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 and industry groups around the world but it is slowly being phased out. .

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.

1.3 About the documentation

In addition to this manual, there are example files included in the Marsyas source tree. Many of these files are also included in the manual, but you may prefer to examine these files in your favorite text editor with your own syntax highlighting.

- '`examples/`': the simplest examples are here. These files are provided only for learning; they have little purpose as actual programs.
- '`examples/Qt4-tutorial/`': a simple Qt4/Marsyas program. Just like the files in '`examples/`', it has little value as a standalone program.

Once you are familiar with everything covered in this manual and those examples, you should examine the source code:

- '`apps/`': the source code for real Marsyas executables. These are real, working programs. This means that they have poor comments, bad variable names, and are difficult to read. :)
- '`marsyas/`': the source code for MarSystems. These are the basic building blocks of Marsyas, and are even more difficult to read.

1.3.1 Help!

The documentation for Marsyas is still a work in progress; we can use all the help we can get. Don't say "oh, I don't know enough" or "I'm not good at writing English." The question

is not "could anybody create something better than my suggestion?" – the question is "is this better than nothing?" Remember the most important thing about documentation:

> Documentation is like sex: when it is good, it is very, very good; and when it is bad, it is better than nothing. (Dick Brandon)

Manual

If you can add something to the docs, please send an email to <marsyas-developers@lists.sourceforge.net>. A formal patch for the texidoc is not required; we can take care of the technical details. Here is an example of a perfect documentation suggestion:

```
To: marsyas-users@lists.sourceforge.net
From: helpful-user@example.net
Subject: doc addition

In 4.2.1 Implicit patching vs. explicit patching, please add
----
It could be helpful to think of this like blah blah blah.
----
to the second paragraph.
```

Examples

Small, easy-to-understand examples are also great. If you have some source code that illustrates something, we can add it to '`examples/`'. You don't have to write any English at all!

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 tarball (something like '`marsyas-0.2.8.tar.gz`'). Uncompress this file using whatever uncompression program you prefer (`tar -xf`, winzip, etc).

2.1.2 Development Version

The latest version can be obtained from the subversion repository stored at the sourceforge website. Although constantly in flux the latest version is typically relatively stable and usable and if it is not we are quick in fixing it. In contrast releases happen infrequently. If you are planning on working extensively with Marsyas and writing your own source code it is highly recommend that you download a subversion working copy. If you only require one of the tools provided in Marsyas and don't plan to explore the framework then the regular releases should suffice. To check out a working copy do:

```
svn co https://marsyas.svn.sourceforge.net/svnroot/marsyas/trunk my-marsyas-dir
```

You can replace '`my-marsyas-dir`' with any directory you want. The version/release (version 0.2 release 10) is independently assigned from subversion revisions and the latest can be found by checking the sourceforge website.

2.2 System requirements

Warning! Marsyas does not work with qmake in Qt-4.3. Please downgrade to Qt-4.2.3 or use the autotools method of compiling Marsyas.

2.2.1 Linux

Required:

- Standard development environment (gcc, g++, etc): probably already installed.
- QT/Trolltech development platform.

Optional:

- LibMAD: mp3 support

2.2.2 MacOSX

Required:

- Xcode: standard development platform on OSX. Please install from your OSX installation CD/DVD, or download.
- QT/Trolltech development platform.

Optional:

- LibMAD: mp3 support

2.2.3 Windows

On Windows, you may use either Cygwin or Microsoft Visual Studio

2.2.3.1 Cygwin

Cygwin is a unix environment for Windows.

Required:

- Cygwin, with gcc and autotools installed (these should be installed by default)
- Microsoft DirectX SDK and Platform SDK

2.2.3.2 Microsoft Visual Studio

Required:

- Microsoft Visual Studio (visual studio express can compile Marsyas after a bit of fiddling with preferences)
- Microsoft DirectX SDK and Platform SDK
- QT/Trolltech development platform

Optional:

- LibMAD: mp3 support

2.3 Configuring Marsyas

2.3.1 ...with qmake

Marsyas can be built with qmake, which is the Makefile generator in Qt. To build Marsyas, go to the base of the source tree and edit '`marsyasConfig.pri`'. You will probably want to change the settings for *release/debug mode*, *WARNINGS/LOGS*, and *MATLAB engine classes*, and *MP3 MAD*.

Once you have selected the options you want, simply type

```
qmake
```

Requirement: Qt-4.2.3. Marsyas does not work with qmake in Qt-4.3. Please downgrade to Qt-4.2.3 or use the autotools method of compiling Marsyas.

2.3.2 ...with autotools

Marsyas may be compiled using the standard GNU configure script:

```
./configure
```

Marsyas can be customized using various configuration options. The current list of available options can be viewed by typing:

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

For example, to compile Marsyas with assertions enabled, mp3 support through libmad, and writing warnings to a file, one would do:

```
./configure --enable-assert --with-mad --enable-log2file
```

A frequent variation (if you don't have root priviledges) is to install Marsyas in your home directory:

```
./configure --prefix=$HOME
make
make install
```

2.3.2.1 Debugging options

```
./configure --enable-debug --enable-assert --enable-warnings
--enable-diagnostics --enable-log2file
make clean
make
```

The resulting program can be run under `gdb` to track down problems.

2.4 Compiling Marsyas

2.4.1 ...on *nix (Linux, FreeBSD, MacOSX)

After Section 2.3 [Configuring Marsyas], page 6, simply type

```
make

(optional, as root unless you changed the installation directory)
make install
```

2.4.2 ...on Windows

After Section 2.3 [Configuring Marsyas], page 6, simply type

```
nmake (nmake debug or nmake all for debug or debug and release builds)
```

This builds marsyas.lib, all (at least most of them) command line apps (sfplay, bextract, etc) and MarPlayer and MarPhasevocoder.

