Marsyas User Manual

For version 0.3

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

Graham Percival and George Tzanetakis

Table of Contents

1 General information

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 and community of developers from around the world. It has been in development for over 10 years and has been used for a variety of projects in academia and industry. More information about such projects as well as a list of publications related to Marsyas can be found at the **Marsyas** website http://marsyas.info/. 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 composed 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 the research of the developers in the emerging area of Music Information Retrieval (MIR). Anyone who finds the framework useful is welcome to use it and contribute to it.

There is a lot of work behind the development of Marsyas. Although Marsyas is and will remain free software, any form of financial or hardware support is more than welcome. The sourceforge page contains a link for people to donate money to the project and any contribution is welcome and will help further improve the framework. Also implementation of specific features can be motivated by donation. Finally for companies desiring to incorporate Marsyas into closed source software products a commercial license is also available (this dual licensing scheme is similar to the one used by Trolltech the company behind the Qt toolkit). For more information about the dual licensing contact George Tzanetakis.

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.

The community of Marsyas developers has grown over the years with currently (Spring 2009) approximately 4-5 developers committing regularly code and close to 30 developers having committed code over the past few years.

We are very proud that Marsyas is used for a variety of projects in both academia and industry and looking forward to continue growing and expanding the framework with your help.

have fun,

George Tzanetakis (gtzan at cs dot uvic dot ca)

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

1.3.1 Latest version

The documentation is uploaded regularly to http://marsyas.info/docs/index.html

1.3.2 User manual and Developer's manual

This manual (the User manual) is the main documentation for Marsyas; it covers the use of existing tools, creating your own tools, and even how to create new MarSystems.

The Developer's manual is aimed at people who commit material back to Marsyas. It covers tips and tools which make contributing easier, explains the automatic testing (and how to include your code in these tests), and how to write documentation.

1.3.3 Help wanted!

Although Marsyas documentation has improved an incredible amount in the past year, we do not have the resources (time / energy / interest / money) to cover everything. When the documentation team has identified a particular need in the manual which they cannot fulfill themselves, they place a notice to that effect:

> **Help wanted: missing info!**
>
> Brief note explaining what is missing.
>
> *If you can fill in any details, please see Section "Contributing documentation" in Marsyas Developer's Manual.*

1.4 Beyond the manuals

1.4.1 Examples

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.

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

1.4.2 MarSystem source documentation

Many MarSystems have been documented; see Section 7.1 [Library reference], page 69 for more information.

1.4.3 Source code

Unfortunately this manual and the source code documentation are not complete. Once you are familiar with everything covered in this manual, you should examine the source code:

- '`src/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. :)
- '`src/marsyas/`': the source code for MarSystems. These are the basic building blocks of Marsyas, and are even more difficult to read.

If you gain any knowledge from the source files that is not covered in the manual, please help improve the documentation. For more information, see Section "Contributing documentation" in *Marsyas Developer's Manual*.

2 Source installation

This chapter shows the steps to get a working version of Marsyas. Compiling and installing a large piece of software like Marsyas is not trivial. In the summer 2008 we switched our build system from a combination of autotools and qmake to using CMake. We believe that this change has greatly simplified the building process across all supported platforms and configurations. Starting with the Belfast release 0.2.17 CMake is the only supported way to build Marsyas.

2.1 Get Marsyas sources

Stable(-ish) Version

Note: Marsyas is currently approaching the next stable release (version 0.5.0) which will bring a lot of changes that also affect the build process. Instructions in this manual do not apply anymore to the last stable release! For this reason we suggest to use the latest development state of Marsyas sources.

To get the latest Marsyas sources visit the Marsyas GitHub project page and use the "Download ZIP" button.

If you still want to use the last stable Marsyas release, you can obtain it from SourceForge: http://sourceforge.net/projects/marsyas/

Development Version

If you intend to keep updating your copy of the Marsyas sources along with the latest development or contribute to development, you will need to use git to interact with the Marsyas git repository.

The Marsyas git repository is hosted at GitHub: https://github.com/marsyas/marsyas/

Read access

To clone the repository onto your computer, use the following git command:

```
git clone https://github.com/marsyas/marsyas.git
```

Write access

The above command will only provide read access to the online repository (for latest updates). If you want to publish your contributions back to the online repository, you will need write access:

1. Create a GitHub account, if you don't have one yet.
2. Ask permission for write access to the repository on marsyas development mailing list.
3. Clone the repository using the following command:

```
git clone git@github.com:marsyas/marsyas.git
```

2.2 Step-by-step building instructions

You will find here detailed step-by-step instructions for building Marsyas with basic features only. These instructions are intended as a beginner's introduction to the procedure of software building in general. For extra features, please look at other sections of the manual.

Please note that there are a lot of optional extensions to the basic Marsyas functionality which require installation of additional supporting software. Moreover, there may be slight differences in installation procedures based on your particular computer setup. Only a few representative cases are described here. If you find troubles you can always ask for help on the user or developer mailing lists. If you find solutions to issues not covered here, please send us information so we can include it in this manual.

The following sections contain instructions for building latest Marsyas on most recent versions of Linux, Mac OS X and Windows.

The following sections cover building older Marsyas versions (up to 0.4.8), and on older platform versions. They are most probably not relevant for the latest Marsyas sources anymore:

2.2.1 Building latest Marsyas on Debian/Ubuntu

These instructions are for building the latest development version of Marsyas. It is assumed that you have already obtained the source code. If not, please consult Section 2.1 [Get Marsyas sources], page 6 for instructions.

2.2.1.1 Install prerequisits

These are the prerequisits and their corresponding Debian package names:

- (Required) GNU Compiler Collection and related tools: 'build-essential'
- (Required) CMake: 'cmake'
- (Recommended) JACK development files: 'libjack-jack2-dev' or 'libjack-dev' for older version
- (Recommended) ALSA development files: 'libasound2-dev'

If you want to use all Marsyas features, you will need a compiler with a decent C++11 support. Any recent Linux distribution will probably include a new enough GNU compiler.

JACK and ALSA are different interfaces between a program and the computer's real-time audio input/output capabilities. Preferrably, you would build with support for both, so that either one of them can be used.

As usual on Debian-based Linux distributions, let's use Apt to install required packages. You can install all of the above packages with a single command like this:

```
sudo apt-get install build-essential cmake libjack-jack2-dev libasound2-dev
```

2.2.1.2 Configure Marsyas using CMake

1. Navigate to the top-level directory of Marsyas sources. For example:

   ```
   cd ~/marsyas
   ```

2. Create a build directory within, which will contain configuration options and compiled Marsyas libraries and programs. Navigate to the build directory:

   ```
   mkdir build
   cd build
   ```

3. Run CMake, passing it the Marsyas source directory as argument:

   ```
   cmake ..
   ```

CMake should now have generated a number of new files in the build directory, including a file named "Makefile" which allows you to compile Marsyas.

If you would like to use CMake options to enable and disable specific features, please see instructions in Section 2.3 [Configuring with CMake], page 21.

2.2.1.3 Compile Marsyas using 'make'

Still in the build directory, use the 'make' command to compile Marsyas.

```
make
```

Geeky note: If your CPU has multiple cores (it is capable of running multiple threads in parallel), you can shorten the compilation time by running several instances of 'make' in parallel by using the '-j' option followed by the number of instances. The example below runs 3 parallel 'make' instances:

```
make -j3
```

You could also compile Marsyas in Debug mode, which would help developers discover bugs in case you run into troubles when using Marsyas. However, Marsyas will run significantly slower when compiled in Debug mode.

To compile in Debug mode, you need to first use 'cmake' to change a CMake option named CMAKE_BUILD_TYPE, and then run 'make'. Please mind the "." at the end of the first command, to indicate the current directory:

```
cmake -DCMAKE_BUILD_TYPE=Debug .
make -j3
```

2.2.1.4 Marsyas usage and system-wide installation

After compiling, you should have several Marsyas programs in the `build/bin` subdirectory and the Marsyas library in the `build/lib` subdirectory.

For example, assuming that you are in the `build` directory, you can run the `sfplay` program to play an uncompressed sound file like this:

```
./bin/sfplay my_sound_file.wav
```

If you want to make the programs and the library available outside of your build directory, you should use the following command to install all parts of Marsyas to appropriate system locations:

```
make install
```

By default, the above command will install programs and libraries under the `/usr/local` prefix. You can change that by setting the CMake option CMAKE_INSTALL_PREFIX to the desired prefix before installation:

```
cmake -DCMAKE_INSTALL_PREFIX=~/marsyas-install .
make install
```

2.2.2 Building latest Marsyas on Mac OS X

These instructions are for building the latest development version of Marsyas. It is assumed that you have already obtained the source code. If not, please consult Section 2.1 [Get Marsyas sources], page 6 for instructions.

2.2.2.1 Install prerequisits

- CMake

  Marsyas uses CMake to configure and guide its building process.

  Download CMake from the following link and install it: http://www.cmake.org/cmake/resources/softw

  At some point in installation you may be asked whether you want to install links to CMake to system locations - choose "yes": this will allow you to run CMake from the Terminal.

- Xcode

  Xcode development environment provides the Clang compiler required to compile Marsyas. If you want to use all Marsyas features, you need a compiler with a decent C++11 support. Hence, please install the latest Xcode version from Apple: https://developer.apple.com/xcode/

  Just as with CMake above, confirm installation of compilation tools to system locations, so as to make them available for use in the Terminal.

2.2.2.2 Configure Marsyas using CMake

1. Start up the Terminal application.
2. Navigate to the top-level directory of Marsyas sources. For example:

   ```
   cd ~/marsyas
   ```

3. Create a build directory within, which will contain configuration options and compiled Marsyas libraries and programs. Navigate to the build directory:

   ```
   mkdir build
   cd build
   ```

4. Run CMake, passing it the Marsyas source directory as argument:

   ```
   cmake ..
   ```

CMake should now have generated a number of new files in the build directory, including a file named "Makefile" which allows you to compile Marsyas.

If you would like to use CMake options to enable and disable specific features, please see instructions in Section 2.3 [Configuring with CMake], page 21.

2.2.2.3 Compile Marsyas using 'make'

Still in the build directory, use the 'make' command to compile Marsyas.

```
make
```

Geeky note: If your CPU has multiple cores (it is capable of running multiple threads in parallel), you can shorten the compilation time by running several instances of 'make' in parallel by using the '-j' option followed by the number of instances. The example below runs 3 parallel 'make' instances:

```
make -j3
```

You could also compile Marsyas in Debug mode, which would help developers discover bugs in case you run into troubles when using Marsyas. However, Marsyas will run significantly slower when compiled in Debug mode.

To compile in Debug mode, you need to first use 'cmake' to change a CMake option named CMAKE_BUILD_TYPE, and then run 'make'. Please mind the "." at the end of the first command, to indicate the current directory:

```
cmake -DCMAKE_BUILD_TYPE=Debug .
make -j3
```

2.2.2.4 Marsyas usage and system-wide installation

After compiling, you should have several Marsyas programs in the `build/bin` subdirectory and the Marsyas library in the `build/lib` subdirectory.

For example, assuming that you are in the `build` directory, you can run the `sfplay` program to play an uncompressed sound file like this:

```
./bin/sfplay my_sound_file.wav
```

If you want to make the programs and the library available outside of your build directory, you should use the following command to install all parts of Marsyas to appropriate system locations:

```
make install
```

By default, the above command will install programs and libraries under the `/usr/local` prefix. You can change that by setting the CMake option CMAKE_INSTALL_PREFIX to the desired prefix before installation:

```
cmake -DCMAKE_INSTALL_PREFIX=~/marsyas-install .
make install
```

2.2.3 Building latest Marsyas on Windows

These instructions are for building the latest development version of Marsyas. It is assumed that you have already obtained the source code. If not, please consult Section 2.1 [Get Marsyas sources], page 6 for instructions.

2.2.3.1 Install prerequisits

- CMake

  Marsyas uses CMake to configure and guide its building process.

  Download CMake Windows installer from: http://www.cmake.org/cmake/resources/software.html

  Run the installer.

  At some point you may be asked whether you want to make CMake available to the whole system - choose "yes": this will allow you to run CMake from Visual Studio Command Prompt.

- Microsoft Visual Studio

  If you want to use all features of Marsyas, you need a compiler with a decent C++11 support. Hence, please install at least Visual Studio 2012 or newer version. The free 'Express' version is sufficient.

  Download and install Visual Studio from: http://www.microsoft.com/visualstudio/eng/downloads

2.2.3.2 Choose correct Visual Studio Command Prompt

Visual Studio versions which support compilation of 64 bit programs provide you with several alternative versions of Command Prompt, intended specifically for compiling 32 bit or 64 bit programs:

- Visual Studio Command Prompt ...x86 - intended for 32 bit compilation.
- Visual Studio Command Prompt ...x64 - intended for 64 bit compilation.

Search the Start menu for the Command Prompt that matches your intended compilation mode.

Geeky details: The different Command Prompt versions automatically populate the PATH environment variable so as to provide appropriate versions of compilation tools and libraries for the intended compilation mode.

2.2.3.3 Configure Marsyas using CMake

1. Start the appropriate Visual Studio Command Prompt, as described above.
2. Navigate to the top-level directory of Marsyas sources. Use the 'cd' command, passing it the path to Marsyas as argument. For example:

   ```
   cd C:\Users\John\marsyas
   ```
3. Create a build directory within, which will contain configuration options and compiled Marsyas libraries and programs. Navigate to the build directory:

   ```
   mkdir build
   cd build
   ```
4. Start CMake GUI, passing it the source directory (one level up: `".."`) as argument:

   ```
   cmake-gui ..
   ```
5. In CMake GUI: Click `"Configure"` to start auto-configuration of Marsyas for Windows.
6. In CMake GUI: A dialog will ask you to choose the desired `"generator"`. From the drop-down menu, choose the version of Visual Studio that you have installed. Click `"Finish"`.

   Note: `"Visual Studio 11"` corresponds to Visual Studio 2012. If you intend to compile in 64 bit mode, choose the generator which contains `"Win64"` in its name.
7. In CMake GUI: Click `"Generate"` to generate a Visual Studio solution for desired configuration. Exit CMake GUI.

CMake should now have generated a number of new files in the build directory, including a Visual Studio solution file named `"marsyas.sln"`.

If you would like to use CMake options to enable and disable specific features, please see instructions in Section 2.3 [Configuring with CMake], page 21.

2.2.3.4 Compile Marsyas using Visual Studio

Still in the build directory, use 'msbuild' (the Visual Studio build tool) to compile Marsyas in Release mode:

```
msbuild /p:Configuration=Release marsyas.sln
```

You could also compile Marsyas in Debug mode, which would help developers discover bugs in case you run into troubles when using Marsyas. However, Marsyas will run significantly slower when compiled in Debug mode:

```
msbuild /p:Configuration=Debug marsyas.sln
```

After compiling, you should have Marsyas programs in the 'bin' subdirectory and the Marsyas library in the 'lib' subdirectory.

2.2.4 Compiling on MacOS X 10.6 Snow Leopard

These steps were used to install Marsyas on a fresh MacOS X 10.6.2 Snow Leopard desktop. on January 24, 2010. These instructions below can be used to compile the DEFAULT Marsyas configuration under Mac OSX Snow Leopard (10.6.2):

1. Download (or find it on your Snow Leopard MacOSX install CD) and install XCode Developer Tools (it's freely provided by Apple, but you may need to register and get a developer account online - see here: http://developer.apple.com/tools/xcode/) in your machine. If you download XCode from the Apple website (make sure you download the Snow Leopard version!) you may get a more updated version of the Developer Tools than the one provided in the OSX instal disk, but any version should work with Marsyas.

2. Download and install the CMake 2.8.0 (or later) binary for Apple OSX in http://www.cmake.org. During install, CMake will ask you if it should install the "command line links" (i.e. links to ccmake, cmake, etc.), which we recommend you to do (you'll need administrator rights to install them, so make sure you are logged in as an administrator).

3. Apple Mac OSX 10.6.2 Snow Leopard already includes SVN (http://subversion.tigris.org/) as a command line tool (you can run "which svn" at the OSX terminal to check that - it should reply with "/usr/bin/svn", which is the path to the SVN binary installed in your system). In case you prefer using a GUI for SVN in Mac OSX, you can try the free SvnX (http://code.google.com/p/svnx/).

4. In case you want to check out the SVN version of Marsyas, just do a SVN checkout from `https://svn.code.sf.net/p/marsyas/code/trunk` - you will get the most updated version (but not necessarily the most stable version ;-) - for that, get the latest tarbal from the Marsyas SourceForge website: http://marsyas.sf.net).

5. Launch CMake (using the CMake GUI that you can find in the OSX Applications folder using Finder, or using the command line tool ccmake in the OSX terminal), and specify the source and build directory you want to use for Marsyas. For example, if you have the Marsyas code directory in you home directory (e.g. ~/Marsyas), you should point your source path to ~/Marsyas/src and the build directory to somewhere like ~/Marsyas/build/ (you may have to create the ~/Marsyas/build directory before hand in case CMake GUI or ccmake does not create it for you. Either using CMake GUI or ccmake in the terminal, just select the option to "configure" using the provided options (i.e. using the default Marsyas build configuration). CMake GUI will ask youif you want to create the "~/Marsyas/build" directory in case you haven't dreated it yourself,

and then ask you to select a generator from a drop down list - here you can ask CMake to create an UNIX makefile, a XCode project, among other options, but for now just use the provided options and click "done" (you can also specify a generator when using the command line tool ccmake - check its documentation). CMake will then look into your system for libraries, compilers, and present you with a list of the configuration options it set up for you. Just keep hitting "configure" until all options are grey (in the CMake GUI) or not signaled with a * (if using ccmake on the terminal), and then just select "generate". This will generate a UNIX makefile with the default build option for Marsyas. You can now close CMake.

6. Open a terminal window in OSX (in case you haven't yet) and go to ~/Marsyas/build directory and just run "make". Marsyas compilation will start, and you will be able to see the compilation evolution in the compiler messages in the terminal. When finished, you will be able to find the compiled binaries in the ~/Marsyas/build/bin directory. If you want to install the Marsyas command line tools (e.g. sfplay, bextract, sfplugin, etc.) into your system, just run "sudo make install" and those tools will be installed in the appropriate system folders in your machine (you will need administrator privileges to use "sudo").

7. As a test to see see Marsyas in action, just go to ~/Marsyas/build/bin and run:

   $ ./sfplay someAudioFile.wav

   where $ is the command line prompt, and someAudioFile.wav is the path to some audio file you have in your machine. It should start playing it, so bump up your computer volume and enjoy (hit CTRL-C to stop it). If you installed Marsyas in your system (by means of "sudo make install") you can just call splay (or any other Marsyas command line tool) from anywhere (i.e. from any directory) in the terminal.

Using additional and optional libraries

Marsyas can also be compiled using some optional libraries for things like MP3 file reading/writing, GUI, python bindings, among other options. You define these building options in CMake, during the configuration/generating step (see step 5. above). You can manually install most of the needed libraries in OSX, but it requires some expertise in dealing with library dependencies, library configuration, building and install.

Using MACPORTS

A more convenient (and recommended) way to install such libraries in OSX is to use a project called MacPorts (http://www.macports.org/). MacPorts provides a large set of libraries and applications available for UNIX customized and ready to use in OSX, and deals with dependencies in projects so you don't have to worry about them.

So, we recommend you install MacPorts in your machine following the steps below:

1. Download MacPorts 1.8.2 (or later) for Snow Leopard at http://www.macports.org
2. Install MacPorts (detailed info can be found at http://www.macports.org/install.php)
3. MacPorts adds its own paths (i.e. /opt/local/bin and /opt/local/sbin) to your PATH during install (MacPorts is installed in /opt/). That is done in ~/.profile, or in case you

have it in you home directory, in .bash_profile. In order to make those paths into the current command line session, you must do "source .profile" (or "source .bash_profile"), or just close and open a new terminal window.

4. A final step is to make sure MacPorts is updated. Just do "sudo port -v selfupdate" at the terminal. That's it.
5. There is a nice GUI front end for MacPorts named Porticus (http://porticus.alittledrop.com/), in case you don't want to lean MacPorts syntax for the command line.

