Marsyas User Manual

For version 0.2

Music **A**nalysis **R**etrieval and **SY**nthesis for **A**udio **S**ignals

George Tzanetakis

Table of Contents

1 Introduction

MARSYAS (**M**usic **A**nalysis **R**etrieval and **S**ynthesis for **A**udio **S**ignals) is a free software framework for audio analysis, synthesis and retrieval written by George Tzanetakis. Please direct any questions/comments about Marsyas to (gtzan@cs.uvic.ca).

The major underlying theme under the design of Marsyas has been to provide an efficient and extensible framework for building audio analysis (and synthesis) applications with specific emphasis on Music Information Retrieval (MIR). A variety of building blocks for performing common audio tasks are provided. Some representative examples are: soundfile IO, audio IO, signal processing and machine learning modules. These blocks can be combined into data flow networks that can be modified and controlled dynamically while they process data in soft real-time.

Another goal has been to accomodate two different types of users: naive and expert (of course in many cases the same person can operate in both modes). Naive users are able to construct networks of primitive objects and experiment with them through the use of controls. They can interact with the system through the use of graphical user interfaces or high level scripts without actually having to compile any code. Marsyas provides a high-level of control at runtime without sacrificing performance. Expert users can create new primitive objects and create more complex applications by writing code and compiling. These two modes of operation will become clearer in the following sections of the manual. As with any piece of software the holy grail is to provide maximum automatic support for the tasks that can be automated while retaining expressiveness and the ability to program complex systems for the particular domain of interest.

This framework has been created mainly to support George Tzanetakis's research mainly in the emerging area of Music Information Retrieval (MIR). Anyone who finds the framework useful is welcome to use it and contribute to it. If you would like specific features to be developed feel free to contact the author or support the project by donnating money through the sourceforge page.

1.1 History

Work on Marsyas started in 1998 during my second year of graduate studies in Computer Science at Princeton University under the supervision of Dr. Perry Cook. The main motivation behind the design and development of the toolkit was and still is to personally code the majority of the tools I need for my research in order to have understanding and control of how they work. Marsyas has been used for every paper I have published since that time. I continued to add code to Marsyas until 2000 when it was clear that certain major design decisions needed to be revised. That made me start a major rewrite/redesign of the framework and that was the start of the first 'real" Marsyas version which was numbered 0.1. Soon after Sourceforge was used to host Marsyas. Version 0.1 is still widely used by a variety of academic groups and industry around the world.

In 2002 while being a PostDoctoral Fellow at Carnegie Mellon University working with Roger Dannenberg I decided to start porting algorithms from the Synthesis Toolkit (STK) by Perry Cook and Gary Scavone into Marsyas. This effort as well as many interesting conversations with Roger made me rethink the design used by Marsyas. The result was

to move to a dataflow model of audio computation with general matrices instead of 1-D arrays as data and an Open Sound Control (OSC) inspired hierarchical messaging system used to control the dataflow network. Marsyas 0.2 is now almost to the point of supporting the full functionality of Marsyas 0.1. Hopefully the writing of this manual will help users migrate from version 0.1. If you are a user that has done work in 0.1 it should be relatively straightforward to figure out how to recode your algorithms in version 0.2. Also if you have code in 0.1 that you would like help porting in 0.2 I would be more than happy to help - just drop me an email.

1.2 Context and Related Work

There is a lot of interesting related work and inspiration behind the design of this framework. As the goal of this introduction is to provide a quick overview of the system I will just briefly mention some of the key ideas that strongly influenced the design of the system without getting into details. Probably the most central inspiration has been the huge legacy of computer music synthesis languages such as the Music V family, Csound etc. More recent work that has been influential to the design of the system has been the architecture of the Synthesis Toolkit (STK) and the hierarchical control naming scheme of Open Sound Control (OSC). Other influences include the use of Design Patterns for creating the object oriented architecture of the system, kernel stream architectures as well as data flow simulation software systems such as SimuLink by Matlab and the FilterGraph by Microsoft. Finally many of the ideas of functional programming such as the clear separation of mutable and immutable data and the use of composition to build complicated systems have been another major source of inspiration.

There is a plethora of programming languages, frameworks and environments for the analysis and synthesis of audio signals. The processing of audio signals requires extensive numerical calculations over large amounts of data especially when real-time performance is desired. Therefore efficiency has always been a major concern in the design of audio analysis and synthesis systems. Dataflow programming is based on the idea of expressing computation as a network of processing nodes/components connected by a number of communication channels/arcs. Computer Music is possibly one of the most successful application areas for the dataflow programming paradigm. The origins of this idea can possibly be traced to the physical re-wiring (patching) employed for changing sound characteristics in early modular analog synthesizers. From the pioneering work on unit generators in the Music N family of language to currently popular visual programming environments such as Max/Msp and Pure Data (PD), the idea of patching components to build systems is familiar to most computer music practitioners.