To generate a MSVC .vcproj for a project just cd into the app dir (e.g. apps/Qt4Apps/Meaws) and do:

```
qmake -t vcapp
```

In case you also want to create a .vcproj for the marsyas lib, cd into marsyas dir and do:

```
qmake -t vclib
```

2.5 Platform-specific notes

2.5.1 Linux

Marsyas is mainly developed under Linux so installing under Linux is quite straightforward. 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. Section 2.3 [Configuring Marsyas], page 6.

Requirement: ALSA headers. On most distributions, this is a package called `alsa-devel` or `libalsa-devel`.

2.5.2 Mac OSX

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.

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. Section 2.3 [Configuring Marsyas], page 6.

When trying to record audio, the sample rate must be specified explicitly:

```
recNet->updctrl("AudioSource/srcRec/mrs_real/israte", 44100.0);
recNet->updctrl("AudioSource/srcRec/mrs_bool/initAudio", true);
```

2.5.3 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.

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

2.5.4 Windows (Visual Studio)

A few of our developers use Visual Studio, so this environment is fairly well supported.

```
Anyone wanting to use MSVC2005 (and probably MSVC6, MSVC2003 and
MSVC2005express) and Qt4.x opensource please try doing the following:

1)  Use the most recent version of qmake, available at:

http://qtnode.net/wiki/Qt4_with_Visual_Studio

2)  Do not forget to put you Qt bin dir in the system path (i.e.
c:\Qt\4.2.2\bin), so you can use qmake anywhere, and to define the
QMAKESPEC env var, that for the case of MSVC2005 should be
win32-msvc2005.

3) open a MSVC2005 command prompt (you can find this in
Start->Programs->Microsoft Visual Studio->Visual Studio Tools); this cmd
prompt has all the env variables correctly configured in case you do not
have them configured in your system)

3) run "qconfigure msvc2005" (without the quotes) and follow the
instructions (basically reply yes whenever asked).

4) after the successful build of qmake and the subsequent generation
of the makefiles for the patched Qt code, just do nmake to build the
Qt lib (this will take a while, so go grab a coffee or something! ;-))

5) When done, you should now be able to create MSVC2005 project using
qmake -t vcapp/vclib from all your marsyas .pro!
```

2.6 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, Changelog:** text files with important information in theory. However, these have not been updated in years. We use `svn` log messages instead of the Changelog.
- **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.
- **marsyasConfig.pri:** edit this file to select your configure options when using qmake.
- **marsyasAll.pro:** only developers should edit this file.

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.
- **bin:** executable files are compiled in '`bin/release`' or '`bin/debug`'.
- **apps:** the source code for the above executables.
- **doc:** contains both the user manual (which you are currently reading). To regenerate the manual in pdf or html type (in the doc subdirectory)

```
make pdf
make html
```

- **apps/Qt4Apps:** 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.

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.

The executable files may be found in '`bin/release/`', while the source code for those files is in '`apps/{DIR}/`'.

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. Building networks is described in Chapter 5 [Programming applications], page 24, and writing new MarSystems is described in Chapter 6 [Developing MarSystems], page 33.

4.1 Architecture overview

4.1.1 Building MarSystems

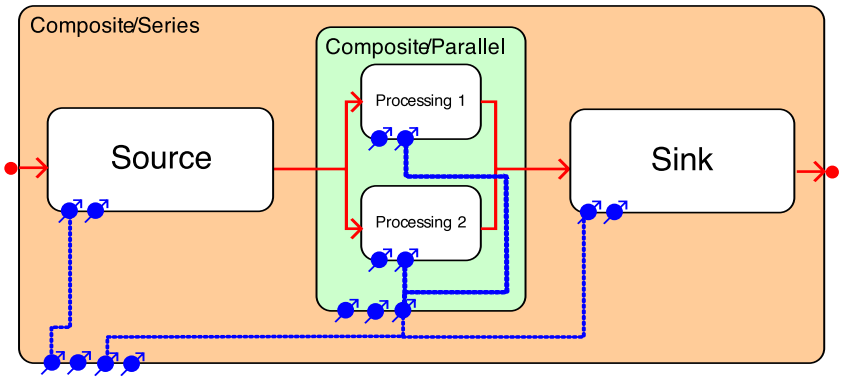
The basic idea behind the design of Marsyas is that any audio analysis/synthesis computation can be expressed as some type of processing object, which we call MarSystem, that reads data from an input slice of floating point numbers, performs some computation/transformation based on data, and writes the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

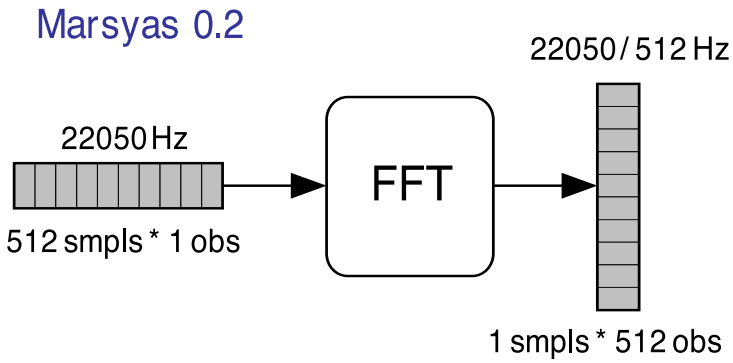
As is very frequently the case with audio processing networks objects the input of each stage is the output of the previous stage. This way of assembling MarSystems is called a Series composite. Once a Series Composite is formed it can basically be used as one MarSystem that does the whole thing. A figure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

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.