MP3 reading support (libmad)

1. Marsyas makes use of the external MAD library (http://www.underbit.com/products/mad/) for MP3 reading support. You can find it in MacPorts as "libmad". Just install it.
2. To build Marsyas with libMAD support, run CMake (remember to select a different build directory - e.g. ~/Marsyas/build_with_MAD) and run "configure", and then select (or turn ON) the WITH_MAD option and run "configure" again. After the second "configure", CMake should present you the paths to the libmad libraries, like this:

   ```
   mad_INCLUDE_DIR /opt/local/include
   mad_LIBRARY /opt/local/lib/libmad.dylib
   ```

   In case it does not, you can always change them manually in CMake so they correspond to the above paths.
3. Execute "configure" once again, and when the "generate" option becomes available in CMake, execute it and the corresponding makefile for building Marsyas with MAD support will be generated. You can then test if things went well by doing:

   ```
   $ cd ~/Marsyas/build_with_MAD/bin
   $  ./sfplay someMP3file.mp3
   ```
4. You can also install this new Marsyas build with MP3 reading support into your system by doing "sudo make install".

MP3 writing support (LAME)

1. Marsyas makes use of the external LAME library (http://lame.sourceforge.net/) for MP3 writing support. You can find it in MacPorts as "lame". Just install it.
2. To build Marsyas with LAME support, run CMake (remember to select a different build directory - e.g. ~/Marsyas/build_with_LAME) and run "configure", and then select (or turn ON) the WITH_LAME option and run "configure" again. After the second "configure", CMake should present you the paths to the lame libraries, like this:

   ```
   lame_LIBRARY /opt/local/lib/libmp3lame.dylib
   ```

```
lame_INCLUDE_DIR /opt/local/include
```

In case it does not, you can always change them manually in CMake so they correspond to the above paths.

Nokia Qt SUPPORT

Marsyas uses the Qt Toolkit (http://qt.nokia.com/) for its GUI applications (e.g. MarPlayer, MarPhaseVocoder, etc.). In order to build Marsyas with Qt support, follow the next steps:

1. Download and install Qt 4.6.1 64 bits for cocoa (there are other Qt versions for carbon and 32 bit, but for Snow Leopard, the 64 bit cocoa SDK version should be the one to get - get the LGPL version here: http://get.qt.nokia.com/qt/source/qt-mac-cocoa-opensource-4.6.1.dmg)
2. 2. In CMake, create a new binary folder (e.g. ~/Marsyas/build_with_Qt), select the option WITH_QT, and CMake should find the Qt libraries in your system and generate a makefile that you can use to build Marsyas with Qt support. In this case, some new applications will get built in ~/Marsyas/build_with_Qt/bin - just try MarPhaseVocoder or MarPlayer, which provide a GUI.

Python bindings

The following optional steps can be used to setup the Marsyas python bindings using SWIG and setup the NumPy,SciPy,Matlplotlib environment, using MacPorts. This setup is a great free substitute for MATLAB and integrates very nicely with Marsyas all in a Python environment.

1. Install python26 from MacPorts.
2. Install swig-python from MacPorts.
3. Install py26-numpy, py26-matplotlib, py26-ipython all from MacPorts.
4. You should now select which python to use by default in your system (Apple already provides and oldish version of python in OSX 10.6.2, but we must set the system to use the MacPorts python instead. For that, we must use the python_select command line tool, that MacPorts also installs with python. First, list the pythons installs in your system using:

   ```
   $  python_select -l
   Available versions:
   current none python26 python26-apple
   ```

   To see which one is the current one do:

   ```
   $ python_select -s
   python26-apple
   ```

   In case it's not the MacPorts installed python (i.e. python26), just select it:

```
$ python_select python26
```

5.

If everything has worked so far you should be able to run the examples shown in the matplotlib webpage in your python or ipython environment.

6.

You can now start CMake and enable WITH_SWIG and reconfigure Marsyas (use a build_with_SWIG binary folder). If SWIG is found correctly (as it should) the corresponding build files are updated and you can then do "make" at the command line in ~/Marsyas/build_with_SWIG.

There is a little trick regarding CMake GUI and the definition of the PATH environment variable for GUI applications in OSX. If you run ccmake from the terminal, it should be able to find the MacPorts python and SWIG libraries (because ccmake uses the PATH env. var, that is set at .profile or .bash_profile - see above). However, CMake GUI, because it's a GUI app launched from the Finder knows nothing about the terminal PATH env. var. In OSX, GUI apps look for environment variables in a special environment.plist file in ~/.MacOSX (see http://developer.apple.com/mac/library/qa/qa2001/qa1067.html).

In case you don't have the .MacOSX folder in your home directory (you may not have it), you'll need to create a folder named ".MacOSX" in your home folder (note that since there is a dot at the beginning of the folder name, it won't be visible to a simple 'ls' - you would need to use 'ls -a') and then create a file named "environment.plist" in the ~/.MacOSX folder. Put the definitions of the environment variables that you want defined for all GUI programs in that plist file using a format like the following example:

```
PATH   String  /opt/local/bin:/opt/local/sbin:/usr/bin:/bin:/usr/sbin:/sbin:/usr/lo
```

(You can use the "Property List Editor" that comes with Apple's developer tools or a 3rd-party plist editor - or just an ordinary text editor if you are careful to get the format correct.)

You must then log out and log in again, because ~/.MacOSX/environment.plist only gets read during login. So do not forget this step!

Now CMake GUI will know about the MacPorts directory (i.e. /opt) and will be able to find the pythonand SWIG libs from MacPorts.

7.

After compilation finished, make sure you are at ~/Marsyas/build_with_SWIG and run "sudo make install". This will install Marsyas into your system, as well as the Marsyas python module at the right places so python can import it.

8.

You should now be ready to run Marsyas inside Python. Try launching a Python interactive interpreter by typing "python" at the terminal and then trying:

```
import marsyas
```

If it works you have succeeded and can try the various python Marsyas scripts in src/swig/python.

A final note

In the above sections we present a way to configure the Marsyas building for a single optional library, but you can select a combination of any of them at the same time. That's why it may be a good idea to have a ~/Marsyas/builds directory where you then create spear directories for all the different building configurations, e.g.:

```
~/Marsyas/builds/make_default
~/Marsyas/builds/make_with_MAD_LAME_SWIG
~/Marsyas/builds/make_with_Qt
~/Marsyas/builds/XCode_with_MAD_Qt
etc...
```

2.2.5 Compiling on Ubuntu 11.04 - Natty Narwhal (2011)

These steps were used to install in a fresh install of Ubuntu (11.04 - Natty Narwhal - 32 bits) on September 2011. All the command were typed in a terminal but it should be straightforward to use the synaptic package manager instead of apt-get. The text in parentheses after each command is a simple explanation and should not be typed in the terminal.

1. sudo apt-get install subversion (install the subversion version control system)
2. svn co https://svn.code.sf.net/p/marsyas/code/trunk my-marsyas-dir
3. cd marsyas
4. mkdir build
5. cd build
6. sudo apt-get install cmake (the regular cmake)
7. sudo apt-get install cmake-curses-gui (the curses gui invoked by ccmake)
8. sudo apt-get install libasound2-dev (alsa development library and headers)
9. ccmake ../src
10. Press [c] to configure and [g] to generate the Makefile and exit
11. make (or make -j k where k is the number of cores of your computer + 1)
12. cd bin
13. ./helloWorld (you should hear a sine wave played)

Additional instructions for creating the Marsyas python bindings through Swig as well as installing matplotlib.

1. sudo apt-get install swig (the swig interface generator for the Python bindings)
2. sudo apt-get install python-dev (the header files for Python)
3. sudo apt-get install python-matplotlib (the matlab-like plotting capabilities for Python)
4. sudo apt-get install ipython
5. cd ..
6. ccmake ../src
7. enable the WITH_SWIG option
8. make
9. sudo ldconfig /usr/local/lib (add /usr/local/lib to the location searched for libraries)

10. sudo make install (install the Marsyas python bindings)

Now you are ready to test that everything works:

1. python windowing.py
2. ipython –pylab windowing.py

You should see a plot that has been created by passing data through a Marsyas processing network. Congratulations - you now can work with Marsyas and matplotlib a powerful combo.

If you decide to add PNG support you need the following additional steps:

1. sudo apt-get install libfreetype6-dev
2. enable the WITH_PNG option
3. make
4. sudo make install

2.2.6 Compiling on Ubuntu

These steps were used to install Marsyas from a fresh install of Ubuntu on August 25, 2008.

1. Open a terminal and type the following commands:
2. sudo apt-get install build-essential
3. sudo apt-get install subversion
4. sudo apt-get install libasound2-dev
5. Either download the Linux binary distribution of cmake 2.6 or compile and install the source code from http://www.cmake.org
6. Test that you can run cmake and svn from the command-line
7. Follow the basic installation instructions.

2.2.7 Compiling with Visual Studio Express 2008 on Windows XP

These instructions were used to compile Marsyas using the Microsoft Visual Studio 2008 Express IDE in Windows XP

1. Download and install Microsoft Visual C++ Studio Express 2008 from http://www.microsoft.com/express/vc. (you might need to install Microsoft Silverlight to view the webpage).
2. Download and install Microsoft Direct X SDK from http://msdn.microsoft.com/en-us/directx/aa937 These instructions worked with installing Direct X August 2008 (DXSDK_Aug08.exe).
3. Download and install the CMake 2.6 binary using the Win32 installer from http://www.cmake.org.
4. Downlaod and install the Tortoise SVN GUI client from http://tortoisesvn.tigris.org/.
5. The Tortoise client is integrated with Windows so right clik on your desktop and checkout Marsyas as described in the basic installation instructions.
6. Launch cmake and specify the source and build directory you want to use for Marsyas for example c:\marsyas\src and c:\marsyas\build. Press configure until all the options are grey and then click ok

7. Go to the c:\marsyas\build directory where you will find a Visual Studio project file named marsyas that you can open with the Visual Studio 9 Express IDE to compile Marsyas.
8. The binaries are created in build\bin\Release or build\bin\Debug depending on which build configuration is used.

The following optional steps can be used to setup Marsyas to be able to read MP3 files through libMAD an external MPEG audio decoder.

1. Download the libMAD library source code from http://www.underbit.com/products/mad/.
2. In the msvc++ folder of the libMAD package there is a Visual Studio project file. Double-click on that and build the library. This should generate a debug (or release) folder depending on your configuration in msvc++ which contains a libmad.lib file.
3. Using cmake, set WITH_MAD to ON and click configure. The CMake configuration of Marsyas needs to determine the location of the libmad.lib library file and the location of the mad.h include file. You can either add the corresponding directory to your PATH or you can manually point the mad_INCLUDE_DIR to the msvc++ folder directory and mad_LIBRARY to the libmad.lib file generated earlier. Finally click configure and okay.

The following optional steps can be used to setup the Marsyas python bindings using SWIG and setup the NumPy,SciPy,Matlplotlib environment in Windows. It is a great free substitute for MATLAB and integrates very nicely with Marsyas all in a Python environment.

1. Download and install Python 2.5 (IMPORTANT: NumPy and SciPy don't yet work with the more recent 2.6) from http://www.python.org/download/. The easiest way is to just use the .msi installer.
2. Download and install numpy superpack for python 2.5 from the sourceforge webpage of numpy http://sourceforge.net/project/showfiles.php?group_id=1369&package_id=175103.
3. Download and install SciPy 2.5. Follow the dowload link at http://www.scipy.org.
4. Download and install ipython from http://ipython.scipy.org/dist/ which is an enhanced interactive python interpreter that can add MATLAB like plot capabilities to Python.
5. Download and install matplotlib from http://matplotlib.sourceforge.net/. This adds plot capabilities.
6. If everything has worked so far you should be able to run the examples shown in the matplotlib webpage in your python or ipython environment.
7. Now you it is time to install the SWIG python bindings. First you will need to download and install SWIG from http://www.swig.org/. The easiest method is to just unzip SWIG-1.3.36.zip somewhere in your hard drive. I chooce c:\swig. Inside this new directory there is a bin directory with the swig executable. This directory needs to be added to your system PATH environment variable so that CMake can find the SWIG installation.
8. We are almost there. Start CMake and enable WITH_SWIG and reconfigure Marsyas. If swig is found correctly the corresponding build files are updated. To install the

marsyas python module you will need to run the INSTALL target of the Marsyas solution inside Visual Studio. This target can be found on the left side of the Visual Studio IDE. There is a tree list list of different kinds of things you can compile. One of them is the INSTALL target. This will build and install all of Marysas including the Python bindings into your system.

9. There is one final little step. For some reason I have not figured out the marsyas module is compiled as _marsyas.dll and gets copies to the default location for installed Python modules. You will need to rename this to _marsyas.pyc for things to work for some versions of Python. It seems that for more recent versions of Python it needs to be called _marsyas.pyd.
10. You should now be ready to run Marsyas inside Python. Try launching a Python interactive interpreter and trying:

    ```
    import marsyas
    ```

    If it works you have succeeded and can try the various python Marsyas scripts in src/swig/python.

2.2.8 Compiling with MinGW on Windows XP

1. Download and install MinGW from http://www.mingw.org/old/download.shtml
2. Download and install Microsoft Direct X SDK from http://msdn.microsoft.com/en-us/directx/aa937
   These instructions worked with installing Direct X August 2008 (DXSDK_Aug08.exe).
3. Download and install the CMake 2.6 binary using the Win32 installer from http://www.cmake.org.
4. Downlaod and install the Tortoise SVN GUI client from http://tortoisesvn.tigris.org/.
5. The Tortoise client is integrated with Windows so right click on your desktop and checkout Marsyas as described in the basic installation instructions.
6. Launch cmake and choose the install type to be "MinGW Makefiles"
7. Specify the source and build directory you want CMake to use for Marsyas for example if marsyas has been installed in c:\marsyas these would be c:\marsyas\src and c:\marsyas\build.
8. Press configure until all the options are grey and then click ok
9. Go to the c:\marsyas\build directory and type "mingw32-make"
10. The binaries are created in build\bin\release or build\bin\debug.

Python: Installed by default on Linux and MacOS X machines; Windows users may install it from this site. Marsyas contains some very useful scripts which are written in Python.

2.2.9 MacOS X additional notes

These random notes might be useful for OS X users.

Qt

On MacOS X and when using Qt-4.3.0 or higher, you must add:

```
export QMAKESPEC=/usr/local/Qt4.3/mkspecs/macx-g++/
```

to your '`~/.profile`' or '`~/.bash_profile`' and then close (and re-open) your terminal window before compiling Marsyas.

Recording audio

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

```
pnet->addMarSystem(mng.create("AudioSource", "srcRec"));
pnet->updctrl("mrs_real/israte", 44100.0);
pnet->updctrl("AudioSource/srcRec/mrs_bool/initAudio", true);
```

MATLAB

These instructions have been tested on OS X 10.5.8 and MATLAB_R2009a.

1. MATLAB and X11 for OS X must be installed
2. The naming conventions of MATLAB are a messy so the CMake configuration assumes that either the MATLAB environment variable is set or looks for MATLAB in /Applications/MATLAB_R2009a.app
3. Compile Marsyas and enable the WITH_MATLAB option. If MATLAB can not be located you will get an error message. Try setting the MATLAB enviornment variable and if that does not work then attempt to set the paths to the MATLAB-related variables manually.
4. You will need to edit your .bash_profile with the following lines:

```
export MATLAB="/Applications/MATLAB_R2009a.app"
export DYLD_LIBRARY_PATH=$MATLAB/bin/maci/:$MATLAB/sys/os/maci/:$DYLD_LIBRARY_PATH
```

5. try out mudbox -t MATLABengine. MATLAB should open and various benchmarks and communication between Marsyas and MATLAB should happen. If not email the mailing lists for some help.

2.3 Configuring with CMake

Marsyas uses CMake to configure and guide its building process. CMake allows for automatic configuration according to the current platform, as well as user-configurable building options. It allows you to enable and disable specific Marsyas features.

After configuration, CMake creates files that contain instructions for the next step of the building process. This may be:

- Makefiles on Linux, Mac OS X or MinGW
- Xcode projects on Mac OS X
- MS Visual Studio solutions on Windows.

2.3.1 Running CMake

Note: The following instructions assume that you have done the prerequisite steps described in Section 2.2 [Step-by-step building instructions], page 6:

1. Obtained Marsyas sources
2. Installed CMake and other prerequisite software
3. Created a `build` directory within the top-level Marsyas directory and navigated your command-line to the `build` directory

There are several ways that configuration using CMake can be performed.

2.3.1.1 Using CMake GUI

Note: On Linux, you will need to install an extra package for the CMake GUI. On Debian/Ubuntu the package name is `cmake-qt-gui`.

You can start CMake GUI either graphically via the list of applications installed on your system, or by using the 'cmake-gui' command in the Linux console, OS X Terminal, or Windows Command Prompt.

- If you start it the graphical way, you will need to set the Marsyas source and build directories manually in the CMake GUI
- If you start it using the 'cmake-gui' command, you can pass it the Marsyas source directory as argument, and the current directory will be used as the build directory. For example, assuming that you are in the `build` directory:

```
cmake-gui ..
```

CMake GUI contains a list of available options: the left column lists option names, and the right column their editable values. By moving the mouse cursor over an option name, its description will pop up. To perform configuration, follow the steps below:

1. If Marsyas has not been configured before, the list of options will be empty. Click "Configure" to do the initial auto-configuration.
2. Modify options as desired, and click "Configure" again.
3. Once you are satisfied with options, click "Generate" to apply them.
4. You can now exit CMake GUI and proceed with the compilation step of the building process (see Section 2.2 [Step-by-step building instructions], page 6).

2.3.1.2 Entirely on command-line

You can also perform CMake configuration entirely on command-line. Options are controlled by using the 'cmake' command and passing it arguments of the form "-D" + [option name] + "=" + [value]. Values can be paths to directories and files, or simply "ON" and "OFF" (or equivalently, "TRUE" and "FALSE"). Multiple options can be changed using the same command.

For example, assuming that you are in the `build` directory:

```
cmake -DMARSYAS_AUDIOIO=OFF -DMARSYAS_TESTS=ON ..
```

After invoking the `cmake` command one or multiple times to apply desired options, you can proceed with the compilation step of the building process (see Section 2.2 [Step-by-step building instructions], page 6)

2.3.1.3 More alternatives

There are more alternative ways to use CMake. Please read detailed instructions on the CMake website:

- General instructions
- Command-line usage

2.3.2 Most prominent options

The following is a list of most important and commonly used options provided either by Marsyas or CMake itself.

2.3.2.1 Input / output

`MARSYAS_AUDIOIO`
: This enables audio input/output. Requires DirectX on Windows and either JACK, ALSA or OSS on Linux. MacOS X audio support is built-in with the basic developer tools.

  Note: This option requires C++11 support - in other words, the WITH_CPP11 option must also be enabled (described below).

`MARSYAS_MIDIIO`
: This enables midi input/output. Requires DirectX on Windows and either ALSA or OSS on Linux. MacOS X audio support is built-in with the basic developer tools.

Note: Audio and MIDI IO support also depend on WITH_JACK, WITH_ALSA, and WITH_OSS options, described below.

2.3.2.2 Code messages and optional portions

`MARSYAS_ASSERT`
: Turns on assertions.

`MARSYAS_PROFILING`
: Turns on profiling.

`MARSYAS_DEBUG`
: Turns on debugging info (large performance penalty).

`DISTRIBUTED`
: (*advanced* option) experimental code for distributed systems.

2.3.2.3 Message logging

These are *advanced* options.

`MARSYAS_LOG_WARNINGS`
`MARSYAS_LOG_DEBUGS`
`MARSYAS_LOG_DIAGNOSTICS`
`MARSYAS_LOG2FILE`
`MARSYAS_LOG2STDOUT`
`MARSYAS_LOG2GUI`

2.3.2.4 Testing

`MARSYAS_TESTS`
: Build Marsyas tests, so they can be run using `make test`.

2.3.2.5 Optional software

All of these options require additional software to be **installed and properly configured**.

`WITH_CPP11`
: Enables compilation in C++11 mode. If disabled, Marsyas will be compiled with limited functionality. Specifically, the audio IO and multi-threading support require this option to be enabled.

This option requires a compiler with adequate C++11 support. Minimum required compiler versions are ensured by CMake, and reported if not satisfied.