Expressing audio processing systems as dataflow networks has several advantages. The programmer can provide a declarative specification of what needs to be computed without having to worry about the low level implementation details. The resulting code can be very efficient and have low memory requirements as data just “flows” through the network without having complicated dependencies. In addition, dataflow approaches are particularly suited for visual programming. One of the initial motivation for dataflow ideas was the exploitation of parallel hardware and therefore dataflow systems are particularly suited for parallel and distributed computation.

Despite these advantages, dataflow programming has not managed to become part of mainstream programming and replace existing imperative, object-oriented and functional languages. Some of the traditional criticisms aimed at dataflow programming include: the difficulty of expressing complicated control information, the restrictions on using assignment and global state information, the difficulty of expressing iteration and complicated data structures, and the challenge of synchronization.

There are two main ways that existing successful dataflow systems overcome these limitations. The first is to embed dataflow ideas into an existing programming language. This is called coarse-grained dataflow in contrast to fine-grained dataflow where the entire computation is expressed as a flow graph. With coarse-grained dataflow, complicated data structures, iteration, and state information are handled in the host language while using dataflow for structured modularity. The second way is to work on a domain whose nature and specific constraints are a good fit to a dataflow approach. For example, audio and multimedia processing typically deals with fixed-rate calculation of large buffers of numerical data.

Computer music has been one of the most successful cases of dataflow applications even though the academic dataflow community doesn't seem to be particularly aware of this fact. Existing audio processing dataflow frameworks have difficulty handling spectral and filterbank data in an conceptually clear manner. Another problem is the restriction of using fixed buffer sizes and therefore fixed audio and control rates. Both of these limitations can be traced to the restricted semantics of patching as well as the need to explicitly specify connections. Implicit Patching the technique used in Marsyas-0.2 is an attempt to overcome these problems while maintaining the advantages of dataflow computation.

2 Installation

2.1 Download

2.1.1 Stable(-ish) Version

(*Important Note: Marsyas does not use CVS and therefore the CVS repository is out of date*)

Marsyas is hosted at SourceForge:

http://marsyas.sourceforge.net/

http://www.sourceforge.net/projects/marsyas

Marsyas is open source software and is distributed as a tar ball (something like marsyas-0.2.8.tar.gz).

To extract the source file type:

```
>tar -zxvf marsyas-0.2.8.tar.gz
```

This will create a subdirectory of the current directory called marsyas-0.2.8 that contains all the source files needed for compiling Marsyas.

2.1.2 Development Version

For the brave, the latest (possibly unstable!) version can be obtained from the subversion repository stored at the sourceforge website. To check out a working copy do:

```
>svn co https://svn.sourceforge.net/svnroot/marsyas/trunk marsyas-0.2.8
```

You can replace marsyas-0.2.8 with any directory you want. The version/release (version 0.2 release 8) is independently assigned from subversion revisions and the latest can be found by checking the sourceforge website.

2.2 Linux Installation

Marsyas is mainly developed under Linux so installing under Linux is quite straightforward and can be done using the standard autoconf installation steps where `>` is whatever command-line prompt you have on your system.

```
> ./configure
> make
> make install                  (as root user)
```

By default Marsyas assumes that in Linux systems the ALSA sound system and corresponding library and headers are installed. Most new Linux distribution are using ALSA.

In addition there are several configure options that can be used to enable/disable assertions, enable/disable debugging and include optional packages such as support for reading mp3s. See Section 2.6 [Configure Options], page 5.

2.3 Mac OSX Installation

Installation under OS X is almost identical to Linux. The developer tools are not installed by default so you will need to install them. You can download XCode from the Apple Developer website. You can check whether they are installed or not by checking that you can run gcc on a terminal. Once gcc is installed then you can compile using the standard autoconf procedure:

```
> ./configure
> make
> make install                          (as root user)
```

In addition there are several configure options that can be used to enable/disable assertions, enable/disable debugging and include optional packages such as support for reading mp3s. See Section 2.6 [Configure Options], page 5.

2.4 Windows (cygwin)

Installation under Windows using the cygwin environment and gcc is similar to Linux. The most recent version of Marsyas 0.2 use RtAudio for audio playback under Cygwin. In order to compile RtAudio you will need to have the DirectX SDk installed. The standard autoconf installation procedure is used:

```
> ./configure
> make
> make install                          (as root user)
```

In general, cygwin is not supported as well as Linux and OS X.

2.5 Windows (Visual Studio)

Installation under Visual Studio is still under construction and not the full functionality of Marsyas 0.2 is supported. Not all the executables are supported. The main distribution contains two subdirectories named marsyasVisualStudio2003, marsyasVisualStudio2005 that contain the necessary project solution files for compiling Marsyas under Visual Studio. I have only tested this using Visual Studio 7.0 under Windows XP. Older version of Visual Studio might not work. Support of the Windows port is not as good as Linux and OS X.