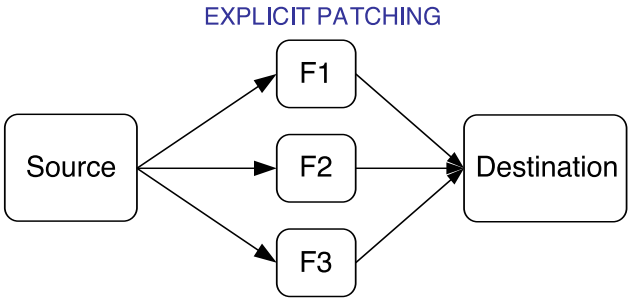
4.2 Implicit patching

4.2.1 Implicit patching vs. explicit patching

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

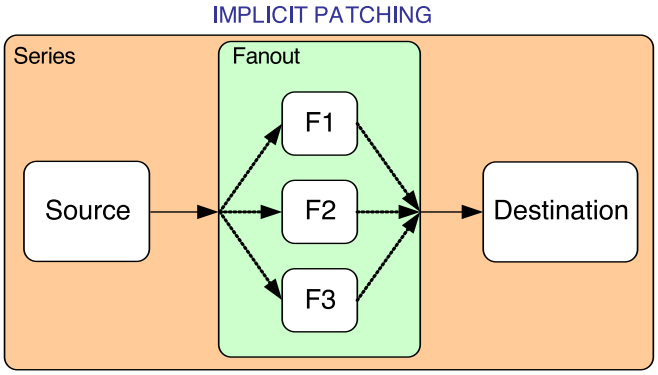
```
# EXPLICIT PATCHING: block definitions
source, F1, F2, F3, destination;
# connect the in/out ports of the blocks
connect(source, F1);
connect(source, F2);
connect(source, F3);
connect(F1, destination);
connect(F2, destination);
```

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



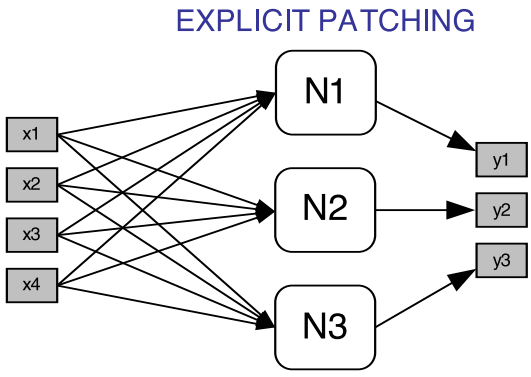
Marsyas uses *implicit patching*: connections are made automagically when blocks are created,

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



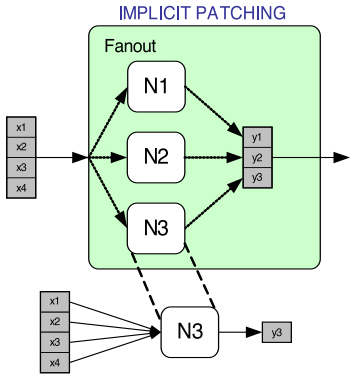
4.2.2 Implicit patching advantages

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

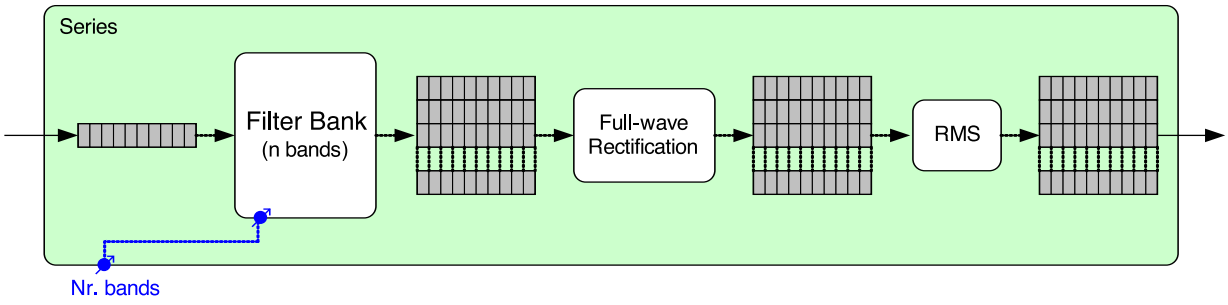


With implicit patching, this is much more manageable.

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

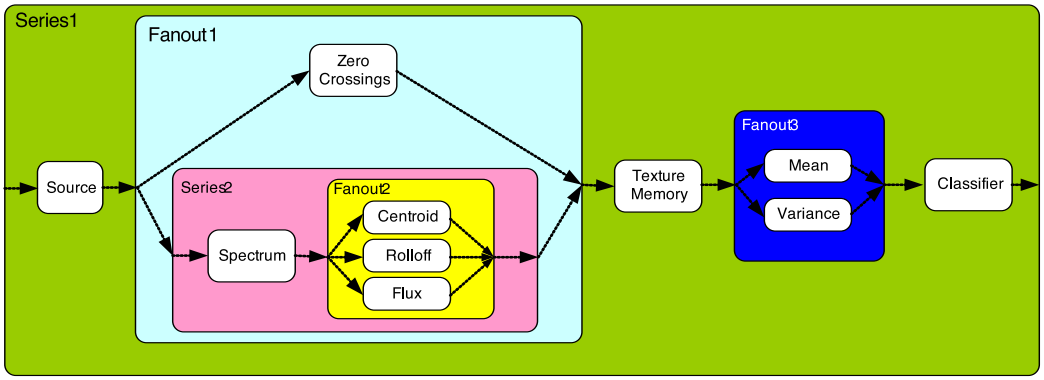


Implicit patching can automagically adjust the connections without requiring any code recompiliation. For example, we can change the number of bands in a filter bank without changing any code.



