
2. `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.
3. filter_ecg.py performs the filtering of an ECG in python using the fir1 python module which in turn calls internally the C++ functions.

### 1.3.2 C++ documentation

The doxygen generated documentation can be found here:

- Online: `http://berndporr.github.io/fir1/index.html`
- PDF: `https://github.com/berndporr/fir1/tree/master/docs/pdf`

### 1.3.3 Unit tests

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

### 1.3.4 Credits

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

Enjoy!

Bernd Porr & Nick Bailey

# 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 (const double *coefficients, const unsigned number_of_taps)
- Fir1 (const char *coeffFile, unsigned number_of_taps=0)
- Fir1 (unsigned number_of_taps, double value=0)
- ~Fir1 ()
- double filter (double input)
- void lms_update (double error)
- void setLearningRate (double _mu)
- double getLearningRate ()
- void reset ()
- void zeroCoeff ()
- void getCoeff (double *coeff_data, unsigned number_of_taps) const
- void setCoeff (const double *coeff_data, const unsigned number_of_taps)

  *Externally sets the coefficient array. This is useful when the actually running filter is at a different place as where the updating filter is employed.*
- std::vector< double > getCoeffVector () const
- 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 (
    const double * coefficients,
    const 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,
    double value = 0 )
```

Inits all coefficients and the buffer to a constant value. This is useful for adaptive filters where we start with zero valued coefficients or moving average filters with value = 1.0/number_of_taps.

#### 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 getCoeff() `void Fir1::getCoeff ( double * coeff_data, unsigned number_of_taps ) const`

Copies the current filter coefficients into a provided array. Useful after an adaptive filter has been trained to query the result of its training.

**Parameters**

| | |
|---|---|
| *coeff_data* | target where coefficients are copied |
| *number_of_taps* | number of doubles to be copied |

**Exceptions**

| | |
|---|---|
| *std::out_of_range* | number_of_taps is less the actual number of taps. |

#### 3.1.3.3 getCoeffVector() `std::vector<double> Fir1::getCoeffVector ( ) const [inline]`

Returns the coefficients as a vector

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

Getting the learning rate for the adaptive filter.

#### 3.1.3.5 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.

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

Returns the number of taps.

#### 3.1.3.7 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. |

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

Resets the buffer (but not the coefficients)

#### 3.1.3.9 setCoeff() `void Fir1::setCoeff (`

```
const double * coeff_data,
const unsigned number_of_taps )
```

Externally sets the coefficient array. This is useful when the actually running filter is at a different place as where the updating filter is employed.

**Parameters**

| | |
|---|---|
| *coeff_data* | New coefficients to set. |
| *number_of_taps* | Number of taps in the coefficient array. If this is not equal to the number of taps used in this filter, a runtime error is thrown. |

#### 3.1.3.10 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) |

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

Sets all coefficients to zero

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

- Fir1.h

# Index