2.6 Configure Options

Marsyas can be customized using various configuration options. In autoconf systems (Linux, OSX, Cygwin) this done in the standard way through the .configure script. For example to compile Marsyas with assertions enabled and with mp3 support through libmad one would do:

```
> ./configure --enable-assert --enable-mad
> make
> make install                          (as root user)
```

The list of available options can be viewed by:

```
> ./configure --help
```

The following options are supported:

- –**enable-assert** turns assertions on (small performance penatly)
- –**enable-debug** compiles Marsyas in debug mode generating the necessary files for gdb (large performance penalty)
- –**enable-mad** enables support for reading mp3 files using libmad (which must be installed)
- –**enable-distributed** compiles code for distributed audio feature extraction (experimental)
- –**enable-readline** realine support for the Marsyas Scripting Language (msl)
- –**enable-oss** use the OSS sound system

A set of optional graphical user interfaces written in QT 4 are also included in the distribution in the qt4GUIs subdirectory. If Qt4 is installed on your machine and you have installed Marsyas on your system you can compile the various GUIs by doing (replace MarPlayer with the specific GUI you want compiled) (these GUIs have only been tested under Linux and are currently under major development so use at your own risk):

```
> cd qt4GUIs/MarPlayer
> qmake
> make
```

A frequent variation (if you don't have root priviledges) is to install Marsyas in your home directory (replace /home/gtzan with the appropriate path for your home directory). This can be accomplished by doing:

```
> ./configure --prefix=/home/gtzan
> make
> make install
```

2.7 Structure of distribution

Marsyas is primarily targeted to researchers and software developers who want to build new systems and applications using existing building blocks. Therefore familiarity with the directory structure of the Marsyas distribution is important for any serious work with the framework.

The main marsyas directory consists of the following important files:

- **INSTALL, COPYING, THANKS, README, AUTHORS, TODO:** text files with important information.
- **ChangeLog:** the change log is not maintained as consistently as it should be but it still provides useful information about the evolution of the software.
- **configure.in, Makefile.am:** the main files edited by the user/programmer that are required for the autotools. You will only need to edit these if you are adding new subdirectories or configuration options to the distribution.

In addition there are the following subdirectories:

- **marsyas:** the main directory containing all the important source code of Marsyas. The source files in this subdirectory are compiled into a static library that other programs can use to access Marsyas functionality.
- **src:** this subdirectory contains several sample executables that do various interesting things using the Marsyas software framework. Some of them are intended to be used as actual research tools others are more demonstration. All new Marsyas users should browse at least some of the source code here.
- **doc:** contains both the user manual (which you are currently reading) as well as the source code documentation that is generated using doxygen. To regenerate the manual in pdf or html type (in the doc subdirectory)

  ```
  >make pdf
  >make html
  ```

- **qt4GUIs:** provides GUI interfaces using QT4. These will not work with earlier versions of QT. You must have QT4 installed in order to use or compile these applications. A README file is supplied with each one that has instructions on how to compile and run it.
- **config:** configuration files used by autotools.
- **distributed:** Experimental Marsyas classes for distributed processing.
- **marsyasMATLAB:** User MATLAB scripts (mfiles).
- **marsyasVisualStudio2003:** Project and Solution files for Visual Studio 2003
- **marsyasVisualStudio2005:** Project and Solution files for Visual Studio 2005

3 Tools

The main goal of Marsyas is to provide an extensible framework that can be used to quickly design and experiment with audio analysis and synthesis applications. The tools provided with the distribution, although useful, are only representative examples of what can be achieved using the provided components. Marsyas is an extensible framework for building applications, so the primary purpose of these examples is to provide source code of working applications.

3.1 Collections and input files

Many Marsyas tools can operate on individual soundfiles or collections of soundfiles. A collection is a simple text file which contain lists of soundfiles.

3.1.1 Creating collections manually

A simple way to create a collection is the unix ls command. For example:

```
> ls /home/gtzan/data/sound/reggae/*.wav > reggae.mf
```

`reggae.mf` will look like this:

```
/home/gtzan/data/sound/reggae/foo.wav
/home/gtzan/data/sound/reggae/bar.wav
```

Any text editor can be used to create collection files. The only constraint is that the name of the collections file must have a `.mf` extension such as `reggae.mf`. In addition, any line starting with the `#` character is ignored. For Windows Visual Studio, change the slash character separating directories appropriately.

3.1.2 `mkcollection`

`mkcollection` is a simple utility for creating collection files. To create a collection of all the audio files residing in a directory the following command can be used:

```
> mkcollection reggae.mf /home/gtzan/data/sound/
```

All the soundfiles residing in that directory or any subdirectories will be added to the collection. `mkcollection` only will add files with `.wav` and `.au` extensions but does not check that they are valid soundfiles. In general collection files should contain soundfiles with the same sampling rate as Marsyas does not perform automatic sampling conversion. The exception to this rule is collection that mix files at 22050Hz and 44100Hz sampling rates, in which case the 44100Hz files are downsampled to 22050Hz. No implicit downsampling is performed to collections that contain only 44100Hz files.