4.2.3 Patching example of Feature extraction

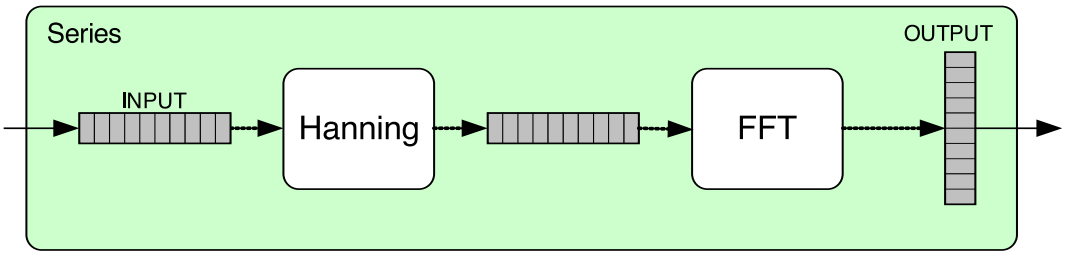
Suppose we wish to create a typical feature extraction program:



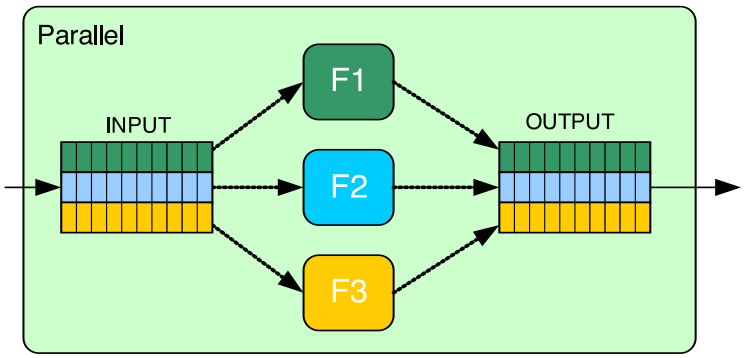

```
MarSystemManager mng;
MarSystem* Series1 = mng.create("Series", "Series1");
MarSystem* Fanout1 = mng.create("Fanout", "Fanout1");
MarSystem* Series2 = mng.create("Series", "Series2");
MarSystem* Fanout2 = mng.create("Fanout", "Fanout2");
MarSystem* Fanout3 = mng.create("Fanout", "Fanout3");
Fanout3->addMarSystem(mng.create("Mean", "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

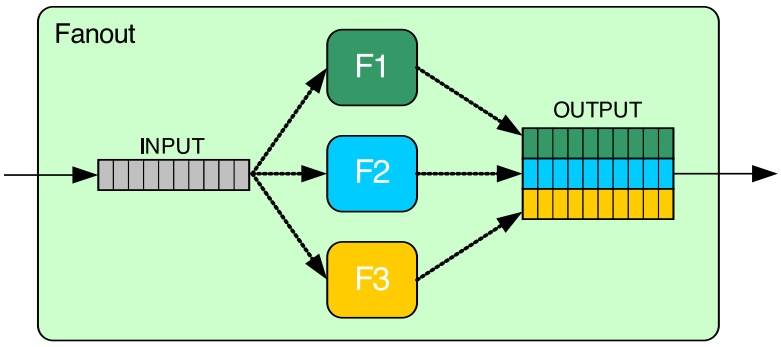
4.3.1 Series



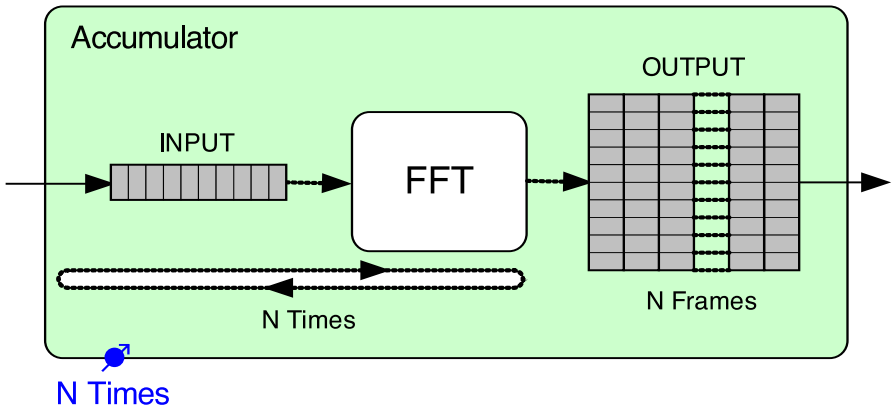
4.3.2 Parallel



4.3.3 Fanout



4.3.4 Accumulator



4.4 Predefined variable types

Marsyas contains some predefined, portable data types:

```
mrs_bool
mrs_natural   (integers)
mrs_real
mrs_complex
```

There is also the `realvec` variable type. A `realvec` is an array of `mrs_real` values; this is the most common type of variable in Marsyas (all the audio processing is done via `realvec`s, for example). Many operations can be performed on `realvec`s, including statistical operations (find the mean, median, standard variance, etc). Using these predefined data types is highly recommended.

```
realvec foo;
foo.create(10);
for (mrs_natural i=0; i<10; i++) {
  foo(i) = i;
}
...
foo.~realvec();  // delete the data

realvec *bar;
bar->create(20,20);
for (mrs_natural i=0; i<20; i++) {
  for (mrs_natural j=0; j<20; j++) {
    (*bar)(i,j) = i+j;
  }
}
...
bar->~realvec();
```

4.5 Architecture limitations

Due to the way that observations behave in Marsyas, in some cases it is impossible to differentiate between a stereo signal and a mono signal which is twice as long. In particular, there is currently no direct way to tell apart a stereo pair of spectrums from a mono spectrum with twice the number of bins.

In these cases, we recommend that you use a Parallel Composite: split the stereo signal into separate mono dataflows (using Parallel), then treat each mono signal individually.

5 Programming applications

5.1 Including libraries and linking

To use any marsyas code in your program(s), you need to include the Marsyas libraries in your project.

5.1.1 Linux and other *nixes (including Cygwin)

The easiest way to compile Marsyas projects is to use a makefile. In your working directory, create a file called '`Makefile`' which contains