`WITH_MAD` mp3 audio decoding with LibMAD

`WITH_VORBIS`
ogg vorbis audio decoding with libvorbis - it requires

`WITH_MATLAB`
Builds the MATLAB engine interface.

`WITH_SWIG`
Builds SWIG bindings. This option enables the following sub-options: WITH_SWIG_PYTHON, WITH_SWIG_JAVA, WITH_SWIG_LUA, and WITH_SWIG_RUBY.

`WITH_SWIG_PYTHON`
Use Swig to generate Python bindings

`WITH_SWIG_JAVA`
Use Swig to generate Java bindings

`WITH_SWIG_LUA`
Use Swig to generate Lua bindings

`WITH_SWIG_RUBY`
Use Swig to generate Ruby bindings

`WITH_QT5` Builds the Qt5 GUI applications. Most Marsyas GUI applications are of this type. Requires Qt 5.0 or higher.

`WITH_QT` Builds the Qt4 GUI applications. There are only a few unmaintained Marsyas GUI applications of this type, preserved mostly for inspiration. Requires Qt 4.2.3 or higher.

`WITH_VAMP`
Build plugins for Vamp (see Section 9.5.6 [SonicVisualiser Vamp Plugins], page 99 for more information).

`WITH_GSTREAMER`
Use GStreamer as an audio source

Linux-specific:

`WITH_JACK`
Enables audio IO using JACK, if available.

`WITH_ALSA`
Enables audio and MIDI IO using ALSA, if available.

`WITH_OSS` Enables audio and MIDI IO using OSS, if available.

2.3.2.6 Documentation

`MARSYAS_DOCUMENTATION_ONLY`
If enabled, only build documentation, not program sources. This allows to build documentation without even the presence of a compiler or any requirements related to program code.

2.4 Post-install setup

2.4.1 Vim editor support

A syntax file for vim color highlighting is in '`misc/marsyas.vim`'. To use this file, copy it to '`$HOME/.vim/syntax`' add the following lines to '`$HOME/.vim/filetype.vim`':

```
if exists("did_load_filetypes")
  finish
endif
augroup filetypedetect
  au! BufNewFile,BufRead *.cpp          setf marsyas
  au! BufNewFile,BufRead *.h            setf marsyas
augroup END
```

2.4.2 Datasets

Useful datasets:

- marsyas-coffee: data set used for large black-box tests in Marsyas. (NOW DEFUNCT, MIGHT POSSIBLY BE USED AGAIN LATER)
- `http://marsyas.info/download/data_sets`: large data sets.

2.5 Structure of distribution

Marsyas is primarily targeting researchers and software developers who want to advance the existing knowledge within Marsyas' area of application and in turn develop Marsyas further. For that purpose, familiarity with the structure of the source code and file system is important.

The root directory contains the following files:

- **AUTHORS, COPYING, README, TODO:** these files are self-explanatory.

In addition, there are the following subdirectories:

- **src/** All the C++ source files.
- **src/marsyas/** The Marsyas framework and modules (MarSystems). This is compiled into a library for use in other executable programs.
- **src/marsyas/core** Framework core.
- **src/marsyas/realtime** Facilities for realtime MarSystem execution.
- **src/marsyas/debug** Facilities for dataflow debugging.
- **src/marsyas/marsystems** Concrete MarSystems.
- **src/marsyas/optional** Optionally-compiled MarSystems that depend on third-party libraries.
- **src/apps/** Various command-line applications.
- **src/Qt4Apps/** GUI applications using Qt 4. Most of those have been ported to use Qt 5.
- **src/qt5apps/** GUI applications using Qt 5. All future GUI applications using Qt shall be developed here.
- **src/tests/unit_tests/** Tests for individual framework modules (MarSystems).

- **src/tests/black-box/** Tests for executable programs.
- **doc/** Source files for documentation (which you are currently reading).
- **scripts/** Convenient scripts to help programming with Marsyas.
- **scripts/MATLAB/** MATLAB scripts.

3 Tour

This chapter assumes that you have succesfully compiled and installed Marsyas as described in Chapter 2 [Source installation], page 6. The compiling configuration should be Release otherwise there are noticeable artifacts in the audio playback. The goal of this chapter is to provide a quick tour of various tools included with the Marsyas distribution that can be used to showcase the capabilities of the framework rather than a complete exposition which is provided in chapter Chapter 4 [Available tools], page 31.

In order to execute the examples described in this section you will need collections of music tracks in an audio format supported by Marsyas. The chapter assumes that the collections provided in `http://marsyas.info/download/data_sets` have been downloaded and placed in a subdirectory named `audio` of the Marsyas distribution. Here is a possible sequence of commands to obtain the datasets in a unix like system (Linux, OSX or Cygwin).

```
cd MY-MARSYAS-DIR
mkdir audio
cd audio
wget http://opihi.cs.uvic.ca/sound/genres.tar.gz
wget http://opihi.cs.uvic.ca/sound/music_speech.tar.gz
tar -zxvf genres.tar.gz
tar -zxvf music_speech.tar.gz
```

This should create directories containing the audio files of the collections for example on my OS X laptop for a particular release of Marsyas the following audio track is available: ‘`/Users/gtzan/src/cxx/marsyas-0.2.20/audio/music_speech/music_wav/nearhou.wav`’

3.1 Command-line tools

In this section a few examples of how Marsyas can be used for various audio processing tasks are provided. Detailed documentation about the various command-line tools is provided in Chapter Chapter 4 [Available tools], page 31.

First we will explore audio playback, plugins, and real-time running audio classification in music/speech.

The following commands can be used to create two collections music.mf and speech.mf each one with 60 30-second audio clips of music and speech respectively (small modifications like changing the directory separator character are required for Windows).

```
cd MY_MARSYAS_DIR/build/bin
mkcollection -c music.mf ../../music_speech/music_wav
mkcollection -c speech.mf ../../music_speech/speech_wav
```

The following commands can be used to have a quick preview of the two collections (the -l 1 arguments plays 1 second of audio from each 30-second clip). You can ctrl-c anytime to exit sfplay.

```
sfplay -l 1 music.mf
sfplay -l 1 speech.mf
```

Now we are ready to train a classifier that can be used for real-time music/speech discrimination. The following command extracts audio features, train a classifier and writes a text file ‘`ms.mpl`’ describing the entire audio processing network that includes the trained

classifier. The sfplugin executable loads this textual description and then processes any audio file classifying approximately every second of it into either music or speech.

```
bextract music.mf speech.mf -cl GS -p ms.mpl
sfplugin -p ms.mpl ../../audio/music_speech/music_wav/winds.wav
sfplugin -p ms.mpl ../../audio/music_speech/speech_wav/allison.wav
sfplugin -p ms.mpl ../../audio/music_speech/music_wav/gravity.wav
```

The next example shows how automatic genre classification with one feature-vector per file can be performed using Marsyas. Similarly we can create a labeled collection for the genres dataset.

```
mkcollection -c cl.mf -l cl ../../audio/genres/classical
mkcollection -c co.mf -l co ../../audio/genres/country
mkcollection -c di.mf -l di ../../audio/genres/disco
mkcollection -c hi.mf -l hi ../../audio/genres/hiphop
mkcollection -c ja.mf -l ja ../../audio/genres/jazz
mkcollection -c ro.mf -l ro ../../audio/genres/rock
mkcollection -c bl.mf -l bl ../../audio/genres/blues
mkcollection -c re.mf -l re ../../audio/genres/reggae
mkcollection -c po.mf -l po ../../audio/genres/pop
mkcollection -c me.mf -l me ../../audio/genres/metal
cat cl.mf co.mf di.mf hi.mf ja.mf ro.mf bl.mf re.mf po.mf me.mf > genres10.mf
```

Extracting the features and getting statistics about the classification performance (accuracy, confusion matrix etc) can be done as follows (make sure the terminal size is wide enough to show the confusion matrix correctly):

```
bextract -sv genres10.mf -w genres10.arff
kea -w genres10.arff
```

Alternatively the generated .ARFF file can also be opened by the well-known Weka machine learning tool.

In addition to classic audio feature extraction and classification Marsyas can be used for a variety of other audio tasks.

```
sfplay ../../audio/music_speech/music_wav/deedee.wav
phasevocoder -q -ob -p 0.8 ../../audio/music_speech/music_wav/deedee.wav
```

The first command simply plays the file. The second one pitch shifts the audio by a factor of 0.8 without changing the duration using a phasevocoder. A more interactive exploration of phasevocoding is described in section Section 3.2 [User interfaces], page 28.

Finally efficient dominant melodic sound source extraction based on spectral clustering of sinusoidal components can be demonstrated as follows:

```
sfplay ../../audio/music_speech/music_wav/nearhou.wav
peakClustering ../../audio/music_speech/music_wav/nearhou.wav
sfplay nearhouSep.wav
```

3.2 User interfaces

A variety of graphical user interfaces are provided with the Marsyas source distribution. Although it is possible to write a user interface that communicates with Marsyas in any language there is specific support for interfacing with the Qt toolkit by Trolltech

http://www.qtsoftware.com/products/. In order to compile the graphical user interfaces you will need to have Qt4 installed and enable the WITH_QT using CMake. More information can be found at the chapter Chapter 2 [Source installation], page 6.

MarPlayer is a simple audio player that provides a seekable playback progress indicator while playing audio and showcases multi-threading using Qt4 and Marsyas.

```
cd MY_MARSYAS_DIR/build/bin
MarPlayer
```

This will launch the MarPlayer GUI. Click on File and open one of the audio files in the collections (or any file in a Marsyas supported format). Clicking on the playback slider will seek to the corresponding location in the audio file.

```
cd MY_MARSYAS_DIR/build/bin
MarPhasevocoder
```

Open a file and experiment with the sliders. The Frequency and Time sliders can be used to pitch shift the recording without changing the duration or speed up or slow down the recording without changing the pitch respectively. The Sinusoids slider can be used to control the number of sinusoids (sorted by amplitude) that are used to approximate the audio signal at each frame. This example showcases user interaction with a relatively performance intensive audio synthesis technique like the phasevocoder which frequently does not have real-time implementations.

The last example of a user interface is a content-based music browsing interface for large audio collections based on self-organizing maps. First you will need to create the genres10.mf collection file as described in Section Section 3.1 [Command-line tools], page 27.

```
cd MY_MARSYAS_DIR
MarGrid2
```

Click on File-Open-Open Collection File and select the genres10.mf collection. Then click on the E button (Extract) which performs feature extraction for all the 1000 files in the collection. This will take a few minutes and you can view the progress in the terminal output. When the feature extraction is complete click on the T button (Train) which trains a self-organizing map that maps the high-dimenstional continuous audio features representing each song to 2D coordinates on a grid. This takes a few seconds. Now click on the P (Predict) button to place each song on the grid. Feature extraction is performed again therefore this takes about the same time as the Train stage. Click on View-Colour-Mapping mode to see a visualization of the genre distributions over the self-organizing map. Note that the genre information is only used for display purposes but not during the calculation of the mapping. If either the audio features or the self-organizing map did not work the colors would essentially appear randomly distributed. Each square contains one more more tracks that are similar to each other based on the audio feautures. Clicking on a squares allows the user to cycle through the songs. Another interesting feature can be activated by selecting View-Continuous which switches songs continuously as the user hovers over the space without requiring explicit clicking. This mode is particularly effective when using touch surface interaction. Once a mapping is calculated it is possible to save the grid and load it without requiring the time consuming stages of feature extraction and training.

3.3 Web information

Marsyas has been used for a variety of projects in both academia and industry. In addition there are several web-interfaces that use Marsyas as a backend for audio analysis and processing. The Marsyas website contains information about projects, publications, screenshots and web-demos based on Marsyas.

http://marsyas.info/about/projects http://marsyas.info/about/videos
http://marsyas.info/about/publications http://marsyas.info/about/webdemos

4 Available 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 the '`bin/`' subdirectory of the build directory, while the source code for those files is in '`src/apps/{DIR}/`'.

Help wanted: missing info!

descriptions of all these programs

If you can fill in any details, please see Section "Contributing documentation" in Marsyas Developer's Manual.

4.1 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.

4.1.1 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.

4.1.2 Labels

Labels may be added to collections by appending tab-seperated labels after each sound file:

```
/home/gtzan/data/sound/reggae/foo.wav \t music
/home/gtzan/data/sound/reggae/bar.wav \t speech
```

The \t represents an actual tab character. This allows you to create a "master" collection which includes different kinds of labelled sound files:

```
cat music.mf speech.mf > all.mf
```

4.1.3 MARSYAS_DATADIR

Collections support the environment variable `MARSYAS_DATADIR`. This allows the use of `.mf` files shared between users (i.e. for a large dataset of audio). For example, the above collection could be rewritten as:

```
MARSYAS_DATADIR/reggae/foo.wav \t music
MARSYAS_DATADIR/reggae/bar.wav \t speech
```

provided that the user configures the environment variable appropriately. For example, using bash on Linux or MacOS X, users on three different machines may set up the variable as:

```
export MARSYAS_DATADIR=/home/gtzan/data/sound/
export MARSYAS_DATADIR=/Users/gtzan/data/sound/
export MARSYAS_DATADIR=/home/gperciva/media/marsyas-data/
```

4.1.4 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 -c reggae.mf -l music /home/gtzan/data/sound/
```

This also labels the data as '`music`'.

If the `-md` flag is added, then the filenames written to the `.mf` file will contain `MARSYAS_DATADIR` when appropriate.

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 on collections.

> **Warning:** `mkcollection` will add a '`.mf`' to the collection filename if it does not contain any extention; otherwise it will leave the collection filename alone.

4.1.5 Plain filenames in collections

Many datasets contain filenames with spaces or awkward characters (extended characters, quote signs, etc). Ideally, all tools (both within marsyas and outside of marsyas) should handle all valid filenames without any problems, but this does not always occur.

To simplify handling of such datasets, we added a `translate-filenames.py` which replaces all filenames (and directories) with plain numbered filenames in a `numbers/` directory.

To convert to plain filenames:

```
marsyas/scripts/translate-filenames.py -n -d ~/mydata/ismir2004
```

To restore original filenames:

```
marsyas/scripts/translate-filenames.py -r -d ~/mydata/ismir2004
```

Options for more advanced handling can be seen by calling the script with `--help`.

4.2 Soundfile Interaction

4.2.1 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. Using the (–ws) option, you may manually set the window size in samples. 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
sfplugin -p playback.mpl bar.wav
```

4.2.2 sfinfo

sfinfo displays information (number of channels, sampling rate, durations) about a support soundfile (.wav and .au by default .mp3 .ogg if the appropriate extensions are installed)

```
sfinfo foo.wav
```

4.2.3 audioCompare

Compares two audio files for similarity with some small tolerance for sample differences. It returns 0 if the files are the same and 1 if they are they are not. It is used for internal testing.

4.2.4 record

Records to a sound-file using the default audio input device. For example the second command records 5 seconds of stereo (-c 2) audio at a sampling rate of 22050 at 80 percent volume (-g 0.8). The results are written to foo.wav.

```
record -l 3 foo.wav
record -l 5 -c 2 -s 22050 -g 0.8 bar.wav
```

4.2.5 orcarecord

Similar to record but with specic change to record 4 stereo channels in parallel using a TASCAM audio interface. Created for the digitization of tapes for the Orchive project. As in the previous example, this following commmand records 8 seconds (-l 8) of audio input at 30 percent volume (-g 0.3).

```
orcarecord -l 8 -g 0.3 foo.wav
```

4.2.6 sound2png

A program that uses Marsyas to generate a PNG of an input audio file. The PNG can be either the waveform or the spectrogram of the audio file.

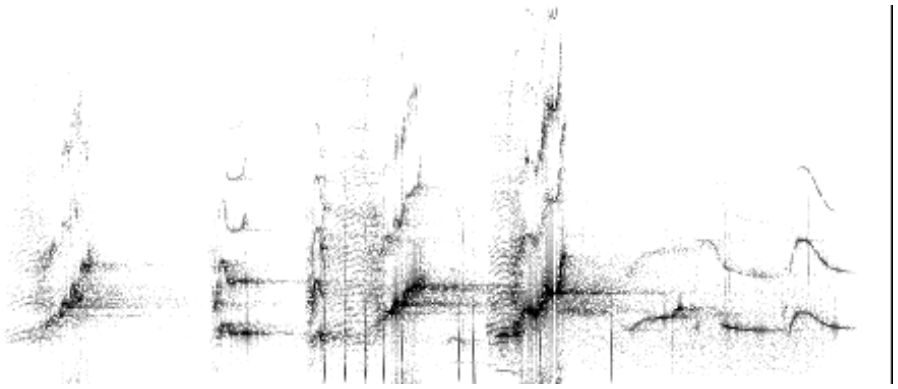
When generating a spectrogram, you can set both the window size and hop size that are used in calculating the FFT. The window size that you give is then used as the amount of

data that the FFT is given, which means that the number of bins for the FFT will be half of the window size. Each bin of the FFT will be drawn in one pixel vertically, so if you use a window size of 512, the resulting PNG will be 256 pixels high.

The hop size for the spectrogram tells the program how much to overlap each FFT by. The width of the output PNG will thus depend on the length of the audio file and the hop size, with smaller hop sizes giving longer PNG images.

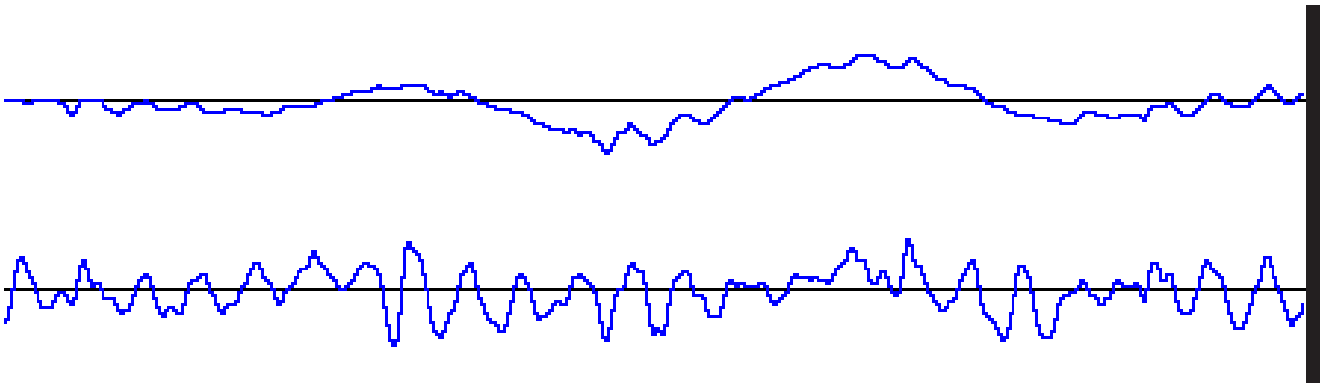
Below is shown an example of using sound2png to generate a spectrogram of an orca call. We use a window size of 1024 and a hop size of 1024. The maximum frequency is set to 8000Hz. A gain of 1.5 is used to make the spectrogram darker:

```
sound2png -m spectrogram A30.wav -ws 1024 -hs 1024 -mf 8000 -g 1.5 out.png
```



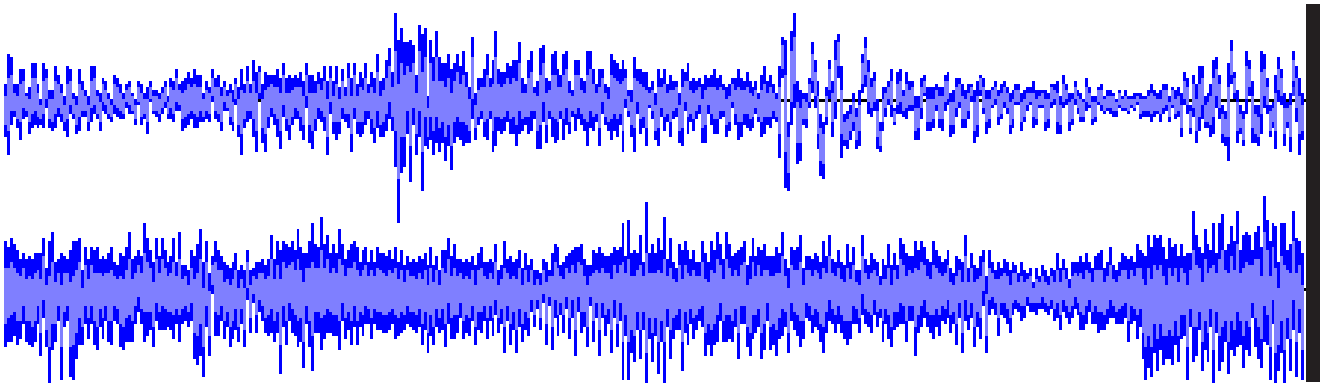
You can also you sound2png to generate pictures of the waveform of an audio file. For this, you use the -w option. An example of this is shown below:

```
sound2png -m waveform tiny.wav -ws 1 out.png
```



When generating pictures of waveforms, you can specify a window size. sound2png takes a chunk of data that is window size samples in length and calculates the maximum and minimum of this window. It then draws a bar from the minimum to the maximum value for each window. An example of this is shown below:

```
sound2png -m waveform small.wav -ws 100 out.png
```



4.2.7 sound2sound

A program that uses Marsyas to do various types of audio processing/digital audio effects that takes as input a single audio file and generate a single audio file that is the result of the processing.