3.2 Simple Soundfile Interaction

3.2.1 sfinfo

sfinfo is a simple command-line utility for displaying information about a soundfile. It is also a simple example of how printing out the controls can show information like channels, sampling rate etc.

```
> sfinfo foo.wav
```

3.2.2 sfplay

sfplay is a flexible command-line soundfile player that allows playback of multiple soundfiles in various formats with either real-time audio output or soundfile output. The following two example show two extremes of using of sfplay: simple playback of foo.wav and playing 3.2 seconds (-l) clips starting at 10.0 seconds (-s) into the file and repeating the clips for 2.5 times (-r) writing the output to output.wav (-f) at half volume (-g) playing each file in the collection reggae.mf. The last command stores the MarSystem dataflow network used in sfplay as a plugin in playback.mpl. The plugin is essentially a textual description of the created network. Because MarSystems can be created at run-time the network can be loaded in a sfplugin which is a generic executable that flows audio data through any particular network. Running sfplugin -p playback.mpl bar.wav will play using the created plugin the file bar.wav. It is important to note that although both sfplay and sfplugin have the same behavior in this case they achieve it very different. The main difference is that in sfplay the network is created at compile time wheras in sfplugin the network is created at run time.

```
> sfplay foo.wav
> sfplay -s 10.0 -l 3.2 -r 2.5 -g 0.5 foo.wav bar.au -f output.wav
> sfplay -l 3.0 reggae.mf
> sfplay foo.wav -p playback.mpl
```

3.3 Feature Extraction

3.3.1 pitchextract

pitchextract is used to extract the fundamental frequency contour from monophonic audio signals. A simple sinusoidal playback is provided for playback of the resulting contour.

3.3.2 extract

extract is a single-file executable for feature extraction. It can be used as part of external systems for feature extraction therefore it outputs the results in a simple tab-separated text file. For more serious feature extraction over multiple files check bextract which is what I use most of the time. It also serves as an example of a network of MarSystems with relatively complicated structure. The following commands extract a single vector of features based on the first 30 seconds of the provided soundfile. By default the feature extractor is based on extracting features based on the magnitude of the Short Time Fourier Transform

(STFT) (i.e means and variances of Spectral Centroid, Rolloff, Flux). The second command extracts the means and variances of Mel-Frequency Cepstral Coefficients.

```
> extract foo.wav
> extract -e SVMFCC foo.wav
```

3.3.3 bextract

bextract is one of the most powerful executables provided by Marsyas. It can be used for complete feature extraction and classification experiments with multiple files. It serves as a canonical example of how audio analysis algorithms can be expressed in the framework.

Suppose that you want to build a real-time music/speech descriminator based on a collection of music files named music.mf and a collection of speech files named speech.mf. These collections can either be created manually or using the mkcollection utility. The following commandline will extract means and variances of Mel-Frequency cepstral coefficients (MFCC) over a texture window of 1 sec. The results are stored in a wekaOut.arff which is a text file storing the feature values that can be used in the Weka machine learning environment for experimentation with different classifiers. At the same time that the features are extracted, a simple Gaussian classifier is trained and when feature extraction is completed the whole network of feature extraction and classification is stored and can be used for real-time audio classification directly as a Marsyas plugin. The plugin makes a classification decision every 20ms but aggregates the results by majority voting to display output approximately every 1 second. The whole network is stored in musp_classify.mpl which is loaded into sfplugin and a new file named new.wav is passed through. The screen output shows the classification results and confidence.

Users familiar with marsyas 0.1 will notice that currently the machine learning part of marsyas 0.2 is not as sophisticated as the one of 0.1. For example there is no evaluate executable for performing cross-validation experiments and the only classifier currently implemented is a simple multidimensional Gaussian classifiar. For my own research I have been increasingly using Weka for all the machine learning experiments so porting this functionality to the new version is not a high priority. On the other hand I have a clear notion of how they can be integrated and most of the necessary components and APIs are already in place. Eventually I would like to port most of Weka into Marsyas but it will be some time until that happens.

```
> bextract -e STFT music.mf speech.mf -w wekaOut.arff -p musp_classify.mpl
> sfplugin -p musp_classify.mpl new.wav
```

Feature extractors that start with SV produce one value for each value and can be used for non-realtime classification such as genre classification. The following command can be used to generate a weka file for genre classification.

```
> bextract -e SVSTFT classical.mf jazz.mf rock.mf -w genre.arff
```

Currently no classifier is generated for the SV feature extractors but it's only a matter of time before this feature is added. The generated file genre.arff can the be loaded into Weka where classification experiments can be conducted.

3.4 Synthesis

3.4.1 phasevocoder

phasevocoder is probably the most powerful and canonical example of sound synthesis provided currently by Marsyas. It is based on the phasevocoder implementation described by F.R.Moore in his book "Elements of Computer Music". It is broken into individual MarSystems in a modular way and can be used for real-time pitch-shifting and time-scaling.

```
> phasevocoder -p 1.4 -s 100
```

3.4.2 sftransform

sftransform is an example of having a doubly nested network with two FFT/inverse FFT identity transformations. It's not particularly useful but show how to nested networks can be created.