```
all:
rm -f *.o
g++ -Wall -O2 -I./ -I/usr/local/include/marsyas -c -o myfile.o myfile.cpp
g++ -Lusr/local/lib -o myfile myfile.o -lmarsyas -lasound
```

(place a tab in front of the three lower lines instead of spaces)

If you installed marsyas in a location other than '`/usr/local/`', then change the `-I` and `-L` directories accordingly. For example, if you installed marsyas in '`${HOME}/usr/`', then you would use `-I${HOME}/usr/include/marsyas` and `-L${HOME}/usr/lib`. Once the '`Makefile`' is written, simply type `make` to compile your file.

5.1.2 MacOS X

To compile Marsyas projects in OSX, use the same '`Makefile`' as is used in other *nixes, but replace the final line with

```
g++ -Lusr/local/lib -o myfile myfile.o -lmarsyas -framework CoreAudio \
-framework CoreMidi -framework CoreFoundation
```

5.1.3 Windows Visual Studio

Please see the documentation for your compiler / development environment.

5.2 Example programs

The most efficient way to learn how to write programs that use MarSystem is to read these examples. We recommend that you use these examples as templates when you begin to write your own programs.

5.2.1 Hello World (playing an audio file)

Instead of printing "Hello World!" , we shall play a sound file. This is relatively straightforward: we create a `MarSystem` which is a series of `SoundFileSource`, `Gain`, and `AudioSink`. Once the network is created and the controls are given, we call `tick()` to make time pass until we have finished playing the file.

helloworld.cpp

```
#include "MarSystemManager.h"
using namespace std;
using namespace Marsyas;

void sfplay(string sfName, float gain) {
        MarSystemManager mng;

        MarSystem* playbacknet = mng.create("Series", "playbacknet");

        playbacknet->addMarSystem(mng.create("SoundFileSource", "src"));
        playbacknet->addMarSystem(mng.create("Gain", "gt"));
        playbacknet->addMarSystem(mng.create("AudioSink", "dest"));

        playbacknet->updctrl("SoundFileSource/src/mrs_string/filename", sfName);
        playbacknet->updctrl("Gain/gt/mrs_real/gain", gain);
        playbacknet->updctrl("AudioSink/dest/mrs_bool/initAudio", true);

        while ( playbacknet->getctrl("SoundFileSource/src/mrs_bool/notEmpty")->toBool() )
                playbacknet->tick();
        }
        delete playbacknet;
}

int main(int argc, const char **argv) {
        string fileName;
        float gain;
        if (argc<2) { cout<<"Please enter filename."<<endl; exit(1); } else
                { fileName = argv[1]; }
        if (argc<3) { gain = 1; } else
                { gain = atof(argv[2]); }
        cout << "Playing file " << fileName << " at volume " <<
                gain << endl;

        sfplay(fileName,gain);
        exit(0);
}
```

5.2.2 Reading and altering controls

Here we have modified the example from the previous section: we have added the ability to start at an arbitrary position (time) inside the audio file. To calculate the starting position in the file, we must know the sample rate and number of channels. We get this information from the `SoundFileSource` with `getctrl`.

controls.cpp

```
#include "MarSystemManager.h"
using namespace std;
using namespace Marsyas;

void sfplay(string sfName, float gain, float start) {
	MarSystemManager mng;

	MarSystem* playbacknet = mng.create("Series", "playbacknet");

	playbacknet->addMarSystem(mng.create("SoundFileSource", "src"));
	playbacknet->addMarSystem(mng.create("Gain", "gt"));
	playbacknet->addMarSystem(mng.create("AudioSink", "dest"));

// calculate the starting position.
	mrs_natural nChannels = playbacknet->getctrl("SoundFileSource/src/mrs_natural/nCh
	mrs_real srate = playbacknet->getctrl("SoundFileSource/src/mrs_real/israte")->toR
	mrs_natural startPosition = (mrs_natural) (start * srate * nChannels);

	playbacknet->updctrl("SoundFileSource/src/mrs_string/filename", sfName);
	playbacknet->updctrl("Gain/gt/mrs_real/gain", gain);
	playbacknet->updctrl("AudioSink/dest/mrs_bool/initAudio", true);

// set the starting position of the source
	playbacknet->updctrl("SoundFileSource/src/mrs_natural/pos", startPosition);

	while ( playbacknet->getctrl("SoundFileSource/src/mrs_bool/notEmpty")->toBool() )
		playbacknet->tick();
	}
	delete playbacknet;
}

int main(int argc, const char **argv) {
	string fileName;
	float gain, start;
	if (argc<2) { cout<<"Please enter filename."<<endl; exit(1); } else
		{ fileName = argv[1]; }
	if (argc<3) { gain = 1; } else
		{ gain = atof(argv[2]); }
	if (argc<4) { start = 0; } else
		{ start = atof(argv[3]); }
	cout << "Playing file " << fileName << " at volume " <<
		gain << " starting at " << start << " seconds" << endl;

	sfplay(fileName,gain,start);
	exit(0);
```

```
}
```

5.2.3 Writing data to text files

Extract data from a network for further analysis (plotting, other programs, etc) is fairly easy to do with `PlotSink`.

writefile.cpp

```
#include "MarSystemManager.h"
using namespace std;
using namespace Marsyas;

void recognize(string sfName) {
       MarSystemManager mng;
       MarSystem* pnet = mng.create("Series", "pnet");
// standard network
       pnet->addMarSystem(mng.create("SoundFileSource", "src"));
       pnet->updctrl("SoundFileSource/src/mrs_string/filename", sfName);
       pnet->addMarSystem(mng.create("Spectrum","spk"));

// add a PlotSink wherever we want to get data from
       pnet->addMarSystem(mng.create("PlotSink","plot"));
       pnet->updctrl("PlotSink/plot/mrs_string/outputFilename", "out");

       while ( pnet->getctrl("SoundFileSource/src/mrs_bool/notEmpty")->toBool() ) {
              pnet->tick();
       }
       delete pnet;
}

int main(int argc, const char **argv) {
       string fileName;
       if (argc<2) { cout<<"Please enter filename."<<endl; exit(1); } else
              { fileName = argv[1]; }
       cout << "Processing file " << fileName << endl;

       recognize(fileName);
       exit(0);
}
```

5.2.4 Getting data from the network

Putting together a network of MarSystems is all well and good, but you probably want to do something with that data. In this example we simply print it to the screen, but the important thing to note is that we have the data at the level of C programming.

gettingdata.cpp