Help info (this should give you all the parameters that you can adjust from the interface):

```
sound2sound -h
```

Bandpass filter with center frequency 500 Hz and a q-factor of 10 the file input.wav processing 30 seconds starting 15 seconds into the file with a gain of 0.8 and write the result to output.wav

```
sound2sound input.wav -m bandpass -f 500 -q 10 -s 15 -l 30 -g 0.8 output.wav
```

Equivalently:

```
sound2sound input.wav --method bandpass --frequency 500 --qfactor 10 --start 15 --leng
```

4.3 Feature Extraction

4.3.1 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. In the following example, the pich is extracted from soundfile, using a windowsize of 1024 (-w 1024), a lower pitch of 36 (-l 36) and an upper pitch of 128 (-u 128).

```
pitchextract -w 1024 -l 36 -u 128 soundfile
```

4.3.2 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. This documentation refers to the latest refactored version of bextract. The old-style `bextract` using the -e command-line option to specify the feature extractor is still supported but use of it discouraged.

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 timbral features (time-domain Zero-Crossings, Spectral Centroid, Rolloff, Flux and Mel-Frequency Cepstral Coefficients (MFCC) over a texture window of 1 sec.

```
bextract music.mf speech.mf -w ms.arff -p ms.mpl -cl GS
bextract ms.mf -w ms.arff -p ms.mpl
bextract -mfcc classical.mf jazz.mf rock.mf -w genre.arff
```

The first two commands are equivalent assuming that ms.mf is a labeled collection with the same files as music.mf and speech.mf. The third-command specifies that only the MFCC features should be extracted and is an example of classifying three classes.

The results are stored in a ms.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. After a header describing the features (attribute in Weka terminology) it consists of lines of comma separated feature values. Each line corresponds to a feature vector. The attributes in the generated .arff file have long descriptive names that show the process used to calculate the attribute. In order to associate filenames and the subsequences of feature vectors corresponding to them each subsequence corresponding to a file is prefixed by the filename as a comment in the .arff file. It is a text file that is straighforward to parse. Viewing it in a text editor will make this clearer.

In addition to Weka, the native Marsyas kea tool Section 4.5.1 [kea], page 44 can be used to perform evaluations (cross-validation, accuracies, confusion matrices) similar to Weka although with more limited functionality.

At the same time that the features are extracted, a classifier (in the example above a simple Naive Bayes classifier (or Gaussian)) 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 stored in ms.mpl.

The resulting plugin makes a classification decision every 20ms but aggregates the results by majority voting (using the Confidence MarSystem) to display time-stamped output approximately every 1 second. The whole network is stored in ms.mpl which is loaded into sfplugin and file_to_be_classified is played and classified at the same time. The screen output shows the classification results and confidence. The second command shows that the live run-time classification can be integrated with `bextract`. In both cases collections can be used instead of single files.

```
sfplugin -p ms.mpl music_file_to_be_classifed.wav
sfplugin -p ms.mpl speech_file_to_be_classifed.wav
bextract -e ms.mf -tc file_to_classified.wav
bextract -e ms.mf -tc collection_to_classified.wav
```

Using the command-line option -sv turns on single vector feature extraction where one feature vector is extracted per file. The single-vector feature representation is useful for many Music Information Retrieval tasks (MIR) such as genre classification, similarity retrieval, and visualization of music collections. The following command can be used to generate a weka file for genre classification with one vector per file.

```
./bextract -sv cl.mf ja.mf ro.mf -w genres.arff -p genres.mpl
```

The resulting genres.arff file has only one feature vector line for each soundfile in the collections. In this case where no -cl command-line argument is specified a linear Support Vector Machine (SVM) classifier is used instead.

Feature sets refer to collections of features that can be included in the feature extraction. It includes several individual feature sets proposed in the MIR and audio analysis literature as well as some common combinations of them. (for details and the most updated list of supported sets experienced users can consult the selectFeatureSet() function in bextract.cpp). The feature sets can be separated into three lagre groups depending what front-end is used: time-domain, spectral-domain, lpc-based.

The following feature sets are supported (for definitions consult the MIR literature, check the corresponding code implementations and send us email with question for details you don't understand) :

'`-timbral --TimbralFeatures`'
: Time ZeroCrossings, Spectral Centroid, Flux and Rolloff, and Mel-Frequency Cepstral Coefficients (MFCC). Equivalent to -mfcc -zcrs -ctd -rlf -flx. This also the default extracted feature set.

'`-spfe --SpectralFeatures`'
: Spectral Centroid, Flux and Rolloff. Equivalent to -zcrs -ctd -rlf -flx.

'`-mfcc --MelFrequencyCepstralCoefficients`'
: Mel-Frequency Cepstral Coefficients.

'`-chroma --Chroma`'
'`-ctd --SpectralCentroid`'
'`-rlf -- SpectralCentroid`'
'`-flx --SpectralFlux`'
'`-zcrs --ZeroCrossings`'
'`-sfm --SpectralFlatnessMeasure`'
'`-scf --SpectralCrestFactor`'
'`-lsp --LineSpectralPair`'
'`-lpcc --LinearPredictionCepstralCoefficients`'

By default stereo files are donwmixed to mono by summing the two channels before extracting features. However, bextract also supports the extraction of feature based on stereo information. There are feature sets that can only be extracted from stereo files. In addition it is possible to use any of the feature sets described above and extract features for both left and right channels that are concatenated to form a feature vector.

'`-spsf --StereoPanningSpectrumFeatures`'
'`-st --stereo`'
: Calculate whatever feature sets are activated for both left and right channels.

For example the first command below calculates MFCC for both left and right channels. The second command calculates the Stereo Panning Spectrum Features which require both channels and also the Spectral Centroid for both left and right.

```
bextract -st -mfcc mymusic.mf -w mymusic.arff
bextract -spsf -st --SpectralCentroid -w mymusic.arff
```

The feature extraction can be configured in many ways (only some of which are possible through command-line options). The following options can be used to control various

aspects of the feature extraction process (most of the default values assume 22050 Hz sampling rate):

‘`-c --collection`’
: the collection of files to be used

‘`-s --start`’
: starting offset (in seconds) into each soundifle from which features will be extracted

‘`-l --length`’
: length (in seconds) of each soundfile from which features will be extracted. A length of -1.0 indicates that the entire duration of the file should be used (the default behavior)

‘`-n --normalization`’
: apply normalization to audio signal

‘`-fe --featExtract`’
: only extract features without training the classifier

‘`-st --stereo`’
: use stereo feature extraction

‘`-ds --downsample`’
: downsample factor (default 1)

‘`-ws --winsamples`’
: size in samples of the analysis window (default 512)

‘`-hp --hopsamples`’
: size in samples of the hop analysis size (default 512 - no overlap)

‘`-as --accSize`’
: size in analysis frames of how many feature vectors are summarized when single vectors per file are calculated (default 1298 - approximately 30 seconds)

‘`-m --memory`’
: size in analysis frames of how many features vectors are summarized for each texture window (default 40 - approximately 1 second)

‘`-cl --classifier`’
: classifier used for training and prediction (default GS - a simple Naive Bayes Classifier)

‘`-e --extractor`’
: old-style specification of feature extraction maintained for backward compatibility (usage discouraged)

‘`-p --plugin`’
: filename of generated Marsyas plugin (.mpl file)

‘`-w --wekafile`’
: filename of generated .arff file (for Weka or kea)

‘`-tc --test`’
: filename of collection or soundfile used for prediction after a model is trained (can be used to conduct MIREX style experimetns)

‘`-pr --predict`’
: filename of a collection or soundfile used for prediction after a model is trained

‘`-wd --workdir`’
: Directory where all generated files will be written by defautl the current directory is used

TimeLines

bextract also supports a mode, called the Timeline mode that allows labeling of different sections of an audio recording with different labels. For example, you might have a number of audio files of Orca recordings with sections of voiceover, background noise, and orca calls. You could train a classifier to recognize each of these types of signal. Instead of a label associated with each file in the collection there is an associate Marsyas timeline file (the format is described below). To run bextract in Timeline mode, there are two steps: training and classifier:

```
bextract -t songs.mf -p out.mpl -pm

Where:

-t songs.mf - A collection file with a song name and its
corresponding .mtl (Marsyas Timeline) file on each line

-p out.mpl  - The Marsyas Plugin to be generated

-pm         - Mute the output plugin
```

and predicting labels for a new audio recording

```
   sfplugin -p out.mpl songmono.wav

Where:

-p out.mpl   - The plugin output by bextract in step #1
```

The songs.mf file is Marsyas collection file with the path to song (usually .wav) files and their corresponding Marsyas Timeline (.mtl) files on each lines. Here is an example song.mf file:

```
/path/to/song1.wav \t /path/to/song1.mtl
/path/to/song2.wav \t /path/to/song2.mtl
/path/to/song3.wav \t /path/to/song3.mtl
```

Please note that the separator character \t must be an actual tab, it cannot be any other kind of whitespace.

The .mtl format has three header lines, followed by blocks of 4 lines for each annotated section. The format is:

```
HEADER:
```

```
-------

number of regions
line size (=1)
total size (samples)

FOR EACH SAMPLE:
----------------
start (samples)
classId (mrs_natural)
end (samples)
name (mrs_string)
```

For example:

```
3
1
2758127
0
0
800000
voiceover
800001
1
1277761
orca
1277762
2
2758127
background
```

Because the .mtl file is kind of obtuse, we have written a small Ruby program to convert Audacity label files to .mtl format. This script can be found at marsyas/scripts/generate-mtl.rb. The script is currently hardcoded to recognize the chord changes from songs from the annotated Beatles archive, but you can easily change this by modifying the "chords_array" variable.

4.3.3 ibt

ibt – standing for INESC-Porto Beat Tracker – is a tempo induction and beat tracking system based on a competing multi-agent system that considers parallel hypotheses regarding tempo and beats. The algorithm works either in real-time, for on-line systems, and off-line, for MIR-based applications.
Benchmarks on causal and noncausal versions reveal competitive results, under alternative conditions.
More technical information can be found in the paper "IBT: A Real-Time Tempo and Beat Tracking System" published in the International Conference on Music Information Retrieval (ISMIR) 2010. Online at: http://www.inescporto.pt/~fgouyon/docs/OliveiraGouyonMartinsReis_ISMIR2010.pdf.

Examples of usage:

```
1. ibt foo.wav
2. ibt -a foo.wav
3. ibt -f foo.wav
4. ibt -mic
5. ibt -a -mic
6. ibt -f -mic
7. ibt -nc foo.wav
```

1. This causally processes foo.wav retrieving the processed beat times, in foo.txt, and the respective median tempo, in foo_medianTempo.txt.
NOTE: By default an initial 5secs is used for the phase+tempo induction stage, where the system is setup.
2. Identical to **1.** but playing audio with "clicks" on beats.
3. Identical to **1.** but saving a file with the audio + "clicks" on processed beats.
4. This captures audio from microphone inpute and processes beats in real-time.
The processed beat times and tempo are saved, respectively, in mic.txt and mic_medianTempo.txt.
NOTE: Although IBT makes an effort to be noise robust it is still quite prone to it, so a noise-clean environment is adviced for using the mode of operation.
5. Identical to **4**. but playing "clicks" on processed beats.
6. Identical to **4**. but saving a file with the captured audio + "clicks" on processed beats.
7. This processes foo.wav non-causally (off-line mode), retrieving the the processed beat times, in foo.txt, and the respective median tempo, in foo_medianTempo.txt.
NOTE: this mode of operation working off-line provides better performance than 1., ideal to MIR-based applications.

‘`-mic --microphone_input`’
: input sound via microphone interface.

‘`-nc --non-causal`’
: for running in non-causal mode.

‘`-t --induction_time`’
: time (in secs) dispended in the initial induction stage.

‘`-o --output`’
: output files (predicted beat times, mean/median tempo):
"beats", "medianTempo", "meanTempo", "beats+medianTempo", "beats+meanTempo", "beats+meanTempo+medianTempo", "none".

‘`-b --backtrace`’
: after induction backtrace the analysis to the beginning (for causal [default] mode).

‘`-di --dumb_induction`’
: for ignoring period induction substituting it by manual defined values.

‘`-a --audio`’
: play the original audio mixed with the synthesized beats.

‘`-f --audiofile`’
output the original audio mixed with the synthesized beats (as fileName_beats.*).

‘`-s --score_function`’
heuristics which conducts the beat tracking: `"regular"` [default], `"correlation"`, `"squareCorr"`.

‘`-m --metrical_time`’
initial time (in secs) which allows tracking metrical changes (0 not allowing at all; -1 for the whole music [default]).

‘`-l --log_file`’
generate log file of the analysis.

‘`-2b --givefirst2beats`’
replace induction stage with ground-truth (two first beats from beatTimes file - .txt or .beats - from the directory or file given as argument).

‘`-2bs --givefirst2beats_startpoint`’
equal to givefirst2beats mode but starting tracking at the first given beat time.

‘`-1b --givefirst1beat`’
replace initial phase by the given ground-truth first beat (from beatTimes file - .txt or .beats - from the directory or file given as argument).

‘`-1bs --givefirst1beat_startpoint`’
equal to givefirst1beat mode but start tracking at the given phase.

‘`-pgt --giveinitperiod`’
replace initial period given by the ibi of the ground-truth two first beats (from beatTimes file - .txt or .beats - from the directory or file given as argument).

‘`-pgt_mr --giveinitperiod+metricalrel`’
equal to giveinitperiod but complementing it with metrically related tempi (2x, 1/2x, 3x, 1/3x).

‘`-pgt_nr --giveinitperiod+nonrel`’
equal to giveinitperiod but complementing it with non-related tempi.

4.4 Synthesis

4.4.1 `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 sound-file and real-time input pitch-shifting and/or time-scaling. Several variations of the algorithm proposed in the literature have been implemented and can be configured through several command-line options. Familiarity with phasevocoder terminology will help understanding their effect on the transformed sound file. Some representative examples are:

```
phasevocoder foo.wav -f foo_identity.wav
phasevocoder foo.wav -f foo_stretched.wav -n 2048 -w 2048 -d 256 -i 512
phasevocoder foo.wav -ob -cm sorted -s 10 -p 1.5 -f foo_pitch_shifted.wav
phasevocoder foo.wav -f foo_stretched.wav -n 4096 -w 4096 -d 768 -i 1024
-cm full -ucm identity_phaselock
phasevocoder foo.wav -f foo_stretched.wav -n 4096 -w 4096 -d 768 -i 1024
-cm analysis_scaled_phaselock -ucm scaled_phaselock
```

In the first example the input file foo.wav is passed through the classic phasevocoder (overlap-add, FFT-frontend and FFT-backend) without any time or pitch modifications. The second example show how time stretching can be achieved by making the analysis hop size (-d) and the synthesis hop size (-i) different. The -n option specified the FFT size and the -w option specifies the window size. In the third example a bank of sinusoidal oscillators (-ob) is used instead of the FFT-backend and the input is pitch shifted by 1.5. The fourth example uses identity phaselocking (-ucm) and the fifth example uses scaled phaselocking (-cm and -ucm) as described by Laroche and Dobson.

‘`-n --fftsize`’
: size of the fft

‘`-w --winsize`’
: size of the window

‘`-v --voices`’
: number of voices

‘`-g --gain`’
: linear volume gain

‘`-b --bufferSize`’
: audio buffer size

‘`-m --midi`’
: midi input port number

‘`-e --epochHeterophonics`’
: heterophonics epoch

‘`-d --decimation`’
: analysis hop size (decimation)

‘`-i --interpolation`’
: synthesis hop size (interpolation)

‘`-p --pitchshift`’
: pitch shift factor (for example 2.0 is an octave)

‘`-ob --oscbank`’
: use bank of oscillators back-end

‘`-s --sinusoids`’
: number of sinusoids to use if convert mode is sorted

‘`-cm --convertmode`’
: analysis front-end mode: full: use all FFT bins, sorted: sort FFT bins by magnitude and only use s sinusoids, analysis_scaled_phaselock: compute extra analysis info for scaled phaselocking

‘`-ucm --unconvertmode`’
: synthesis back-end mode: classic: propagate phases for all bins, loose_phaselock: described by Puckette, identity_phaselock: pick peaks, propagate phases for peaks and lock regions of influence around them, scaled_phaselock: refinement that takes into account information from the previous frame

‘`-on --onsets filename_with_onsets`’
: takes as input a simple text file with locations of onsets that are used to re-initialize phases and not time stretch transient frames that contain the onsets.

4.5 Machine Learning

4.5.1 `kea`

The previous version of Marsyas 0.1 contained machine learning functionality but until 2007 the new version 0.2 mostly relied on Weka for machine learning experiments. Although this situation was satisfactory for writing papers it was not possible to create real-time networks integrating machine learning. Therefore an effort was made to establish programming conventions for how machine learning MarSystems should be implemented. Last but not least we have always wanted to have as much functionality related to audio processing systems implemented natively in Marsyas.

`kea` is one of the outcomes of this effort. Kea (a rare bird from New Zealand) is the Marsyas counterpart of Weka and provides similar capabilities with the command-line interface to Weka although much more limited (at least for now).

Any weka .arff file can be used as input to kea although ususally the input is the extracted .arff files from `bextract`. The following command-line options are supported.

‘`-m --mode`’
: specifies the mode of operation (train, distance_matrix, pca). The default mode is train.

‘`-cl --classifier`’
: the type of classifier to use if mode is train Available classifiers are GS, ZEROR, SVM

‘`-w --wekefile`’
: the name of the weka file

‘`-id --inputdir`’
: input directory

‘`-od --outputdir`’
: output directory

‘`-dm --distance_matrix`’
: filename for the distance matrix output if mode is distance_matrix

The main mode (train) basically performs 10-fold non-stratified cross-validation to evaluate the classification performance of the specified classifier on the provided .arff file. In addition to classification accuracy It outputs several other summary measures of the classifier's performance as well as the confusion matrix. The format of the output is similar to Weka.

The mode distance_matrix is used to compute a NxN similarity matrix based on the input .arff file containing N feature vector instances. The output format is the one used for MIREX 2007 music similarity task. This functionality relies on specific naming conventions related to the Marsyas MIREX2007 submission. By default the output goes to dm.txt but can be specified by the -dm command-line option. The following examples show different ways `kea` can be used.

The pca mode reduces the input feature vectors by projecting them to the first 3 principal components using Principal Component Analysis (PCA). Each component is normalized to lie in the range [0-512]. The resulting transformed features are simply written to stdout.

```
kea -w iris.arff
kea -m train -w iris.arff -cl SVM
kea -m distance_matrix -dm dmatrix.txt -w iris.arff
kea -m pca -w iris.arff
```

4.6 Auditory Scene Analysis

4.6.1 peakSynth

4.6.2 peakClustering

peakClustering performs sinusoidal analysis followed by a spectral clustering for grouping spectral peaks that operates across time and frequency over "texture" windows. It can be used for predominant melody separation on polyphonic audio signals. More technical information can be found in the paper "Normalized Cuts for Predominant Melodic Source Separation" published in the IEEE Transactions on Audio, Speech and Language processing. Examples of usage:

```
peakClustering foo.wav
```

This will result in a file named fooSep.wav that contains the separated predominant melody source such as a singing voice in a rock song or a saxophone line in a jazz tune.