3.5 Marsystem Interaction

3.5.1 sfplugin

sfplugin is the universal executable. Any network of Marsystems stored as a plugin can be loaded at run-time and sound can flow through the network. The following example with appropriate plugins will peform playback of foo.wav and playback with real time music speech classification of foo.wav.

```
> sfplugin -p plugins/playback.mpl foo.wav
> sfplugin -p musp_classify.mpl foo.wav
```

3.5.2 msl

One of the most useful and powerful characteristics of Marsyas is the ability to create and combine MarSystems at run time. msl (marsyas scripting language) is a simple interpreter that can be used to create dataflow networks, adjust controls, and run sound through the network. It's used as a backend for user interfaces therefore it has limited (or more accurately non-existent) editing functionality. The current syntax is being revised so currently it's more a proof-of-concept. Here is an example of creating a simple network in msl and playing a sound file:

```
>msl
[ msl ] create Series playbacknet
[ msl ] create SoundFileSource src
[ msl ] create Gain g
[ msl ] create AudioSink dest
[ msl ] add src  > playbacknet
[ msl ] add g    > playbacknet
[ msl ] add dest > playbacknet
[ msl ] updctrl playbacknet SoundFileSource/src/string/filename technomusic.au
[ msl ] run playbacknet
```

The important thing to notice is that both the creation of MarSystems and their assembly into networks can be done at run-time without having to recompile any code. If anyone would like to pick a project to do for Marsyas it would be to use the GNU readline utility for it's commandline editing capabilities and try to come up with some alternative syntax (I have some ideas in that direction).

4 Architecture

In order to fully take advantage of the capabilities of Marsyas it is important to understand how it works internally. The architecture of Marsyas reflects an underlying dataflow model that we have found useful in implementing real and non-real time audio analysis and synthesis systems. In marsyas 0.2 a lot of things can be accomplished by assembling complex networks of basic building blocks called MarSystems that process data. This is the so called "Black-Box" functionality of the framework. In addition the programmer can also write directly her/his own building blocks directly in C++ following a certain API and coding conventions offering the so called "White-Box" functionality. The next two sections describe building networks and writing new MarSystems respectively.

4.1 Architecture overview

4.1.1 Building MarSystems

The basic idea behind the design of Marsyas is that any audio analysis/synthesis computation can be expressed as some type of processing object, which we call MarSystem, that reads data from an input slice of floating point numbers, performs some computation/transformation based on data, and writes the results to another slice of floating point numbers. Networks of MarSystems can be combined and encapsulated as one MarSystem.

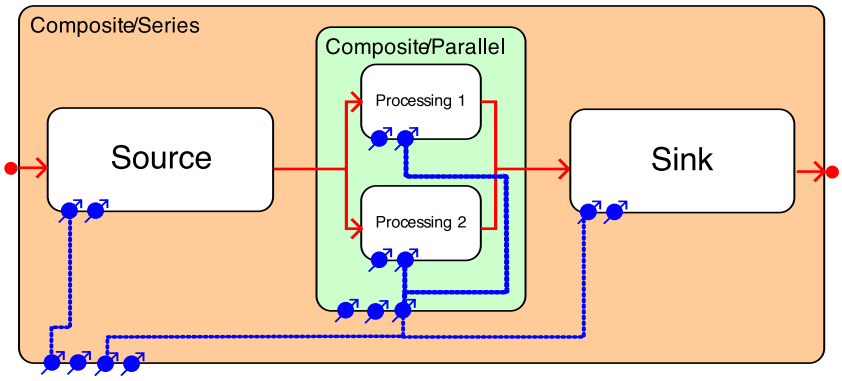
For example consider an audio processing series of computations consisting of reading samples from a soundfile, performing an short-time fourier transform (STFT) to calculate the complex spectrum, performing an inverse STFT to convert back from the frequency domain to time domain, then applying a gain to the amplitude of the samples and writing the result to a soundfile.

As is very frequently the case with audio processing networks objects the input of each stage is the output of the previous stage. This way of assembling MarSystems is called a Series composite. Once a Series Composite is formed it can basically be used as one MarSystem that does the whole thing. A figure showing a block diagram-like presentation of this network is shown in the figure bellow.

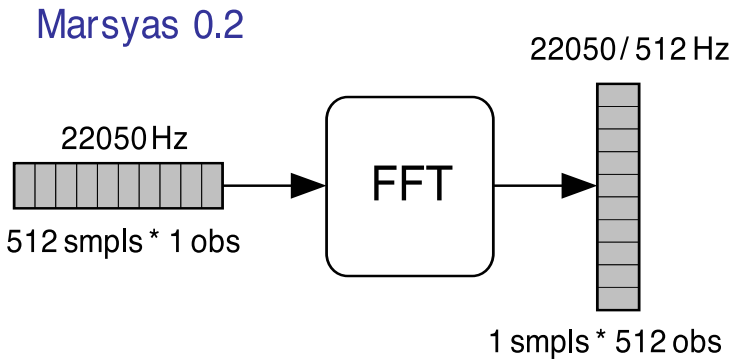
The main method that each MarSystem must support is **process** which takes two arguments both arrays of floating point numbers used to represent slices (matrices where one dimension is samples in time and the other is observations which are interpreted as happening at the same time). When the **process** method is called it reads data from the input slice, performs some computation/transformation and writes the results to the output slice. Both slices have to be preallocated when process is called. One of the main advantages of Marsyas is that a lot of the necessary buffer allocation/reallocation and memory management happens behind the scene without the programmer having to do anything explicitly.