```
#include "MarSystemManager.h"
using namespace std;
using namespace Marsyas;

void recognize(string sfName) {
       MarSystemManager mng;
       MarSystem* pnet = mng.create("Series", "pnet");
// standard network
       pnet->addMarSystem(mng.create("SoundFileSource", "src"));
       pnet->updctrl("SoundFileSource/src/mrs_string/filename", sfName);
       pnet->addMarSystem(mng.create("Spectrum","spk"));

// variables to get data from network
       realvec in(pnet->getctrl("mrs_natural/inObservations")->toNatural(), pnet->getctrl("
       realvec out(pnet->getctrl("mrs_natural/onObservations")->toNatural(), pnet->getctrl

// counter variables
       int i,j;
       mrs_natural numberObservations = pnet->getctrl("mrs_natural/onObservations")->toN
       mrs_natural numberSamples = pnet->getctrl("mrs_natural/onSamples")->toNatural();
Samples will be 1, due to the Spectrum MarSystem.

       while ( pnet->getctrl("SoundFileSource/src/mrs_bool/notEmpty")->toBool() ) {
// don't tick because process() does that.
//              pnet->tick();
// get data from network
              pnet->process(in,out);

// display data
              for (i=0; i<numberObservations; i++) {
                     for (j=0; j<numberSamples; j++) {
                            cout<<out(i,j)<<" ";
                     }
              }
              cout<<endl;

              const realvec& processedData = pnet->getctrl("Spectrum/spk/mrs_realvec/pro
              cout << processedData << endl;
```

```
        }
        delete pnet;
}

int main(int argc, const char **argv) {
        string fileName;
        if (argc<2) { cout<<"Please enter filename."<<endl; exit(1); } else
                { fileName = argv[1]; }
        cout << "Processing file " << fileName << endl;

        recognize(fileName);
        exit(0);
}
```

5.2.5 Command-line options

Getting options from the command-line is fairly easy; Marsyas provides a handy object which parses the command-line for you.

commandOptions.cpp

```
#include "CommandLineOptions.h"

using namespace std;
using namespace Marsyas;

CommandLineOptions cmd_options;

int helpOpt;
int usageOpt;
mrs_natural naturalOpt;
mrs_real realOpt;
mrs_string stringOpt;

void
printUsage()
{
        MRSDIAG("commandOptions.cpp - printUsage");
        cerr << "Usage: commandOptions " << "file1 file2 file3" << endl;
        cerr << endl;  cerr << "where file1, ..., fileN are sound files in a MARSYAS su
        exit(1);
}
```

```
void
printHelp()
{
	MRSDIAG("commandOptions.cpp - printHelp");
	cerr << "commandOptions: Sample Program"<< endl;
	cerr << "------------------------------" << endl;
	cerr << endl;
	cerr << "Usage: commandOptions file1 file2 file3" << endl;
	cerr << endl;
	cerr << "where file1, ..., fileN are sound files in a Marsyas supported format"
	cerr << "Help Options:" << endl;
	cerr << "-u --usage               : display short usage info" << endl;
	cerr << "-h --help               : display this information " << endl;
	cerr << "-n --natural               : sets a 'natural' variable " << endl;
	cerr << "-r --real               : sets a 'real' variable " << endl;
	cerr << "-s --string               : sets a 'string' variable " << endl;
	exit(1);
}

void
initOptions()
{
	cmd_options.addBoolOption("help", "h", false);
	cmd_options.addBoolOption("usage", "u", false);
	cmd_options.addNaturalOption("natural", "n", 9);
	cmd_options.addRealOption("real", "r", 3.1415927);
	cmd_options.addStringOption("string", "s", "hello world");
}

void
loadOptions()
{
	helpOpt = cmd_options.getBoolOption("help");
	usageOpt = cmd_options.getBoolOption("usage");
	naturalOpt = cmd_options.getNaturalOption("natural");
	realOpt = cmd_options.getRealOption("real");
	stringOpt = cmd_options.getStringOption("string");
}



void doStuff(string printMe) {
	cout<<printMe<<endl;
}

int main(int argc, const char **argv) {
	initOptions();
```

```
        cmd_options.readOptions(argc,argv);
        loadOptions();

        vector<string> soundfiles = cmd_options.getRemaining();

        if (helpOpt)
        printHelp();

        if ( (usageOpt) || (argc==1) )
        printUsage();

        cout<<"Command-line options were:"<<endl;
        cout<<"              --natural: "<<naturalOpt<<endl;
        cout<<"              --real: "<<realOpt<<endl;
        cout<<"              --string: "<<stringOpt<<endl;
        cout<<"(these may simply be the default values)"<<endl;
        cout<<endl;
        cout<<"The rest of the command-line arguments were: "<<endl;

        vector<string>::iterator sfi;
        for (sfi = soundfiles.begin(); sfi != soundfiles.end(); ++sfi)
        {
                doStuff( *sfi );
        }
}
```

5.3 Other programming issues

5.3.1 Scheduler

To schedule events using the new scheduler code, see the Marsyas Expression Syntax documentation at http://www.cs.uvic.ca/~inb/work/expr.html

5.3.2 Visualizing data with gnuplot

Gnuplot is an open-source data plotting utility available on every operating system that Marsyas supports. More information (including downloads and installation instructions) can be found on the Gnuplot website.

Data in Marsyas can be plotted easily: simply write the realvec to a text file and call gnuplot on the result.

```
void someFunction() {
  string filename = "dataToPlot.txt";
realvec data;
```

```
data.allocate(size);
// ... do whatever processing here...
  data.writeText( filename );
  data.~realvec();
}
```

After compiling and running the program, the `dataToPlot.txt` file may be plotted with gnuplot.