'`-n --fftsize`'
: size of fft

'`-w --winsize`'
: size of window

'`-s --sinusoids`'
: number of sinusoids per frame

'`-b --buffersize`'
: audio buffer size

'`-o --outputdirectoryname`'
: output directory path

‘`-N --noisename`’
name of degrading audio file

‘`-p --panning`’
panning informations <foreground level (0..1)>-<foreground pan (-1..1)>-<background level>-<background pan>

‘`-t --typeSimilarity`’
similarity information a (amplitude) f (frequency) h (harmonicity)

‘`-q -quitAnalyse`’
quit processing after specified number f seconds

‘`-T --textureSize`’
number of frames in a texture window

‘`-c -clustering`’
number of clusters in a texture window

‘`-v --voices`’
number of voices

‘`-F --clusterFiltering`’
cluster filtering

‘`-A --attributes`’
set attributes

‘`-g --ground`’
set ground

‘`-SC --clusterSynthetize`’
cluster synthetize

‘`-P --peakStore`’
set peak store

‘`-k -keep`’ keep the specified number of clusters in the texture window

‘`-S --synthetise`’
synthetize using an oscillator bank (0), an IFFT mono (1), or an IFFT stereo (2)

‘`-r --residual`’
output the residual sound (if the synthesis stage is selected)

‘`-i --intervalFrequency`’
<minFrequency>_<maxFrequency> select peaks in this interval (default 250-2500 Hz)

‘`-f --fileInfo`’
provide clustering parameters in the output name (s20t10i250_2500c2k1uTabfbho means 20 sines per frames in the 250_2500 Hz frequency Interval, 1 cluster selected among 2 in one texture window of 10 frames, no precise parameter estimation and using a combination of similarities abfbho)

‘`-npp --noPeakPicking`’
do not perform peak picking in the spectrum

‘`-u --unprecise`’
do not perform precise estimation of sinusoidal parameters

‘`-if --ignoreFrequency`’
ignore frequency similarity between peaks

‘`-ia --ignoreAmplitude`’
ignore amplitude similarity between peaks

‘`-ih --ignoreHWPS`’
ignore harmonicity (HWPS) similarity between peaks

‘`-ip --ignorePan`’
ignore panning similarity between peaks

‘`-uo --useOnsets`’
use onset detector for dynamically adjusting the length of texture windows

4.7 Marsystem Interaction

4.7.1 `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
```

Writing a basic sfplugin plugin

The sfplugin application expects that certain controls are available in any network it tries to handle. Therefore, one of the simplest demonstration plugins one can write is a plugin containing a SoundFileSource and an AudioSink, demonstrated below. As the sfplugin does not know were the sources and sinks are in the network it is necessary to link the composite's controls with appropriate controls in the network.

```
// create the network that will become the plugin
MarSystem* sys = mng.create( "Series", "head" );

// create the two required MarSystems
sys->addMarSystem( mng.create( "SoundFileSource", "src" ) );
sys->addMarSystem( mng.create( "AudioSink", "dest" ) );

// while we don't actually want to play a file now, supply a valid
// filler name to keep the program happy; sfplugin will update it later
sys->updctrl( "SoundFileSource/src/mrs_string/filename",
      "../../SuperGroovyLateralGeniculateNucleus.au" );

// since we're not playing the song now, set initAudio to false;
```

```
// sfplugin will update this to true when the network is executed there
sys->updctrl( "AudioSink/dest/mrs_bool/initAudio", false );

// set those pesky control links!
sys->linkctrl( "mrs_string/filename", "SoundFileSource/src/mrs_string/filename" );
sys->linkctrl( "mrs_bool/initAudio" , "AudioSink/dest/mrs_bool/initAudio" );
sys->linkctrl( "mrs_natural/pos"    , "SoundFileSource/src/mrs_natural/pos" );
sys->linkctrl( "mrs_bool/hasData"   , "SoundFileSource/src/mrs_bool/hasData" );

// finally, write the network to a file; the plugin can be run as
// follows: sfplugin -p some_plugin.mpl ../../SamsSavorySuperiorColliculus.au
ofstream ofs( "some_plugin.mpl" );
ofs << (*sys) << endl;
```

4.7.2 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. Msl has been inactive for a while as the SWIG bindings to scripting language seem to be the way to go.

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. It also possible to use the GNU readline utility to provide more friendly user editing and command completion in Msl.

4.8 All of the above

4.8.1 mudbox

In computer science the term sandbox is frequently used to refer a technique where a piece of software is isolated from the surrounding operating system environment to reduce security risks. Unlike these clean sandboxes in Marsyas the mudbox is a playground for experimentation and messing around. This is all the more appropriate given that several of the main Marsyas developers have lived in British Columbia where sandbox turn into

mudboxes most of the year. It was motivated by the frequent need to experiment with various MarSystems under construction without having to create a new application for each case and potentially have to modify the build system. Typically code for a MarSystem gets tested in mudbox, then is gradually expanded to a tool and eventually becomes it's own application or gets integrated to one of the existing ones. More information can be found Section "Playing in the mudbox" in *Marsyas Developer's Manual*. The examples in mudbox are short and can provide quick templates for various types of tasks. The specific example to be executed is specified by the –toy-with, -t command-line argument:

```
mudbox -t onsets foo.wav
mudbox --toy-with vibrato foo.wav
mudbox -h
```

The first command will generate a stero file foo.wav_onsets.wav with one channel containing the detected onsets. The second command will apply a vibrato type of effect using a delay line to foo.wav. The third command will display many (but not necessarily) all of the available "toys" in mudbox. In general, mudbox is supposed to be experimental so don't expect careful error checking or proper output messages. In most cases you will need to read the corresponding source code in mudbox.cpp to figure out what's happening. This is a feature of mudbox not a bug :-).

5 Graphical User Interfaces using Qt4

Marsyas provides integration support for the Qt4 user interface toolkit. By using the MarSystemQtWrapper any processing network can be wrapped into a proper QObject with signals and slots enabling direct integration with Qt objects and widgets. The wrapper hides issues of threading, buffering, etc. Another supported functionality that uses the Qt signal/slot system is the ability to send and process Open Sound Control

5.1 MarPlayer

MarPlayer is a simple graphical user interface for playback of soundfiles supported in Marsyas. It has shuttle controls and allows for arbitrary seeking and looping. From a developer perspective it provides an example of how the Qt Designer interface toolkit for desinging and layout and graphical appearance of the user interface.

5.2 MarPhaseVocoder

MarPhaseVocoder is a front-end to the marsyas phasevocoder. It is an example of how all the interface code can be directly written in C++. It uses a common architecture with three layers for separating the handling of graphical user interface action, their mapping, and their connection to the wrapper.

5.3 MarGrid

MarGrid is a music collection visualization interface. It utilizes feature extraction and a self-organized map to map each song in a collection to a square in a grid of cells. Once the self-organized map has been trained it can be used to predict grid positions for new songs.

5.4 MarMonitors

MarMonitors is a proof-of-concept interface that allows the inspection of realvec controls of any MarSystem. It's a step towards generic interface for designing, debugging processing network that are not tied to a specific application.

5.5 MarLPC

MarLPC is a user interface to a system for adding breathiness to the singing voice using Adaptive Pre-Emphsis Linear Prediction (Nordstrom, Tzanetakis, Driessen 2008). It also serves as an example of how Open Sound Control (OSC) can be used to change controls through Qt. The patch scripts/PureData/MarLpc.pd can be usd to demontrate this communication.

6 Architecture concepts

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 9 [Writing applications], page 76, and writing new MarSystems is described in Chapter 10 [Programming MarSystems], page 130.

6.1 Architecture overview

6.1.1 Building MarSystems

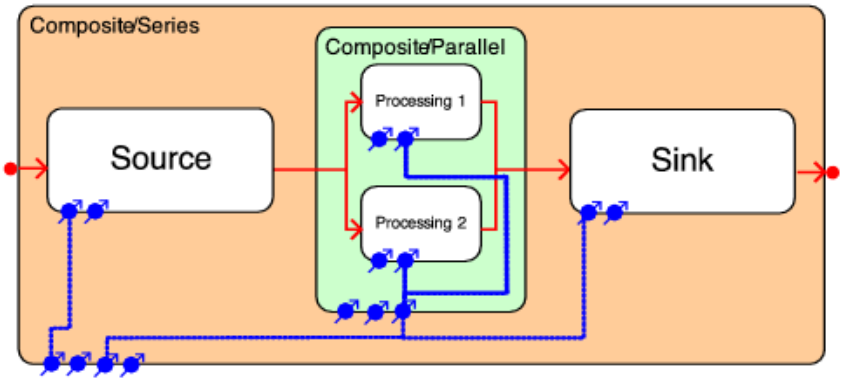
The basic idea behind the design of Marsyas is that any audio analysis/synthesis computation can be expressed as some type of processing object, which we call MarSystem, that reads data from an input slice of floating point numbers, performs some computation/transformation based on data, and writes the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

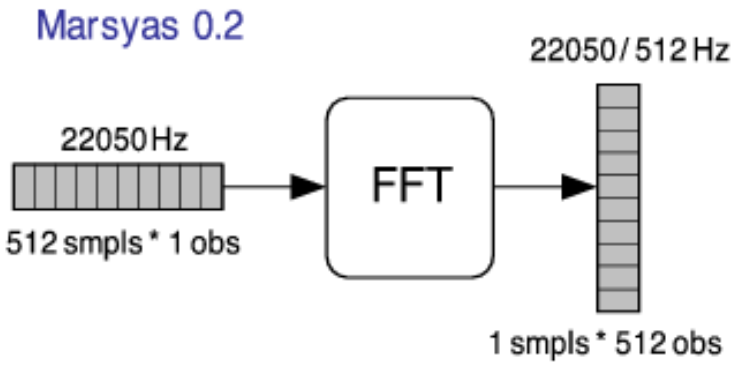
As is very frequently the case with audio processing networks objects the input of each stage is the output of the previous stage. This way of assembling MarSystems is called a Series composite. Once a Series Composite is formed it can basically be used as one MarSystem that does the whole thing. A figure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

6.1.2 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.

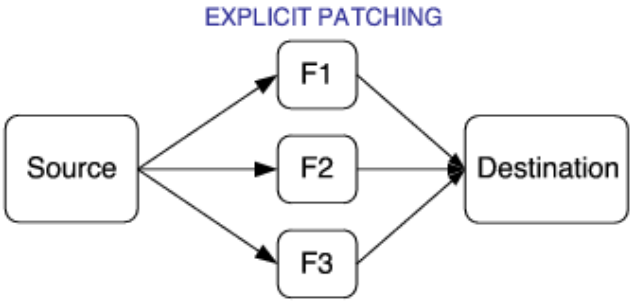


6.2 Implicit patching

6.2.1 Implicit patching vs. explicit patching

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

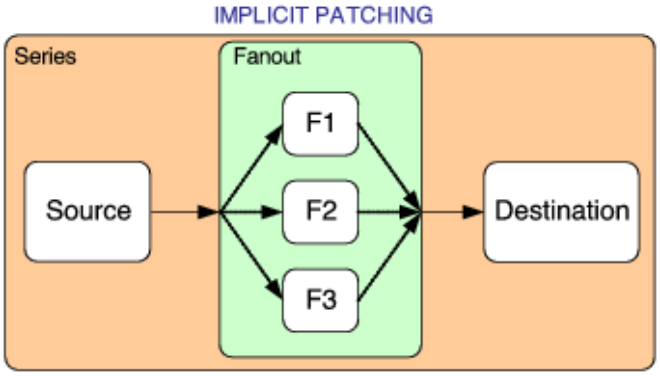
```
# EXPLICIT PATCHING: block definitions
source, F1, F2, F3, destination;
# connect the in/out ports of the blocks
connect(source, F1);
connect(source, F2);
connect(source, F3);
connect(F1, destination);
connect(F2, destination);
connect(F3, destination);
```



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

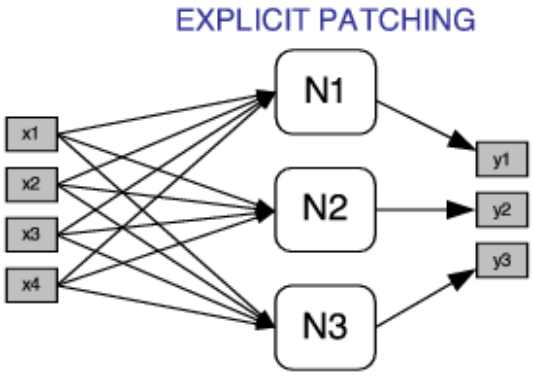
```
# IMPLICIT PATCHING
source, F1, F2, F3, destination;
Fanout(F1, F2, F3);
```

```
Series(source, Fanout, destination);
```



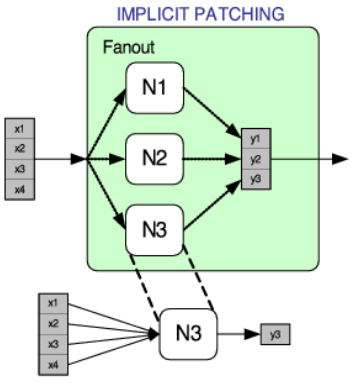
6.2.2 Implicit patching advantages

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

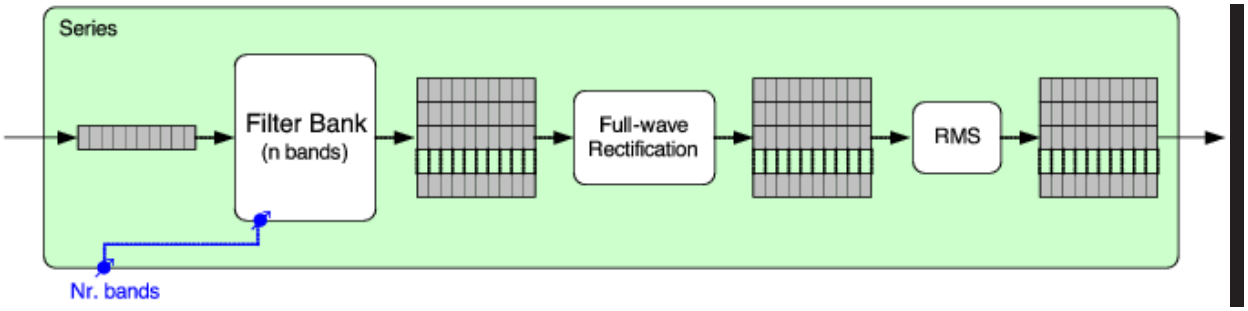


With implicit patching, this is much more manageable.

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

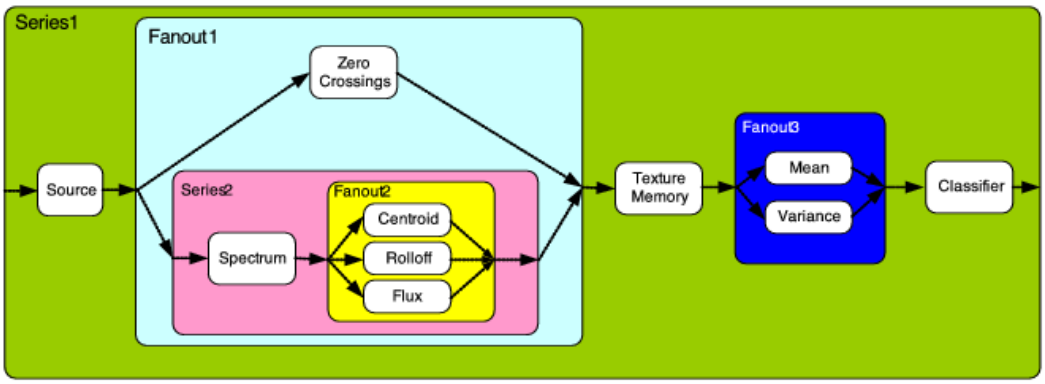


Implicit patching can automagically adjust the connections without requiring any code recompiliation. For example, we can change the number of bands in a filter bank without changing any code.



6.2.3 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"));
```

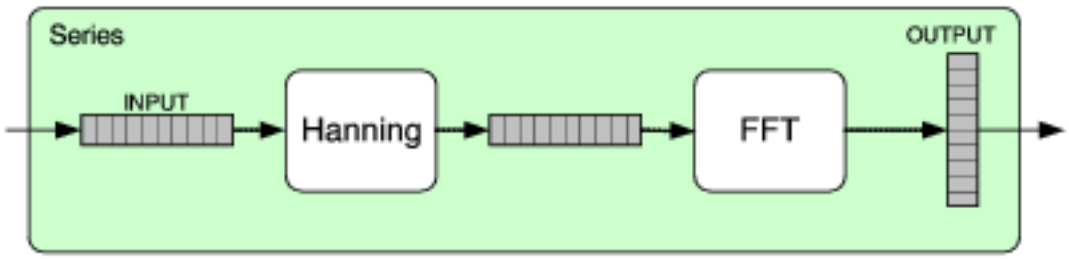
6.3 MarSystem composites

Help wanted: missing info!

descriptions of composites, add the other composites

If you can fill in any details, please see Section "Contributing documentation" in Marsyas Developer's Manual.

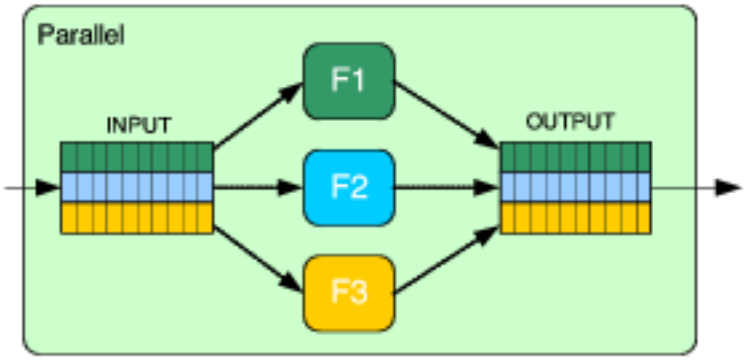
6.3.1 Series



The Series composite is the most basic structure for connecting MarSystems into dataflow networks through implicit patching. It is similar to Unix pipes or the chuck operator in the Chuck programming language. The output of the first object in the Series connection becomes the input to the second object, the output of the second object becomes the input to the third object etc. A MarSystem can potentially have different size and characteristics for the input and output slices. In Series connections these characteristics are constrained to ensure that the output slice of each MarSystem is the same as the input slice of the MarSystem that is downstream from it. The figure shows how a Series composite can be used to express a complex spectrum computation from a windowed segment of audio. The Hanning MarSystem windows the incoming segment and the FFT MarSystem converts the audio to a complex spectrum. Notice how the dimensions of the slices can change at the different stages of processing.

6.3.2 Parallel

A Parallel composite is used when you have an input with multiple observations (i.e. channels) and you want to run calculations in parallel on each of the observations. This is shown in graphical form below:



In the above diagram, the Parallel MarSystem is fed a realvec with 3 observations (and a large number of samples). The MarSystem F1 receives the first observation, coloured in green. The second MarSystem in the Parallel, F2, gets the second observation, coloured in blue, and the third MarSystem, F3, receives the third observation, coloured in yellow.