4.1.2 Dataflow model

Marsyas follows a dataflow model of audio computation.



Marsyas uses general matrices instead of 1-D arrays. This allows slices to be semantically correct.



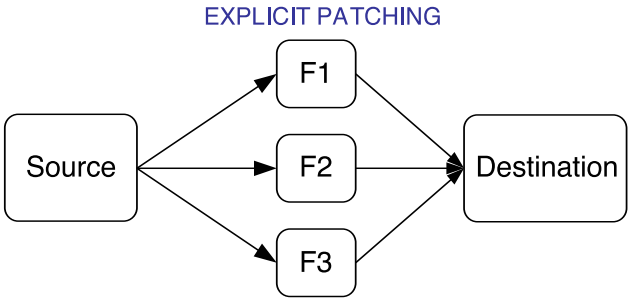
4.2 Implicit patching

4.2.1 Implicit patching vs. explicit patching

Many audio analysis programs require the user to explicitly (manually) connect every processing block,

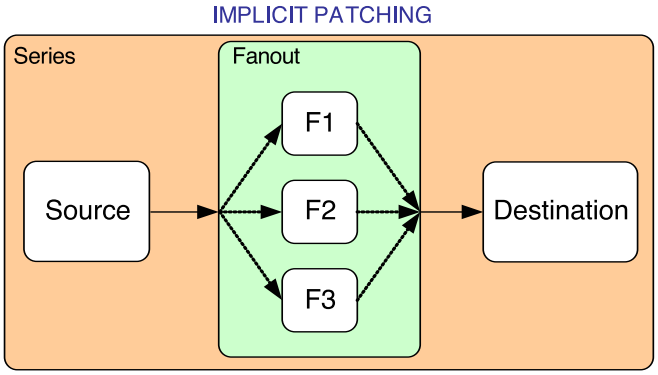
```
# EXPLICIT PATCHING: block definitions
source, F1, F2, F3, destination;
# connect the in/out ports of the blocks
connect(source, F1);
connect(source, F2);
connect(source, F3);
connect(F1, destination);
connect(F2, destination);
```

```
connect(F3, destination);
```



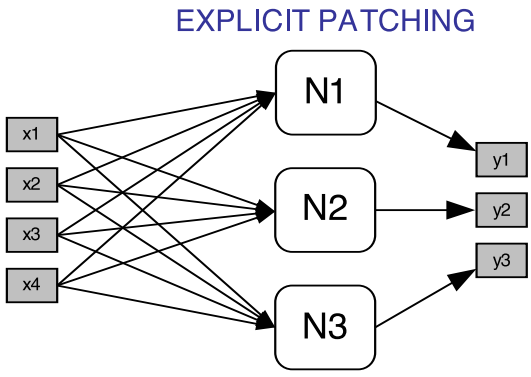
Marsyas uses *implicit patching*: connections are made automagically when blocks are created,

```
# IMPLICIT PATCHING
source, F1, F2, F3, destination;
Fanout(F1, F2, F3);
Series(source, Fanout, destination);
```



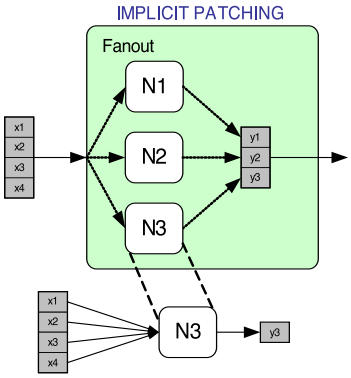
4.2.2 Implicit patching advantages

Creating a neural network with explicit patching soon becomes a mess,

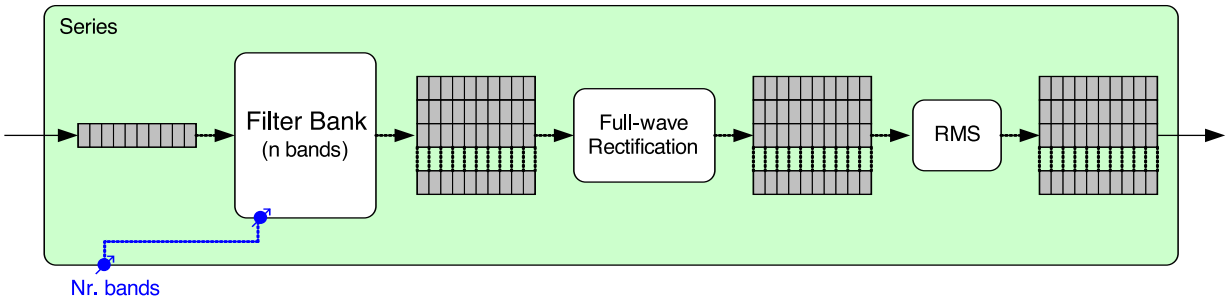


With implicit patching, this is much more manageable.

```
# IMPLICIT PATCHING
fanoutLayer1(ANN_Node11, ..., ANN_Node1N);
...
fanoutLayerM(ANN_NodeM1, ..., ANN_NodeMN);
ANN_Series(fanoutLayer1, ..., fanoutLayerM);
```