```
gnuplot> plot "dataToPlot.txt"
```

6 Developing MarSystems

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.

6.1 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`' or '`MarSystemTemplateBasic .h .cpp`'.

6.2 Adding the MarSystem to Marsyas

The new MarSystem should be placed in the '`marsyas/`' directory, and must be added to '`MarSystemManager.cpp`' and the build process. Again, the easiest way is to look for `Gain` and do the same thing with your new MarSystem.

MarSystemManager

- '`marsyas/MarSystemManager.cpp`'

Build process: autotools

- '`marsyas/Makefile.am`'
- '`lib/release/Makefile.am`'

  Now execute `automake` and `./configure` before recompiling with `make`.

Build process: qmake

- '`marsyas/marsyas.pro`'

  Now execute `qmake` before recompiling with whichever compiler you use.

6.3 myUpdate() and myProcess()

Every MarSystem includes `myUpdate(...)` and `myProcess(...)`. `myProcess()` is called every time the MarSystem receives a `tick()` (ie all the time). `myUpdate()` is called whenever the values of controls that have state are changed.

In other words, resource-intensive operations (such as changing the buffer size, computing trigonomic functions, etc) that only depend on the controls should be performed inside `myUpdate()`.

Taking a real-world example, consider a simple one-pole high/low-pass filter where we wish to perform the following operations:

```
mrs_real fc = ctrl_fc ->to<mrs_real>();
mrs_real tanf = tan( PI * fc / 44100.0);
mrs_real c = (tanf - 1.0) / (tanf + 1.0);
for (t=1; t < inSampes_; t++) {
    az = c*in(0,t) + in(0,t-1) - c*out(0,t-1);
    out(0,t) = (1-az)/2;
}
```

Since `tanf` and `c` only depend on `fc`, they may be computed inside `myUpdate()` instead of `myProcess()`. If `fc` has not changed, we can use the old value `c` to perform the loop over our sound buffer; if the value of `fc` has changed, then `c` will be recomputed inside `myUpdate()`.

7 Interoperability

Intro text

7.1 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/

7.2 WEKA

It *can* be done; read the source code.

7.3 MATLAB

It *can* be done; read the source code.

7.4 Python

It *can* be done; read the source code.

7.5 Trolltech Qt4

To get started, look at the tutorial files:

tutorial.pro

```
#  your files
SOURCES = main.cpp
HEADERS = mainwindow.h
SOURCES += mainwindow.cpp
```

```
HEADERS += backend.h
SOURCES += backend.cpp

#  these files are common to every Marsyas/QT app
#  you will need to update these paths (and the paths in backend.h)
#  if you copy this directory elsewhere.
HEADERS += ../../apps/Qt4Apps/MarSystemQtWrapper.h
SOURCES += ../../apps/Qt4Apps/MarSystemQtWrapper.cpp

#  basic system variables
TARGET = runme

MARSYAS_INSTALL_DIR  = ${HOME}/usr/
#MARSYAS_INSTALL_DIR  = /usr/local
message("Marsyas was installed in $$MARSYAS_INSTALL_DIR, right?")

INCLUDEPATH += $$MARSYAS_INSTALL_DIR/include/marsyas
unix:LIBS += -lmarsyas -L$$MARSYAS_INSTALL_DIR/lib # -lmad -lvorbis -lvorbisfile
!macx:LIBS += -lasound
macx:LIBS += -framework CoreAudio -framework CoreMidi -framework CoreFoundation
```

main.cpp

```
/*
** Copyright (C) 2007 Graham Percival <gperciva@uvic.ca>
**
** This program is free software; you can redistribute it and/or modify
** it under the terms of the GNU General Public License as published by
** the Free Software Foundation; either version 2 of the License, or
** (at your option) any later version.
**
** This program is distributed in the hope that it will be useful,
** but WITHOUT ANY WARRANTY; without even the implied warranty of
** MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
** GNU General Public License for more details.
**
** You should have received a copy of the GNU General Public License
** along with this program; if not, write to the Free Software
** Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.
*/

#include "mainwindow.h"
int main(int argc, char *argv[]) {
// to keep this example as simple as possible, we only take the
// filename from the command-line.
```

```
        string fileName;
        if (argc<2) { cout<<"Please enter filename."<<endl; exit(1); } else
                { fileName = argv[1]; }

        QApplication app(argc, argv);
        MarQTwindow marqt(fileName);
        marqt.show();
        return app.exec();
}
```

mainwindow.h

```
/*
** Copyright (C) 2007 Graham Percival <gperciva@uvic.ca>
**
** This program is free software; you can redistribute it and/or modify
** it under the terms of the GNU General Public License as published by
** the Free Software Foundation; either version 2 of the License, or
** (at your option) any later version.
**
** This program is distributed in the hope that it will be useful,
** but WITHOUT ANY WARRANTY; without even the implied warranty of
** MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
** GNU General Public License for more details.
**
** You should have received a copy of the GNU General Public License
** along with this program; if not, write to the Free Software
** Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.
*/

#include <QApplication>
#include <QPushButton>
#include <QSlider>
#include <QVBoxLayout>
#include <QLCDNumber>

#include <iostream>
using namespace std;

#include "backend.h"

class MarQTwindow : public QWidget {
        Q_OBJECT
public:
        MarQTwindow(string fileName);
```

```
        ~MarQTwindow();

public slots:
        void setMainPosition(int newPos);

private:
        MarBackend *marBackend;
        QLCDNumber *lcd;
};
```

mainwindow.cpp

```
/*
** Copyright (C) 2007 Graham Percival <gperciva@uvic.ca>
**
** This program is free software; you can redistribute it and/or modify
** it under the terms of the GNU General Public License as published by
** the Free Software Foundation; either version 2 of the License, or
** (at your option) any later version.
**
** This program is distributed in the hope that it will be useful,
** but WITHOUT ANY WARRANTY; without even the implied warranty of
** MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
** GNU General Public License for more details.
**
** You should have received a copy of the GNU General Public License
** along with this program; if not, write to the Free Software
** Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.
*/