To build this system you would write code that would look something like:

```
MarSystem* parallel = mng.create("Parallel", "parallel");
parallel->addMarSystem(mng.create("Gain", "F1"));
parallel->addMarSystem(mng.create("Gain", "F2"));
parallel->addMarSystem(mng.create("Gain", "F3"));
```

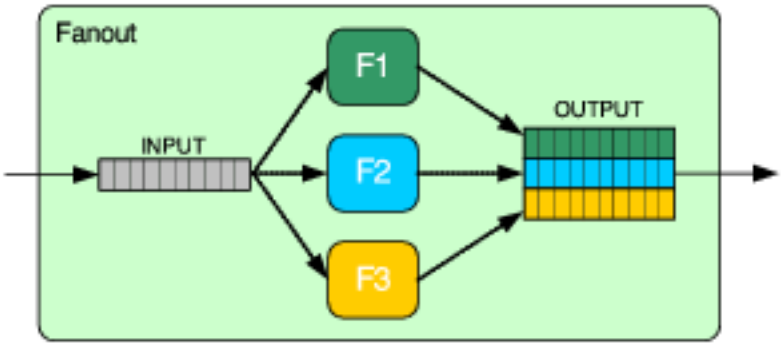
One real-life use of Parallels is if you have a stereo sound source and you want to run an analysis on the left and right channels seperately. An example of generating a spectrum for the right and left channels of a sound source would look something like:

```
MarSystem* net = mng.create("Series", "net");
net->addMarSystem(mng.create("SoundFileSource", "src"));

MarSystem* stereobranches = mng.create("Parallel", "stereobranches");
MarSystem* left = mng.create("Series", "left");
MarSystem* right = mng.create("Series", "right");

left->addMarSystem(mng.create("Windowing", "hamleft"));
left->addMarSystem(mng.create("Spectrum", "spkleft"));
left->addMarSystem(mng.create("PowerSpectrum", "leftpspk"));

right->addMarSystem(mng.create("Windowing", "hamright"));
right->addMarSystem(mng.create("Spectrum", "spkright"));
right->addMarSystem(mng.create("PowerSpectrum", "rightpspk"));

stereobranches->addMarSystem(left);
stereobranches->addMarSystem(right);
```

```
net->addMarSystem(stereobranches);
```

6.3.3 Fanout

A Fanout is another composite MarSytem that, like a Parallel, contains other MarSystem. It differs from a Parallel in that it takes a single observation and sends a copy of this observation to all the MarSystems inside of it. This is shown graphically below:



In the above diagram, we send a realvec with a single observation (and many samples) into the Fanout. The Fanout then makes a copy of this data for each of the MarSystems within in. In the above diagram, it makes a copy of the input observation and sends it to F1 (shown in green), F2 (shown in blue) and F3 (shown in yellow).

Thus, the input of the Fanout shown above is one observation (and many samples) and the output of the Fanout is three observations (and the same number of samples as the input had).

To build this system you would write code that would look something like:

```
MarSystem* fanout = mng.create("Fanout", "fanout");
fanout->addMarSystem(mng.create("Gain", "F1"));
fanout->addMarSystem(mng.create("Gain", "F2"));
fanout->addMarSystem(mng.create("Gain", "F3"));
```

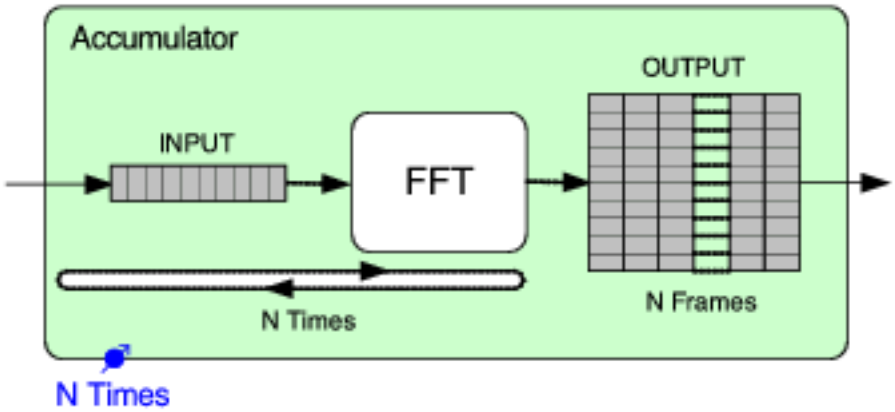
One real-life example of using a Fanout is if you want to run many different kinds of algorithms on your data. For example, you might to run a series of different filterbanks on your audio. To do this you would write code that would look something like:

```
MarSystem* net = mng.create("Series", "net");
net->addMarSystem(mng.create("SoundFileSource", "src"));

MarSystem* filterbank = mng.create("Fanout", "filterbank");
filterbank->addMarSystem(mng.create("Filter", "cf8"));
filterbank->addMarSystem(mng.create("Filter", "cf12"));
filterbank->addMarSystem(mng.create("Filter", "cf18"));
filterbank->addMarSystem(mng.create("Filter", "cf20"));

net->addMarSystem(filterbank);
```

6.3.4 Accumulator

An Accumulator MarSystem accumulates result of multiple tick process calls to internal MarSystem. It then generates output only once when all the results are accumulated. It is used to change the rate of process requests.

For example, if the nTimes control for the Accumulator is set to 6, then each time the Accumulator receives a tick(), it sends 6 tick()s to the MarSystems that are inside it. This is shown graphically below:



One common use case for an Accumlator is if your algorithm needs to operate on a block of data, for example, it might need to analyze a number of frames of an FFT in order to detect a particular audio event.

An example of using an Accumulator could look like be:

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

MarSystem* acc = mng.create("Accumulator", "acc");
MarSystem* accuInternal = mng.create("Series", "accuInternal");

accuInternal->addMarSystem(mng.create("SoundFileSource", "src"));
accuInternal->addMarSystem(mng.create("Windowing", "ham"));
accuInternal->addMarSystem(mng.create("Spectrum", "spk"));
acc->addMarSystem(accuInternal);

//IMPORTANT NOTE:
//
//note that you can only add one Marsystem to an Accumulator
//any additional Systems added are simply ignored outputwise !!
//e.g. if you want to use multiple Marsystems in a row and accumulate
//their combined output, you need to put them in a series which you add
//to the accumulator

net->addMarSystem(acc);
net->updctrl("mrs_natural/inSamples", 512);
```

```
net->updctrl("Accumulator/acc/mrs_natural/nTimes", 10);
```

In the above example, a SoundFileSource is followed by a hamming Window, a Spectrum and a PowerSpectrum. All of these are added to an Accumulator, which is then added to a global Series. The nTimes control of the Accumulator is then set to 10.

The output of this MarSystem would be a realvec with 512 observations, corresponding to each of the bins of the FFT generated by the Spectrum, and with 10 columns (or samples). Your algorithm could then analyze this two-dimensional matrix to look for the audio feature you are investigating.

6.3.5 Shredder

A Shredder composite MarSystem does the inverse to what an Accumulator does. While an Accumulator builds up a realvec containing the results from multiple ticks, a Shredder splits this realvec into multiple chunks, effectively increasing the rate at which data is output.

A toy example of using both a Shredder and an Accumulator together would look something like:

```
MarSystem* net = mng.create("Series", "net");
MarSystem* acc = mng.create("Accumulator", "acc");
MarSystem* shred = mng.create("Shredder", "shred");

acc->addMarSystem(mng.create("SoundFileSource", "src"));
net->addMarSystem(acc);
shred->addMarSystem(mng.create("AudioSink", "dest"));
net->addMarSystem(shred);

net->updctrl("Accumulator/acc/SoundFileSource/src/mrs_string/filename", sfName);
net->updctrl("Shredder/shred/AudioSink/dest/mrs_bool/initAudio", true);

net->updctrl("mrs_natural/inSamples", 256);

net->updctrl("Accumulator/acc/mrs_natural/nTimes", 10);
net->updctrl("Shredder/shred/mrs_natural/nTimes", 10);
```

In the above example, an Accumulator is added first to a global Series network, and then a Shredder is added to this Series. A SoundFileSource is then added to the Accumulator, and an AudioSink is added to the Shredder. The nTimes control of both the Shredder and Accumulator are then set to 10.

This network would read in data from the SoundFileSource, and the Accumulator would build up a buffer of 10 chunks of this audio. The Shredder would then take these 10 slices of audio data and would split them back into 10 chunks, which would then be output to the AudioSource.

This whole procedure would be akin to buffering a section of audio 10 times the size of the number of samples that were input. So, since we set the inSamples of the network to 256, the number of samples output by the Acccumulator would be 2560.

6.4 Linking of controls

THIS SECTION WAS JUST A COPY-PASTE FROM A LONG EMAIL FROM THE DEVELOPERS MAILING LIST - STILL NEEDS A BIT OF REVISION, BUT ALREADY PROVIDES IMPORTANT INFORMATION ABOUT THE LINKING OF CONTROLS IN MARSYAS (lgmartins)

This refactoring is the "part II" of the last refactoring to the linking mechanism, done some weeks ago.

In the first part of the refactoring we changed the way linked controls store their values (i.e their data - a real number, an integer, a string, a realvec, etc). Before this refactoring, each control had it's own data allocation, and so everytime we changed a control's value, such change had to be propagated (i.e. copied) to all the linked controls (if any). Such a copy meant that the same data would be replicated in memory a number of times equal to the number of existing links. Furthermore, keeping all those copies in sync each time we change a control value implied a lot of copying. This was really ineficient, both computationally and memory-wise, and all the code for managing such a synchronization was a mess.

So, the first step was to make all MarControls that are linked to each other share a same MarControlValue (i.e. the marsyas object that in fact stores in memory the data in MarControls). This automatically solves the synchronization problem, although it may create some others if we start thinking about multi-treaded code in Marsyas (but let's forget about multi-threads for now).

When linking a control (let's say ctrl_A) to another control (ctrl_B), it's now just a matter of instructing control_A to start using the MarControlValue used by control_B. The old MarControlValue of ctrl_A can therefore be deleted from memory, in case no other MarControl is still pointing to it (i.e. refcounting).

To do so, in marsyas pseuso-code, we would write:

```
ctrl_A->linkTo(ctrl_B);
```

Or, in case we didn't have the direct pointer to the controls, using their pathnames and the MarSystem API:

```
msys->linkControl("mrs_xxx/A", "mrs_xxx/B");
```

This brings us to the first important detail: during the linking of two controls, it's important the order of the linking operation. Doing ctrl_A->linkTo(ctrl_B) will discard the current value of ctrl_A, which will now use the value currently stored in ctrl_B. If we reverse it (ctrl_B->linkTo(ctrl_A);), ctrl_B value will be discarded (we can think as "overwriten") in favour of the value of ctrl_A. This is only relevant at the time of the linking operation, and users should, if nothing else, be aware of which value the want to keep when linking two controls. Of course, after the linking is done, changing the value of ctrl_B will also change the value of ctrl_A, and vice-versa, making this detail syntatically meaningless.

However, this order of the linking operation is now stored internally by all MarControls in marsyas. And the reason for this has to do with the unlinking operation of controls. This was the work done in "part II" of the linking/unlinking refactoring, just completed today.

In the first stage of the refactoring, we were not storing anywhere the order of the linking of two controls. So we just kept a table with references to all the MarControls linked together (i.e. sharing the same value/data), without any information to who linked to whom originally.

This was quite elegant in fact, allowing to easily unlink any control from a set of linked controls by just cloning in memory its current value (i.e. creating a new but equal valued MarControlValue object), removing it from the reference table of the MarControlValue object holding its value, and re-pointing the control to, from that time on, start using the new cloned data object instead.

The problem is that in Marsyas, the most interesting use of unlinking controls is a bit more demanding. The way just described of completely unlinking a control from all the other linked controls may not be desirable at all.

Suppose the following scenario: we have a composite MarSystem (i.e. a MarSystem with other MarSystems inside, connected in series, for e.g.) where we have a control (say, ctrl_X) that we want to always link to, for example, the output control of the last child MarSystem (let's say ctrl_processedData control of the last child MarSystem). This is exactly the case of the composite FlowThru, so you can refer to its .cpp/.h code for an actual implementation. So we start by linking ctrl_X to the ctrl_processedData of the last child:

```
ctrl_processedData = msys->getctrl("xxx/lastChildMsys1/mrs_realvec/processedData");
ctrl_X->linkTo(ctrl_processedData);
```

Semanticaly, this is the order that makes more sense (but nothing prevents us to do it the other way around - the link will work as well!), since we are overwriting ctrl_X's value with the current value of ctrl_processedData, which is in fact an "output" control (controls in Marsyas do not have an explicit in/out attribute - "output" controls are considered controls to which makes no sense writing to because they will be ignored and overwriten by their owning MarSystem; "input" controls on the other hand will be used internally by their owning MarSystem as parameters, but they may also be read to know the current control value currently being used).

Graphically, this link can be represented as:

```
ctrl_processedData <--- ctrl_X
```

If we now need to link some other controls from other MarSystems to ctrl_X, we just need to explicitly do it:

```
ctrl_Y->linkTo(ctrl_X);
ctrl_W->linkTo(ctrl_X);
ctrl_Z->linkTo(ctrl_W); //this control will be indirectly connected to ctrl_X, altough
```

Graphically:

```
ctrl_processedData <--- ctrl_X <--- ctrl_Y
                            ^
                            |--- ctrl_W <--- ctrl_Z
```

So, after this we have ctrl_X, ctrl_Y, ctrl_W and ctrl_Z effectively all linked to and sharing the same value with ctrl_processedData, with minimal computational burden and a small memory footprint.

Now imagine that we want to add a new child to the FlowThru composite. Since we want ctrl_X (and all the controls linked to it) sharing the value of the last child in the FlowThru composite, we must update the ctrl_X —> ctrl_processedData link! In other words, we need to unlink ctrl_X from ctrl_processedData of the previous last child MarSystem, and re-link it to the ctrl_processedData of the new last child of the composite. For that we could think that the following way would cut it:

```
ctrl_processedData = msys->getctrl("xxx/lastChildMsys2/mrs_realvec/processedData");
ctrl_X->unlink();
ctrl_X->linkTo(ctrl_processedData);
```

However, calling unlink in this way (i.e. the way implemented in "part I" of the refactoring) would unlink ctrl_X from all the other linked controls (i.e. ctrl_W, ctrl_Y, etc) and relinking it alone to the new ctrl_processedData (i.e. represented as (2) below) from the new last child. All the other previously linked controls would still be linked to the same ctrl_processedData of the previous last child in the composite. The only way around this would be to manually keep track of what links would have to be unlinked and relinked to the new control.

Graphically:

```
ctrl_processedData(1) <--- ctrl_Y
                                ^
                                |--- ctrl_W <--- ctrl_Z


ctrl_processedData(2) <--- ctrl_X
```

As it is easy to imagine, any minimally complex network of multiple-nested MarSystems and linked controls would be totally insane to manage! Here's an example of what we would have to explicitly do to achieve what we really wanted:

```
ctrl_processedData = msys->getctrl("xxx/lastChildMsys2/mrs_realvec/processedData");
ctrl_X->unlink();
ctrl_X->linkTo(ctrl_processedData_X);
ctrl_Y->unlink();
ctrl_Y->linkTo(ctrl_X);
ctrl_W->unlink();
ctrl_W->linkTo(ctrl_X);
ctrl_Z->unlink();
ctrl_Z->linkTo(ctrl_W); //we must remind that this one was originally connected to W a
```

(OK, if you managed to follow me till here without getting suicidal tendencies or falling totally asleep, please go to http://marsyas.info/community/getting_involved and welcome aboard ;-)).

The best way to make Marsyas solve this automatically for us was to keep a simple record of the original link orders, as explained above. If we know who originally linked to whom, it's then easy to unlink a control from a set of controls, but keep all the controls that were originally connected (or indirectely connected) still attached to it. So after the "part II" of the refactoring, if we want to reconnect ctrl_X (and all the controls originally linked to it) to the new last child in the composite, we would only need to do:

```
ctrl_X->unlinkFromTarget(); //this is in fact not needed! See below...
ctrl_processedData = msys->getctrl("xxx/lastChildMsys2/mrs_realvec/processedData");
ctrl_X->linkTo(ctrl_processedData);
```

As it's possible to see, the unlink() method is no longer used (it was deprecated) and we should now use unlinkFromTarget() for achieving the desired result. By "target" I mean the following: everytime we link a control *to* another control (and here we start to see the importance of the order of linking, other than the known issue of which control gets its

value overwriten by the other as explained above), we are directly "targeting" or linking to that control. So for the case of ctrl_X, when we linked it to ctrl_processedData, we made the latter the "target" of ctrl_X. As we can see, each control, even if linked to any arbitrary number of controls, only has one "target" control, i.e. the one it was originaly linked to. So, when we call ctrl_X->unlinkFromTarget() above, we are just unlinking ctrl_X from its target (i.e. ctrl_processedData(1) and any other controls that have that control as their target) but keeping the links with all the other controls that have ctrl_X as their direct or indirect target. Graphically we get:

```
ctrl_processedData(1)

ctrl_processedData(2) <--- ctrl_X <--- ctrl_Y
                                ^
                                |--- ctrl_W <--- ctrl_Z
```

In truth, we do not even need to explicitly call unlinkFromTarget() before linking ctrl_X to the new target. Since each linked control can only have one target, re-linking it to other control will automatically unlink it from its previous target.

All these linked controls can be seen as a directed graph problem (but please do not suggest me using Ncuts in it ;-)), where there is no possibility of existing loops (e.g. attemping to link ctrl_X with ctrl_Z, as depicted above, would do nothing, since it's very easy to detect that they are already linked - we just need to compare the pointers to the respective MarControlValue of each MarControl: if they point to the same object it means the controls are already linked, so very easy and efficient to avoid any loops!).

In any case, if we need to completely unlink a control from all its linked controls (regardeless of targets etc - i.e. the way the now deprecated unlink() method worked), we can simply use:

```
ctrl_X->unlinkFromAll();
```

Although not so usefull as the former version of unlinking, there are some situations in Marsyas where we need this (if you dare or if you are really curious about this, you can have a look ate MarSystem copy constructor and at the put() method ;-)).

As a conclusion, if when constructing networks of linked controls in Marsyas we take into mind the importance of the linking order, it's really easy to construct semantically meaninful links that will allow easy reconfiguration of MarSystem networks. Actually, this is a mandatory feature if we want in the future to have MarSystem insertion/deletion at runtime, without messing up all the linked controls in the network.

6.5 Scheduling

6.5.1 Scheduling in Marsyas

Scheduling is the art of delaying actions (events) until later periods in time. In Marsyas time and event are are more general than many other systems. This allows for a wide range of user specified control rates and actions.

6.5.1.1 Time

Time is simply a counter. What it counts is up to the writer of a new timer class. The two supplied timers, TmSampleCount and TmRealTime, count audio samples and system micro-seconds respectively.

There are a number of issues surrounding control rates in Marsyas and most other time-aware processing systems. Marsyas' primary task is to process audio data. This data passes through the network in buffers of size N. Usually the network is prompted to process the buffer of N samples by an outer loop. During processing control of the network is lost as the buffer of data passes through each MarSystem object. The scheduler checks for and dispatches events when the network is ticked and before the buffer of data passes through the network. Events are therefore dispatched at the start of each buffer processed.

Event dispatch is therefore governed by the audio sample rate. Ultimately every custom timer bears some relation to the audio sample rate. Since events are dispatched at the start of each tick, events are actually dispatched at a rate of every N samples. This implies that there is a granularity on event dispatch based on the sample rate and buffer size. If an event is to be dispatched at a point in time that falls inside a buffer, ie at 256 for a buffer size of 512 samples, then that event will not be dispatched until the next buffer boundary at which point its dispatch time will be <= current counter time.

We might wish to have sample accurate event timing. After all, there are other audio processing systems that can claim the accuracy we might desire. The major obstacle to achieving this accuracy is due to Marsyas' dataflow architecture. When a buffer of N samples passes through the network it is processed multiple times (normally once by each object in the network). This means that each sample recurs a number of times equal to the number of Marsystems that process it. More specifically, if the time is T when the network is ticked, then after the first MarSystem processes the buffer of N samples time will be at T+N. The next MarSystem will start processing the buffer at time T. In this way, time bounces between T and T+N as the buffer passes through the network. This situation makes it exceedingly difficult to make the network consistent for events that occur between T and T+N. If a control value C is to be changed at time T+K where 0<K<N then for C to remain consistent for all MarSystem objects that might be interested then it must be changed to the previous state at the start of buffer processing then to the new value at the event time - each time the buffer is processed. This would be somewhat difficult and computationally expensive to accomplish within the dataflow model. For this reason the scheduler only dispatches events on buffer boundaries - when the network is ticked - as this is the only point when all MarSystems are at the same point in time.

In many systems multiple control rates are desireable. Consider two timers based on separate sample rates such as 44.1kHz and 12.34kHz. Since both of these rates are regular, that is they repeat at a constant rate, then a simple conversion function can be used to convert between the two rates. By converting one of the rates to the other a single timer can be used in the scheduler. However for other timers an errorless conversion function may not be possible. Consider a control rate based on detecting someone tapping a drum stick. The rate of tapping could be described as irregular because it is not known exactly when the next tap will happen. Combining this timer with the audio sample rate creates the problem that no perfect conversion function is possible that can compute what the sample time will be for N taps in the future. In Marsyas this problem is bypassed through the

support of multiple timers. Events are therefore posted on the timers they are specified on and no timer conversion takes place.

6.5.1.2 Event

An event is simply something that happens at some point in time. In Marsyas an event is a class that contains a user-defined function. What this function contains is up to the writer and can act on the Marsyas network or outside of it. Marsyas is not threaded and neither is the scheduler so the scheduler will wait for the event to complete its action before resuming. Be careful not to do excessive processing during the dispatch of an event.

6.5.2 Components of the Marsyas Scheduler

The scheduler is made up of a collection of files from those used by the scheduler to those that support it. The classes directly related to the scheduler along with their relationships is shown in See Figure 6.1.

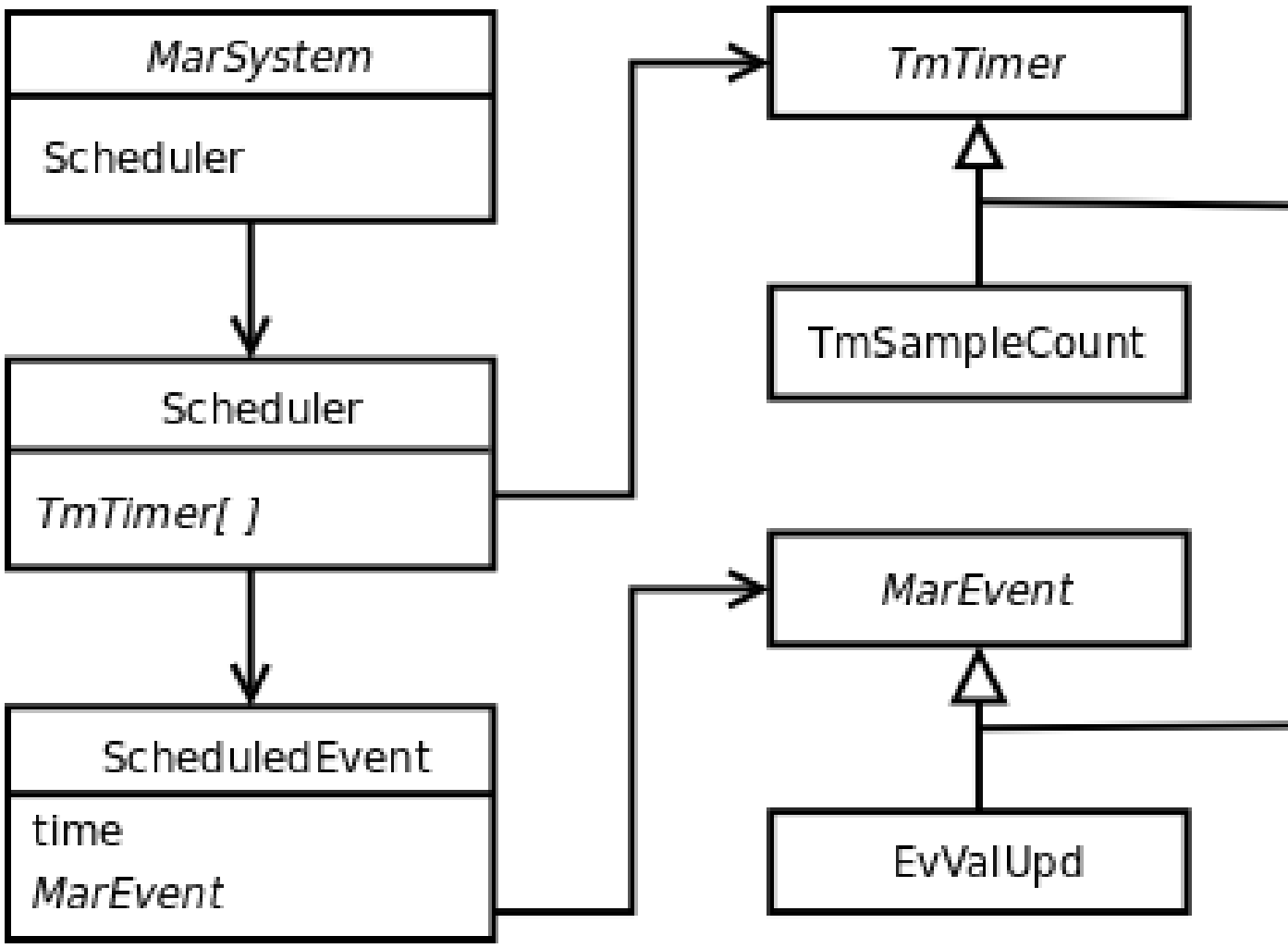


Figure 6.1: Scheduler class diagram.

6.5.2.1 MarSystem

Each MarSystem object has its own scheduler. This makes it possbile to post events on the MarSystem object directly. However, once a MarSystem is contained within another (within a Composite) it no longer responds to tick messages. This means that the schedulers in any of the contained objects will remain dormant. Therefore the only operational scheduler in a network is the one in the MarSystem being ticked.

Posting events on the scheduler is done through a number of `updctrl` methods. Each one takes a `TmTime` class as its first parameter.

```
void updctrl(EvEvent* me);
void updctrl(TmTime t, EvEvent* ev);
void updctrl(TmTime t, Repeat rep, EvEvent* ev);
void updctrl(TmTime t, std::string cname, MarControlPtr control);
```

```
void updctrl(TmTime t, Repeat rep, std::string cname, MarControlPtr control);
```

Additional methods for adding and removing timers and discovering the current time on a timer are available. The `updtimer` method is provided to modify timer parameters at run-time.

```
mrs_natural getTime(std::string timer_name);
void updtimer(std::string cname, TmControlValue value);
void addTimer(std::string class_name, std::string identifier);
void addTimer(TmTimer* t);
void removeTimer(std::string name);
```

6.5.2.2 Scheduler

It is the schedulers job to see that events are passed to the correct timer when they are posted. On each network tick the scheduler prompts each of the timers it manages to dispatch their pending events.

6.5.2.3 Timers

Timers define a control rate on which events may be scheduled. It is also the job of the timer to manage a queue of events and dispatch them at their scheduled dispatch time.

Creating a custom timer is simply a matter of defining its control rate, its units (ie seconds), and implementing a function to count the elapsed time between ticks. See the '`TmSampleCount`' timer as an example of a custom timer.

6.5.2.4 Events

Events are the actions that happen at specified points in time. In Marsyas events inherit from the '`EvEvent`'. Custom events are constructed by inheriting from '`EvEvent`' and overriding the `dispatch` and `clone` methods.

The overridden `EvEvent::dispatch` method is where the custom event action is defined. Since Marsyas is not threaded the network will block during dispatch. This could result in breaks in audio for real-time audio if the action takes too much time.

The `clone` method is intended to be used by the '`TmTimer`' to copy the event when it's posted. This would force the user to take care of its deletion and avoid confusion about who must do this. At this time clone is not used for this task so that once an '`EvEvent`' is posted it is under the control of the '`TmTimer`' it was posted on. It should not be reposted or deleted by the user. The search is on for a better solution.

'`EvEvent`' supports repetition without having to create new events. The `setRepeat(Repeat r)` method takes a '`Repeat`' object that defines how to repeat the event. The default behaviour is no repetition. A true value from the `repeat()` method tells the scheduler to repeat the event. This method queries the supplied '`Repeat`' object. The `getRepeatInterval()` returns the repeat rate. The `repeat_interval(string interval)` may be used to convert the supplied interval to a count. It is used in the `EvEvent::doRepeat()` method.

6.5.2.5 Repeat

Repetition of events is defined using the '`Repeat`' class. This class is simply a tuple of the repetition time interval and repetition count. There are three ways to define repetition.

`Repeat()` defines no repetition. `Repeat(string interval)` defines an infinite repetition at a rate specified by the supplied interval. `Repeat(string interval, mrs_natural count)` defines a finite repetition of count repeats to occur every interval.

Time is specified as a single string without a timer name. It is assumed that the specified interval time will be meaningful on the timer that the event is posted in.

6.5.2.6 TmTime

Time is specified using the '`TmTime`' class as `TmTime(string timer, string time)`. The first parameter is the name of the timer on which the second parameter has meaning. As an example, `TmTime("TmSampleCount/Virtual","5s")` specifies 5 seconds from the point of time it is used on the TmSampleCount timer called Virtual.

6.5.2.7 TmTimerManager

Rather than instantiating timers and adding them to the scheduler using the `MarSystem::addtimer(TmTimer* tmr)` method, timers may be specified and added by name using the `MarSystem::addtimer(string name, string ident)` method where name is the timer name, ie TmSampleCount, and ident is the unique identifier, ie Virtual. Of course, if the timer name is not known then this method will fail. New timers can be added to the factory using the method laid out in '`TmTimerManager.cpp`'. See [Timer Factory], page 108.

7 System details

7.1 Library reference

7.1.1 Source documentation

MarSystems are documented in the source documentation available at `http://marsyas.info/docs/index.html`. Most MarSystems are organized into groups.

7.1.2 MarSystem groups

The groups are fairly self-explanatory, but we should clarify a few groupings:

- **Processing**: audio block => audio block. Examples are `Gain` and `Filter`
- **Analysis**: audio block => other block. Examples are `Spectrum` and `Rms`.
- **Synthesis**: other block => audio block. Examples are `SineSource` and `NoiseSource`.

There is one special group: `Basic` Processing. This is a subset of the `Processing` group.

To see all MarSystems, look at *Data Structures* instead of *Main Page*.

7.1.3 Limited documentation

The "main" MarSystems are fairly well documented, but many MarSystems lack even basic documentation. In this case, the only option is to read the source code.

> **Help wanted: missing info!**
>
> Once you have learned how to use an undocumented MarSystem, please help document it. The source documentation only requires adding a special comment in the source file, so it is very easy! Please see Section "Contributing source documentation" in *Marsyas Developer's Manual*.
>
> *If you can fill in any details, please see Section "Contributing documentation" in Marsyas Developer's Manual.*

7.2 Predefined types

7.2.1 Variables types

Marsyas contains some predefined, portable data types:

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

The use of these variable types instead of normal C++ like `int` and `double` is **strongly** recommended.

Warning: a number like '100' is interpreted as a *mrs_natural*. If you want to pass this value to a *mrs_real*, you must specify '100.0'.

7.2.2 Constants

A number of useful constants, such as PI and TWOPI (at double precision instead of float) are defined in 'marsyas/common.h'.

7.3 Realvec

The basic unit of data in Marsyas is the realvec, which is essentially a one- or two-dimensional array of real numbers. All data flows from MarSystem to MarSystem in the form of realvecs; the actual function which does the main work of every MarSystem is

```
void
MarSystem_name::myProcess(realvec& in, realvec& out)
{
...
}
```

In other words, each MarSystem takes a realvec as input, and writes a realvec as output.

7.3.1 Reading and writing to a realvec

Realvecs may be accessed using var_name(index) = value.

Non-pointer way (allocation in stack)

```
mrs_natural i,j;

realvec foo;
foo.create(10);
for (i=0; i<10; i++) {
foo(i) = i;
}
for (i=0; i<10; i++) {
cout<<foo(i)<<endl;
}

realvec bar;
bar.create(5,10);
for (i=0; i<5; i++) {
for (j=0; j<10; j++) {
bar(i,j) = i+j;
}
}
```

Pointer way (allocation on heap)

```
// pointer for this example
realvec *baz;
```

```
// automatically calls calls create
baz = new realvec(10,20);

// we could do it this way if we wanted (instead of the above line)
//baz = new realvec;
//baz->create(10,20);

for (i=0; i<10; i++) {
for (j=0; j<20; j++) {
(*baz)(i,j) = i+j;
}
}
delete baz;  // don't forget to free the allocated memory
```

7.3.2 Resizing a realvec

The size of a realvec may be changed dynamically; you may also get the current size.

```
stretch();

getSize();
// for two-dimensional realvecs
getCols();
getRows();
```

If you do not know how big your realvec should be, use `stretchWrite(...)`. This function checks to make sure that the realvec is big enough; if not, it resizes the realvec (by doubling the size) so that it can store the value. This function is much slower than normal writing.

7.3.3 Realvec arithmetic

Realvecs may be set (=), added, subracted, and the like.

```
// realvec foo created, filled with data, etc.
...

realvec bar;
bar = foo;
// bar is now the same size and contains the same data

// arithmetic
baz = foo + bar;
baz = foo + 3;
```

7.3.4 Applying a function to a realvec

Use apply().

```
void
TranscriberExtract::toMidi(realvec* pitchList)
{
```

```
    pitchList->apply( hertz2pitch );
}
```

7.3.5 Realvec input and output to a file

```
/* you can do
realvec foo;
...
cout<<foo;
file<<foo;
*/
friend std::ostream& operator<<(std::ostream&, const realvec&);
friend std::istream& operator>>(std::istream&, realvec&);


// does the "file<<foo" method
void write(std::string filename) const;
void read(std::string filename);


// input/output functions for line-separated text files
void readText(std::string filename);
void writeText(std::string filename);
```

7.3.6 Statistical analysis

This is an incomplete list:

```
maxval();
minval();
mean();
median();
sum();
std();
var();
meanObs();
stdObs();
varObs();
normObs();
sort();
abs();
sqr();
sqroot();
covariance();
covariance2();
correlation();
det();
divergenceShape();
bhattacharyyaShape();
```

7.3.7 Other realvec functions

For a complete (and up-to-date) list of all realvec functions, please see the file '`marsyas/realvec.h`'

7.4 System limitations

There are many bugs in Marsyas (far too many to list here!), but there *are* a few issues which are more fundamental. We list these issues here, along with workarounds.

7.4.1 Stereo vs. mono in a spectrum

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.

8 Scripting

MarSystems may be used without programing in C++; the bindings for Python, Lua, Ruby, and Java allow the use of other languages.

8.1 Interactive python

8.1.1 Getting started with python

The `WITH_SWIG` and `WITH_SWIG_PYTHON` options in the CMake build system must have been set *ON*.

8.1.2 Swig python bindings bextract example

```
# bextract implemented using the swig python Marsyas bindings
# George Tzanetakis, January, 16, 2007

import marsyas

# Create top-level patch
mng = marsyas.MarSystemManager()
fnet = mng.create("Series", "featureNetwork")

# functional short cuts to speed up typing
create = mng.create
add = fnet.addMarSystem
link = fnet.linkControl
upd = fnet.updControl
get  = fnet.getControl

# Add the MarSystems
add(create("SoundFileSource", "src"))
add(create("TimbreFeatures", "featExtractor"))
add(create("TextureStats", "tStats"))
add(create("Annotator", "annotator"))
add(create("WekaSink", "wsink"))

# link the controls to coordinate things
link("mrs_string/filename", "SoundFileSource/src/mrs_string/filename")
link("mrs_bool/hasData", "SoundFileSource/src/mrs_bool/hasData")
link("WekaSink/wsink/mrs_string/currentlyPlaying","SoundFileSource/src/mrs_string/currentlyl
link("Annotator/annotator/mrs_natural/label", "SoundFileSource/src/mrs_natural/currentLabel
link("SoundFileSource/src/mrs_natural/nLabels", "WekaSink/wsink/mrs_natural/nLabels")

# update controls to setup things
upd("TimbreFeatures/featExtractor/mrs_string/disableTDChild", marsyas.MarControlPtr.from_st
```

```
upd("TimbreFeatures/featExtractor/mrs_string/disableLPCChild", marsyas.MarControlPtr.from_s
upd("TimbreFeatures/featExtractor/mrs_string/disableSPChild", marsyas.MarControlPtr.from_st
upd("TimbreFeatures/featExtractor/mrs_string/enableSPChild", marsyas.MarControlPtr.from_str
upd("mrs_string/filename", marsyas.MarControlPtr.from_string("bextract_single.mf"))
upd("WekaSink/wsink/mrs_string/labelNames",
                          get("SoundFileSource/src/mrs_string/labelNames"))
upd("WekaSink/wsink/mrs_string/filename", marsyas.MarControlPtr.from_string("bextract_pytho

# do the processing extracting MFCC features and writing to weka file
previouslyPlaying = ""
while get("SoundFileSource/src/mrs_bool/hasData").to_bool():
        currentlyPlaying = get("SoundFileSource/src/mrs_string/currentlyPlaying").to_string
        if (currentlyPlaying != previouslyPlaying):
                print "Processing: " +  get("SoundFileSource/src/mrs_string/currentlyPlayin

        fnet.tick()

        previouslyPlaying = currentlyPlaying
```

9 Writing applications

9.1 Including libraries and linking

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

9.1.1 ... using qmake

The easiest way to compile Marsyas programs is to use qmake. You may do this even if you used autotools to configure Marsyas itself; the author of this documentation uses autotools to build Marsyas, and qmake to build his programs.

Create a '`my_program_name.pro`' file:

```
#  your files
SOURCES = my_program_name.cpp
#HEADERS += extra_file.h
#SOURCES += extra_file.cpp


### if running inside the source tree
### adjust as necessary:
MARSYAS_INSTALL_DIR = ../../
INCLUDEPATH += $$MARSYAS_INSTALL_DIR/src/marsyas/
LIBPATH += $$MARSYAS_INSTALL_DIR/lib/release/


### if installed elsewhere
#MARSYAS_INSTALL_DIR = /usr/local
#MARSYAS_INSTALL_DIR = ${HOME}/usr/
#INCLUDEPATH += $$MARSYAS_INSTALL_DIR/marsyas/
#LIBPATH += $$MARSYAS_INSTALL_DIR/lib/


### basic OS stuff; do not change!
win32-msvc2005:LIBS += marsyas.lib
unix:LIBS += -lmarsyas -L$$MARSYAS_INSTALL_DIR/lib
!macx:LIBS += -lasound
macx:LIBS += -framework CoreAudio -framework CoreMidi -framework CoreFoundation
```

Then type `qmake` to generate a Makefile. Now you many compile normally.

9.1.2 ... writing your own Makefile

We *highly* recommend that you use qmake to create a Makefile. However, if you enjoy swearing at your computer and cursing k&r, rms, gnu, and every other three-letter programmer acronym in existence, go ahead. read on.

Here are sample Makefiles to get you started:

Makefile.linux

```
SOURCE = helloworld.cpp
TARGET = helloworld
#MARSYAS_INSTALL = /usr/local/
MARSYAS_INSTALL = ${HOME}/usr/

all:
	rm -f *.o
	g++ -Wall -O2 -I./ -I${MARSYAS_INSTALL}/include/marsyas -c \
		-o ${TARGET}.o ${SOURCE}
	g++ -L${MARSYAS_INSTALL}/lib -o ${TARGET} ${TARGET}.o -lmarsyas \
		-lasound
```

Makefile.osx

```
SOURCE = helloworld.cpp
TARGET = helloworld
#MARSYAS_INSTALL = /usr/local/
MARSYAS_INSTALL = ${HOME}/usr/

all:
	rm -f *.o
	g++ -Wall -O2 -I./ -I${MARSYAS_INSTALL}/include/marsyas -c \
		-o ${TARGET}.o ${SOURCE}
	g++ -L${MARSYAS_INSTALL}/lib -o ${TARGET} ${TARGET}.o -lmarsyas \
		-framework CoreAudio -framework CoreMidi -framework CoreFoundation
```

9.1.3 ... on Windows Visual Studio

Create the `.cpp` and `.h` files you will be working with in the project. Don't worry about the VS project file, it will be generated by qmake.

Type `qmake -project` at the command line to generate the .pro file, which Qt uses to create the Makefile (or you can make your own as above). Don't worry about linking to external libraries, we will take care of that in Visual Studio.

Once you have a `.pro` file, type

```
qmake -tp vc -spec win32-msvc2008
```

Or equivalent based on your system. This will generate a .vcproj file. Open this in Visual Studio.

Navigate to Tools->Options->Projects and Solutions->VC++ Directories and ensure:

- Executable files includes qt's bin directory Include files includes %qt-root%\include, %marsyas-root%\src\marsyas, %marsyas-root%\src\marsyasqt\, %marsyas-root% (for config.h)
- Library files include dsdk, marsyas, and qt's lib directories

- In project properties->linker: General-> Ignore import libary: No Link Library Dependencies: Yes Input-> Additional Dependencies: add `c:\marsyas\build\lib\Release\marsyas.lib c:\marsyas\build\lib\Release\marsyasqt.lib "C:\Program Files\Microsoft DirectX SDK (November 2008)\Lib\x86\dsound.lib" imm32.lib winmm.lib ws2_32.lib` System-> Subsystem: Console

These settings worked for the author.

9.2 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.

9.2.1 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")->to<mrs_bool
	{
		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);
}
```

9.2.2 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")->to<
        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")->to<mrs_boo
        {
                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);
}
```

9.2.3 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/filename", "out");

        while ( pnet->getctrl("SoundFileSource/src/mrs_bool/notEmpty")->to<mrs_bool>() )
        {
                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);
}
```

9.2.4 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"));
        pnet->addMarSystem(mng.create("Gain", "g2"));

        while ( pnet->getctrl("SoundFileSource/src/mrs_bool/notEmpty")->to<mrs_bool>() )
        {
                pnet->tick();
                // gets data from the Spectrum for read only!
                const realvec& processedData =
                    pnet->getctrl("Spectrum/spk/mrs_realvec/processedData")->to<mrs_realve
                cout << "Original Spectrum = " << processedData << endl;

                // if we need to get the Spectrum and modify it, here is the way
                // to do it.  Notice that we must open a new scope using curly
                // brackets so that MarControlAccessor is automatically de-
stroyed
```

```
                // when we are finished modifing the realvec control.
                {
                        MarControlAccessor
                        acc(pnet->getctrl("Spectrum/spk/mrs_realvec/processedData"));
                        realvec&
                        processedData2 = acc.to<mrs_realvec>();

                        // we can now write to processedData without worries of
                        // breaking encapsulation
                        processedData2 *= 2.0;
                        cout << "Modified Spectrum = " << processedData2 << 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);
}
```

9.2.5 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 supported format"
	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 );
        }
}
```

9.3 Writing Qt4 applications

Writing applications that use Marsyas and Qt4 is quite simple, thanks to '`libmarsasqt`'.