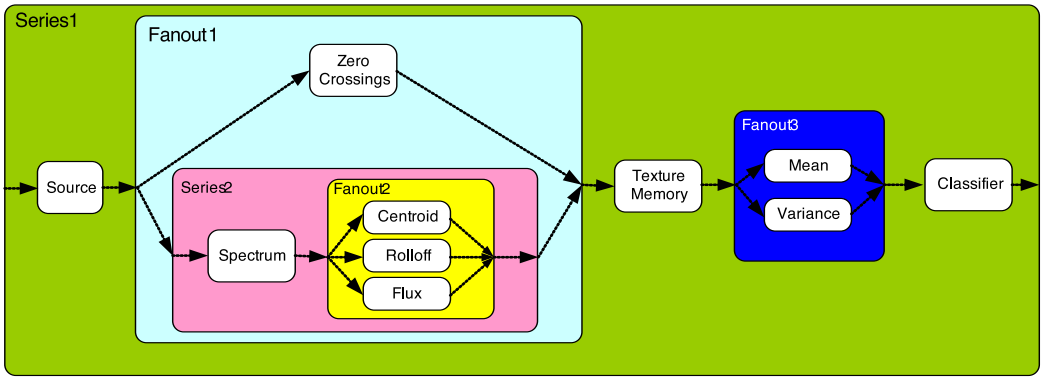


Implicit patching can automagically adjust the connections without requiring any code recompiliation. For example, we can change the number of bands in a filter bank without changing any code.



4.2.3 Patching example of Feature extraction

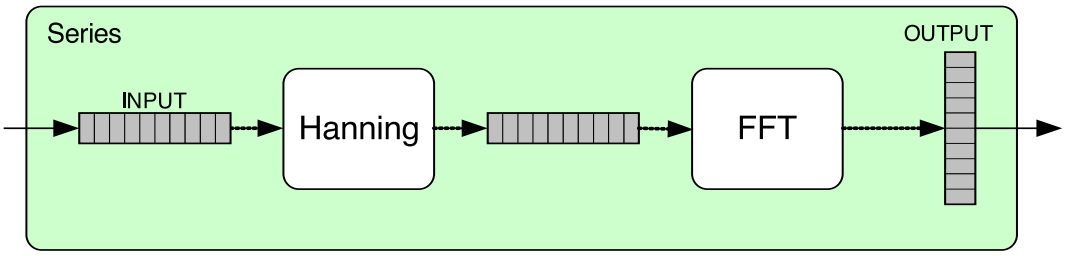
Suppose we wish to create a typical feature extraction program:



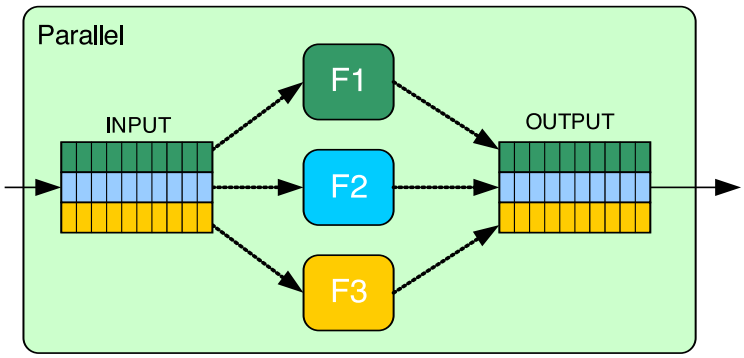

```
MarSystemManager mng;
MarSystem* Series1 = mng.create("Series", "Series1");
MarSystem* Fanout1 = mng.create("Fanout", "Fanout1");
MarSystem* Series2 = mng.create("Series", "Series2");
MarSystem* Fanout2 = mng.create("Fanout", "Fanout2");
MarSystem* Fanout3 = mng.create("Fanout", "Fanout3");
Fanout3->addMarSystem(mng.create("Mean", "Mean"));
Fanout3->addMarSystem(mng.create("Variance", "Variance"));
Fanout2->addMarSystem(mng.create("Centroid", "Centroid"));
Fanout2->addMarSystem(mng.create("RollOff", "Rolloff"));
Fanout2->addMarSystem(mng.create("Flux", "Flux");
Series2->addMarSystem(mng.create("Spectrum, "Spectrum");
Series2->addMarSystem(Fanout2);
Fanout1->addMarSystem(mng.create("ZeroCrossings", "ZeroCrossings");
Fanout1->addMarSystem(Series2);
Series1->addMarSystem(mng.create("SoundFileSource", "Source"));
Series1->addMarSystem(Fanout1);
Series1->addMarSystem(mng.create("Memory", "TextureMemory"));
Series1->addMarSystem(Fanout3);
Series1->addMarSystem(mng.create("classifier", "Classifier"));
```