#include "mainwindow.h"

MarQTwindow::MarQTwindow(string fileName) {
// typical Qt front-end
        QPushButton *quit = new QPushButton(tr("Quit"));
        connect(quit, SIGNAL(clicked()), qApp, SLOT(quit()));

        QPushButton *updatePos = new QPushButton(tr("Update position"));

        QSlider *volume = new QSlider (Qt::Horizontal);
        volume->setRange(0,100);
        volume->setValue(50);

        lcd = new QLCDNumber();
        lcd->setNumDigits(10);
```

```
        QVBoxLayout *layout = new QVBoxLayout;
        layout->addWidget(volume);
        layout->addWidget(updatePos);
        layout->addWidget(lcd);
        layout->addWidget(quit);
        setLayout(layout);

// make the Marsyas backend
        marBackend = new MarBackend();
        marBackend->openBackendSoundfile(fileName);

// make connections between the Qt front-end and the Marsyas backend:

//              Qt -> Marsyas
        connect(volume, SIGNAL(valueChanged(int)),
                        marBackend, SLOT(setBackendVolume(int)));

//              Marsyas -> Qt
        connect(marBackend, SIGNAL(changedBackendPosition(int)),
                        this, SLOT(setMainPosition(int)));

//              Qt -> Marsyas (getBackendPosition) -> Qt (changedBackend-
Position)
        connect(updatePos, SIGNAL(clicked()),
                        marBackend, SLOT(getBackendPosition()));
}

MarQTwindow::~MarQTwindow() {
        delete marBackend;
}

void MarQTwindow::setMainPosition(int newPos) {
        lcd->display(newPos);
}
```

backend.h

```
/*
** Copyright (C) 2007 Graham Percival <gperciva@uvic.ca>
**
** This program is free software; you can redistribute it and/or modify
** it under the terms of the GNU General Public License as published by
** the Free Software Foundation; either version 2 of the License, or
** (at your option) any later version.
```

```
**
** This program is distributed in the hope that it will be useful,
** but WITHOUT ANY WARRANTY; without even the implied warranty of
** MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
** GNU General Public License for more details.
**
** You should have received a copy of the GNU General Public License
** along with this program; if not, write to the Free Software
** Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.
*/

#include <QObject>
#include <QTimer>
#include "MarSystemManager.h"
#include "../../apps/Qt4Apps/MarSystemQtWrapper.h"

#include <iostream>
using namespace std;
using namespace Marsyas;

class MarBackend: public QObject {
       Q_OBJECT

public:
       MarBackend();
       ~MarBackend();
       void openBackendSoundfile(string fileName);

public slots:
       void setBackendVolume(int value);
       void getBackendPosition();

signals:
       void changedBackendPosition(int value);

private:
// in order to make the MarSystem act like a Qt object,
// we use this wrapper:
       MarSystemQtWrapper *mrsWrapper;
// ... and these pointers:
       MarControlPtr filenamePtr;
       MarControlPtr gainPtr;
       MarControlPtr positionPtr;

// typical Marsyas network:
       MarSystem *playbacknet;
};
```

backend.cpp

```
/*
** Copyright (C) 2007 Graham Percival <gperciva@uvic.ca>
**
** This program is free software; you can redistribute it and/or modify
** it under the terms of the GNU General Public License as published by
** the Free Software Foundation; either version 2 of the License, or
** (at your option) any later version.
**
** This program is distributed in the hope that it will be useful,
** but WITHOUT ANY WARRANTY; without even the implied warranty of
** MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
** GNU General Public License for more details.
**
** You should have received a copy of the GNU General Public License
** along with this program; if not, write to the Free Software
** Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.
*/

#include "backend.h"
using namespace Marsyas;

MarBackend::MarBackend() {
// make a typical Marsyas network:
       MarSystemManager mng;
       playbacknet = mng.create("Series", "playbacknet");
       playbacknet->addMarSystem(mng.create("SoundFileSource", "src"));
       playbacknet->addMarSystem(mng.create("Gain", "gain"));
       playbacknet->addMarSystem(mng.create("AudioSink", "dest"));
       playbacknet->updctrl("AudioSink/dest/mrs_bool/initAudio", true);

// wrap it up to make it pretend to be a Qt object:
       mrsWrapper = new MarSystemQtWrapper(playbacknet);
       mrsWrapper->start();

// make these pointers so that we can interface with the network
// in a thread-safe manner:
       filenamePtr = mrsWrapper->getctrl("SoundFileSource/src/mrs_string/filename");
       gainPtr = mrsWrapper->getctrl("Gain/gain/mrs_real/gain");
       positionPtr = mrsWrapper->getctrl("SoundFileSource/src/mrs_natural/pos");

// demonstrates information flow:  Marsyas->Qt.
       QTimer *timer = new QTimer(this);
```

```
        connect(timer, SIGNAL(timeout()), this, SLOT(getBackendPosition()));
        timer->start(1000);
}

MarBackend::~MarBackend() {
        delete mrsWrapper;
        delete playbacknet;
}

void MarBackend::openBackendSoundfile(string fileName) {
        mrsWrapper->updctrl(filenamePtr,fileName);
        mrsWrapper->play();
}

void MarBackend::setBackendVolume(int vol) {
        float newGain = vol/100.0f;
        mrsWrapper->updctrl(gainPtr, newGain);
}

void MarBackend::getBackendPosition() {
        int newPos = (int) positionPtr->to<mrs_natural>();
        emit changedBackendPosition(newPos);
}
```

After examining that project, see '`apps/Qt4Apps/MarPlayer/`', followed by the other examples in the '`apps/Qt4Apps/`' directory.

7.6 OCaml

To combine Marsyas and OCaml, see the MarsyasOCaml documentation at http://www.cs.uvic.ca/~inb/work/marsyasOCaml/

The Index