4.3 MarSystem Composites

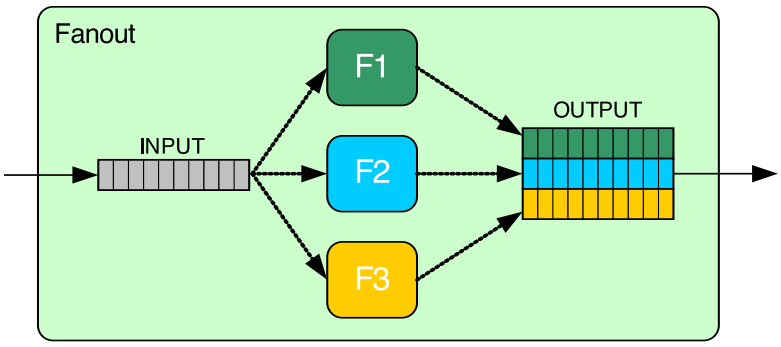
4.3.1 Series



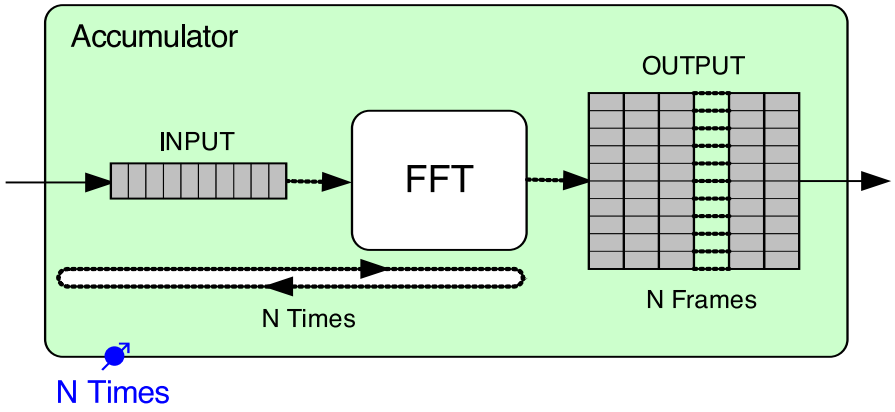
4.3.2 Parallel



4.3.3 Fanout



4.3.4 Accumulator



4.4 Assembling a network

4.4.1 Including libraries and linking

Unless you write your program inside the '`marsyas/src`' directory, you need to include the Marsyas libraries in your project.

5 Programming

5.1 Writing your own MarSystems

It's relatively straightforward to extend Marsyas by writing your own Marsystems. As mentioned before each MarSystems must basically support the process method that handles the dataflow and the update method that handles the control messages. There are certain conventions that need to be followed so typically it is a better idea to copy and modify an existing Marsystem rather than writing one from scratch. The simple canonical example of a MarSystem that is what I use as a template when I write a new Marsystem is Gain.h and Gain.cpp

6 Interoperability

Intro text

6.1 Audio and MIDI

- **RtAudio**: Multiplatform C++ API for realtime audio input/output
  - Linux (native ALSA, JACK, and OSS)
  - MacOSX©
  - Windows© (DirectSound© and ASIO©)
  - SGI©
- **RtMIDI**: Multiplatform C++ API for realtime MIDI input/output
  - Linux (ALSA)
  - MacOSX©
  - Windows©
  - SGI©

http://www.music.mcgill.ca/~gary/rtaudio/
http://www.music.mcgill.ca/~gary/rtmidi/

6.2 Open Sound Control (OSC)

This is coming soon, right George? It had better be soon...

6.3 WEKA

6.4 MATLAB

6.5 Python

6.6 Trolltech Qt4

7 Future Work

As is the case with most software projects, there is a lot of future work that I would like to do and I welcome any assistance, feedback, requests for features you might have. Most of my plans can be found in the TODO file of the distribution. However it is not clearly written and intended for my own personal use.

Some of the more long-term and ambitious plans for the future will be described in this section. If you would like to contribute some code to the project and you don't know where to start this section might give you some ideas.

python bindings msl interpreter with vertical syntax msl interpreter with chuck-like syntax SDIFF IO Open Sound Control (OSC)

8 Users and Projects

One of the greatest feelings a researcher/programmer can have is learning about people around the world doing exciting things with her/his software. There have been many amazing projects done with Marsyas and I hope to include them all in this section. If you are working on Marsyas and your name is not here, I would love to learn about your project and include it in this section. Also send me an email if you are one of the people mentioned in this section and you have some more information about how marsyas was used in your project. I would also like to thank all of you who have found Marsyas useful and helped me make it a better software framework.

(NOTES TO MYSELF about what I should write)

Mark Cardle Moodlogic AllMusic Inc. The Netherlands IslandGame Luis Gustavo Martins Stephaan Lippens Tao Li Karin Koshina ??? (spelling) Chris West ??? (spelling) George Tourtellot Corrie Elder Kris West

The subdirectory distributed contains sources and sinks that can be used to transmit sound and in general marsyas data over the network using TCP and UDP protocols. It enables cool thing like reading soundfiles on one computer sending fft frames for analysis to multiple computers and then assembling the results. The actual code has only be tested under Linux and is under construction so you at your own risk or even better fix the problems.

The Index