> **Warning:** Marsyas compiled with autotools does not compile '`libmarsyasqt`' by default. To enable it, please pass `--enable-marsyasqt` to configure.

9.3.1 Including and linking to libmarsyasqt

Create a '`.pro`' file based on this template:

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


### if running inside the source tree
MARSYAS_INSTALL_DIR = ../../../
INCLUDEPATH += $$MARSYAS_INSTALL_DIR/src/marsyasqt_wrapper/
LIBPATH += $$MARSYAS_INSTALL_DIR/lib/release/


### if installed elsewhere
#MARSYAS_INSTALL_DIR = /usr/local
#MARSYAS_INSTALL_DIR = ${HOME}/usr/
#INCLUDEPATH += $$MARSYAS_INSTALL_DIR/src/marsyas/
#LIBPATH += $$MARSYAS_INSTALL_DIR/lib/


unix:LIBS += -lmarsyas -lmarsyasqt
unix:!macx:LIBS += -lasound
macx:LIBS += -framework CoreAudio -framework CoreMidi -framework CoreFoundation
```

It is **highly** recommended that you separate the Qt code (ie `mainwindow.cpp`) from the Marsyas code (ie `backend.cpp`).

9.3.2 MarSystemQtWrapper

The actual interaction between Qt and Marsyas is performed with a `MarSystemQtWrapper` object. Add the following lines to your header file that deals with Marsyas:

```
#include "MarSystemQtWrapper.h"
using namespace MarsyasQt;
```

You may now create a pointer to a MarSystemQtWrapper in your object:

```
MarSystemQtWrapper *mrsWrapper;
```

To use this object in your source file, create a MarSystem like normal, then pass it to a new `MarSystemQtWrapper` object:

```
MarSystemManager mng;
playbacknet = mng.create("Series", "playbacknet");
... set up playbacknet...

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

... do something for a while...
mrsWrapper->stop();
delete mrsWrapper;
```

9.3.3 Passing controls to a MarSystemQtWrapper

Since your MarSystem is wrapped up, you cannot access controls using the normal methods. Instead, you must use a MarControlPtr:

```
MarControlPtr filenamePtr;
MarControlPtr positionPtr;

filenamePtr = mrsWrapper->getctrl("SoundFileSource/src/mrs_string/filename");
positionPtr = mrsWrapper->getctrl("SoundFileSource/src/mrs_natural/pos");

mrsWrapper->updctrl(filenamePtr,fileName);
... wait a bit...
newPos = (mrs_natural) positionPtr->to<mrs_natural>();
```

9.3.4 Other classes in MarsyasQt

MarsyasQt includes more than just `MarSystemQtWrapper`. Please refer to source documentation in Section 7.1 [Library reference], page 69.

9.3.5 Qt4 example

To get started, look at the Qt4 tutorial files:

tutorial.pro

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


### if running inside the source tree
MARSYAS_INSTALL_DIR = ../../../
```

```
INCLUDEPATH += $$MARSYAS_INSTALL_DIR/src/marsyasqt/
INCLUDEPATH += $$MARSYAS_INSTALL_DIR/src/marsyas/
LIBPATH += $$MARSYAS_INSTALL_DIR/lib/release/


### if installed elsewhere
#MARSYAS_INSTALL_DIR = /usr/local
#MARSYAS_INSTALL_DIR = ${HOME}/usr/
#INCLUDEPATH += $$MARSYAS_INSTALL_DIR/marsyas/
#LIBPATH += $$MARSYAS_INSTALL_DIR/lib/


### basic OS stuff; do not change!
win32-msvc2005:LIBS += marsyas.lib marsyasqt.lib
unix:LIBS += -lmarsyas -lmarsyasqt -L$$MARSYAS_INSTALL_DIR/lib
!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 "MarSystemQtWrapper.h"

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

class MarBackend: public QObject
{
        Q_OBJECT

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

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

signals:
        void changedBackendPosition(mrs_natural value);

private:
```

```
// in order to make the MarSystem act like a Qt object,
// we use this wrapper:
       MarSystemQtWrapper *mrsWrapper;
// ... and these pomrs_naturalers:
       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 pomrs_naturalers so that we can mrs_naturalerface with the net-
work
// 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(mrs_natural vol)
{
        mrs_real newGain = vol/100.0;
        mrsWrapper->updctrl(gainPtr, newGain);
}

void MarBackend::getBackendPosition()
{
        mrs_natural newPos = (mrs_natural) 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.

9.3.6 Other Qt4 issues

TODO: beautify

Question and answer:

```
I'd like to analyze sound file to extract pitches and amplitudes.  I
have working code without using MarSystemQtWrapper, but for GUI display
reasons, I'd like to do this analysis in a separate thread.  (ie a
MarSystemQtWrapper)

Is there any way to do this kind of thing inside a MarSystemQtWrapper?

        while (allNet->getctrl("mrs_bool/hasData")->toBool())
                allNet->tick();
```

add the control you are interested to the monitored controls by calling trackctrl (or something like that) in MarSystemQtWrapper. The MarSystemQtWrapper will emit a signal either any time the control is changed or at period intervals depending on the withTimer settting. Either way you can connect the signal ctrlChanged(MarControlPtr cname);

to a slot in your GUI and then call allnet->pause() when that happens.

—– this is working on Meaws, but I'll need to read my code again to figure out how it works. Fortunately, I write nice clean readable code. :) -gp

UPDATE: look at src/qt4apps/regressionChecks instead.

Converting between QString and mrs_string

```
QString qs = "Hello world!";
mrs_string mrs_s = qs.toStdString();
```

```
mrs_string mrs_s = "foo.wav";
QString qs = QString( mrs_s.c_str() );
```

9.4 Other programming issues

9.4.1 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"
```

9.5 Interoperability

9.5.1 Open Sound Control (OSC)

Help wanted: missing info!

can be done; read the source code.

If you can fill in any details, please see Section "Contributing documentation" in Marsyas Developer's Manual.

9.5.2 WEKA

Help wanted: missing info!

can be done; read the source code.

If you can fill in any details, please see Section "Contributing documentation" in Marsyas Developer's Manual.

9.5.3 MATLAB

MATLAB is a high-level technical computing language and interactive environment for algorithm development, data visualization, data analysis, and numeric computation http://www.mathworks.com/

Marsyas supports the use of the MATLAB engine API for exchanging data at runtime between Marsyas C++ code and MATLAB enviroment. This allows effortlessly passing and getting data to/from MATLAB inside Marsyas, and have access to MATLAB plotting facilities, numerical routines, toolboxes, etc. More information about the MATLAB engine can be found at in their technical documentation.

Passing data from Marsyas to MATLAB => MATLAB_PUT()

To export data from Marsyas into MATLAB, simply use the macro `MATLAB_PUT(marsyasVar, MATLABvarName)`.

MATLAB_PUT() can be used with C++ int, float, double, mrs_real, mrs_natural, mrs_complex, std::string, std::vector, Marsyas::realvec, among other data types. For a complete list of the types supported, please refer to '`src/marsyas/MATLAB/MATLABengine.h`'.

```
//create a Marsyas realvec (i.e. a 2 by 3 matrix):
realvec marsyas_realvec(2,3);
marsyas_realvec(0,0) = 0.0;
marsyas_realvec(0,1) = 0.1;
marsyas_realvec(0,2) = 0.2;
```

```
marsyas_realvec(1,0) = 1.0;
marsyas_realvec(1,1) = 1.1;
marsyas_realvec(1,2) = 1.2;

// Send the realvec to MATLAB, and call it marsyasMatrix in MATLAB
MATLAB_PUT(marsyas_realvec,"marsyasMatrix");
// marsyasMatrix is now a normal matrix of doubles in MATLAB
// environment, which can be used as any regular MATLAB matrix.
```

Please refer to 'src/apps/mudbox/mudbox.cpp' for more examples of using the MATLAB engine API in Marsyas (look for the toy_with_MATLABengine() function).

Executing commands in MATLAB from Marsyas C++ code => MATLAB_EVAL()

It is possible to execute commands in MATLAB from Marsyas C++ code, as if they were being input into the MATLAB command line. For this use the macro `MATLAB_EVAL(command)`.

```
//create a Marsyas realvec (i.e. a 2 by 3 matrix):
realvec marsyas_realvec(2,3);
marsyas_realvec(0,0) = 0.0;
marsyas_realvec(0,1) = 0.1;
marsyas_realvec(0,2) = 0.2;
marsyas_realvec(1,0) = 1.0;
marsyas_realvec(1,1) = 1.1;
marsyas_realvec(1,2) = 1.2;

// Send the realvec to MATLAB, and call it marsyasMatrix in MATLAB
MATLAB_PUT(marsyas_realvec,"marsyasMatrix");

// now we can, for e.g., ask MATLAB to transpose the passed matrix...
MATLAB_EVAL("marsyasMatrix = marsyasMatrix'");
// and, why not, plot the matrix...
MATLAB_EVAL("imagesc(marsyasMatrix)");
// run our own m.files...
MATLAB_EVAL("myFunction(marsyasMatrix);");
// or execute any other MATLAB command!
MATLAB_EVAL("whos");
// Pretty cool, hum? ;-)
```

Please refer to src/apps/mudbox/mudbox.cpp for more examples of using the MATLAB engine API in Marsyas (look for the toy_with_MATLABengine() function).

Getting data from MATLAB into Marsyas => MATLAB_GET()

To import data into Marsyas from MATLAB, simply use the macro `MATLAB_GET(MATLABvarName, marsyasVar);`

MATLAB_GET() can be used for getting data from MATLAB into C++ int, float, double, mrs_real, mrs_natural, mrs_complex, std::vector, Marsyas::realvec, among other data types. For a complete list of the types supported, please refer to src/marsyas/MATLAB/MATLABengine.h.

```
//create a Marsyas realvec (i.e. a 2 by 2 matrix):
realvec marsyas_realvec(2,2);
marsyas_realvec(0,0) = 0.0;
marsyas_realvec(0,1) = 0.1;
marsyas_realvec(1,0) = 1.0;
marsyas_realvec(1,1) = 1.1;

// Send the realvec to MATLAB, and call it marsyasMatrix in MATLAB
MATLAB_PUT(marsyas_realvec,"marsyasMatrix");

// now we can, for e.g., ask MATLAB to transpose the passed matrix...
MATLAB_EVAL("marsyasMatrix = marsyasMatrix'");
// and calculate its determinant
MATLAB_EVAL("matrixDet = det(marsyasMatrix);");

// we can now get the determinant value back into Marsyas
// (note: Marsyas already has its own efficient C++
// implementation for determinant calculation of realvecs!
// This is just an example of what can be done with the Marsyas MATLAB engine.
mrs_real det;
MATLAB_GET("matrixDet", det);
cout << det << endl;

// but we can also get the (now transposed) matrix back into Marsyas:
MATLAB_GET("marsyasMatrix", marsyas_realvec);
cout << marsyas_realvec << endl;
```

Please refer to src/apps/mudbox/mudbox.cpp for more examples of using the MATLAB engine API in Marsyas (look for the toy_with_MATLABengine() function).

9.5.4 Python

Here are instructions for python/ruby SWIG bindings. These instructions assume Python and/or Ruby are installed. You might need to install a more recent version of Python (for example MacPython from the Python website) than the one that comes pre-bundled with OS X depending on the version of your system.

At the top level directory do :

./configure –enable-bindings

1. make
2. sudo make install

Assuming that python and ruby have been locating your bindings have been compiled and installed. To check that your bindings are installed.

1. Start any python IDE or shell (for example python or ipython)
2. At the prompt type:
3. import marsyas
4. If you get no error your bindings have been installed

5. Go to swig/python
6. Try some of the examples such as test2.py or bextract.py

9.5.5 OCaml

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

9.5.6 SonicVisualiser Vamp Plugins

9.5.6.1 Instalation

Requirements:

- **CMake** (http://www.cmake.org/).
- **SonicVisualiser** (http://www.sonicvisualiser.org/).
- **Vamp Plugin SDK:**
  - Latest revision: `$ svn co https://vamp.svn.sourceforge.net/svnroot/vamp/vamp-plugin-sdk/trunk vamp-plugin-sdk`.
  - Distribution: http://sourceforge.net/projects/vamp/files/vamp-plugin-sdk/.
  - Build and install the plugin sdk using the supplied instructions.
    (vamp-simple-host, a command-line host for Vamp plugins, is installed along)
- **Vamp Tester [optional]**
  A command line application for testing plugins, giving you a very detailed report about the compliance of the plugin:
  - Latest revision: `$ svn co https://vamp.svn.sourceforge.net/svnroot/vamp/vamp-plugin-tester/trunk vamp-plugin-tester`.
  - Distribution (source + binaries): http://sourceforge.net/projects/vamp/files/vamp-plugin-te
  - Add vamp-plugin-tester path to environment `PATH`.

MacOSX:

(Tested in 10.6 with gcc 4.2 for i386 and x86_64, linking to both the static and dynamic build of Marsyas.
Note that x86_64 is not supported by Sonic Visualiser yet, but fat binaries with both architectures work fine.)

1. **Building Marsyas Vamp Plugin:**

   - `$ cmake-gui`
     `WITH_VAMP ON`;
     `MARSYAS_STATIC OFF` (static linking not supported)
     `VAMP_LIBRARY`: dynamic/static version of libvamp-sdk, from the install path -> `/usr/local/lib/libvamp-sdk.dylib` or `/usr/local/lib/libvamp-sdk.a`
     `VAMP_INCLUDE_DIR`: vamp sdk src main path -> `/usr/local/include/`
     (choose your own paths if different)
   - `$ make`
     (If you generate an XCode project, the mvamp target won't be included so you'll need to build using plain "make")

(vamp plugin compiled as shared dynamic library at `marsyas_path/build_dir/lib/libmvamp.dylib`)

2. **Marsyas Vamp Plugin Integration in SonicVisualiser Transforms:**
   - Copy `libmvamp.dylib` to one of the following paths:

     `~/Library/Audio/Plug-Ins/Vamp (user use)`
     `/Library/Audio/Plug-Ins/Vamp (system use)`

   (`mvamp-plugins.cat`, in pre-compiled version or in `path/to/marsyas/src/mvamp/`, is a txt file for organizing the plugin functions within SonicVisualiser Transforms - for such you may also copy it to the chosen path above, yet it seems to be unnecessary)
3. **(Optional) Testing Plugin with vamp-plugin-tester:**
   - `$ VAMP_PATH=Path/To/Vamp_Plugins_Directory`
   - `$ vamp-plugin-tester -a`
4. **(Optional) Testing Plugin with vamp-simple-host:**
   - `$ VAMP_PATH=Path/To/Vamp_Plugins_Directory`
   - List the plugin libraries and Vamp plugins in the library search path: `$ vamp-simple-host -l`
   - Run the plugin: `$ vamp-simple-host [-s] pluginlibrary[.dylib]:plugin file.wav [outputno] [-o out.txt]`
     (check pluginlibrary and plugin name with `-l` above)

Linux OS:

(Tested in Linux Ubuntu 9.04-32bits with gcc 4.3 -> may work on others)

1. **Building Marsyas Vamp Plugin:**
   (Alternatively simply download pre-compiled plugin for Linux i686 at http://marsyas.info/download/vamp_plugins)
   - `$ cmake-gui`
     `WITH_VAMP ON`;
     `MARSYAS_STATIC OFF` (static linking not supported)
     `VAMP_LIBRARY`: dynamic/static version of libvamp-sdk, from the install path -> `/usr/local/lib/libvamp-sdk.so` or `/usr/local/lib/libvamp-sdk.a`
     `VAMP_INCLUDE_DIR`: vamp sdk src main path -> `/usr/local/include/`
     (choose your own paths if different)
   - `$ make`
     (vamp plugin compiled as shared dynamic library at `marsyas_path/build/lib/libmvamp.so`)
2. **Marsyas Vamp Plugin Integration in SonicVisualiser Transforms:**
   - Copy `libmvamp.so` (or pre-compiled `mvamp.so`) to one of the following paths:

     `/home/(user)/vamp (user use)`
     `/usr/local/lib/vamp (system use)`

   (`mvamp-plugins.cat`, in pre-compiled version or in `path/to/marsyas/src/mvamp/`, is a txt file for organizing the plugin functions within SonicVisualiser Transforms - for such you may also copy it to the chosen path above, yet it seems to be unnecessary)
3. **(Optional) Testing Plugin with vamp-plugin-tester:**

- `$ VAMP_PATH=Path/To/Vamp_Plugins_Directory`
- `$ vamp-plugin-tester -a`

4. **(Optional) Testing Plugin with vamp-simple-host:**
   - `$ VAMP_PATH=Path/To/Vamp_Plugins_Directory`
   - List the plugin libraries and Vamp plugins in the library search path: `$ vamp-simple-host -l`
   - Run the plugin: `$ vamp-simple-host [-s] pluginlibrary[.so]:plugin file.wav [outputno] [-o out.txt]`
     (check pluginlibrary and plugin name with `-l` above)

Windows OS:
(Tested in Windows Vista-32bists with VS2008 and Windows 7 with VS2010 -> may work on others)

1. **Building Marsyas Vamp Plugin:**
   - Run CMake-GUI:
     `WITH_VAMP ON`;
     `MARSYAS_STATIC OFF` (static linking not supported)
     `VAMP_LIBRARY`: static version of libvamp-sdk -> `path\to\vamp-plugin-sdk-x.x\build_dir\Release\VampPluginSDK.lib`
     `VAMP_INCLUDE_DIR`: vamp sdk src main path -> `path\to\vamp-plugin-sdk-x.x`
     (choose your own paths if different)
   - Load `marsyas_path\build\Marsyas.sln` in Visual Studio
   - Open mvamp project proprieties:
     - Go to `Configuration Properties -> Linker -> Command Line -> Additional options`
     - Add `/EXPORT:vampGetPluginDescriptor`
   - Build marsyas + mvamp in Release mode and **in Win32** architecture.
     (x64 not currently supported by SonicVisualiser)
2. **Marsyas Vamp Plugin Integration in SonicVisualiser Transforms:**
   - Copy `mvamp.dll` from `marsyas_path\build\bin\Release\` to `C:\Program Files\Vamp Plugins\` (default `VAMP_PATH`)
     (if used different path for Vamp Plugins define it in environment `VAMP_PATH`)
   - Run-time linking to Marsyas (one of the two options below):
     - Set `marsyas_path\build\bin\Release\` to environment variable `PATH`
     - Copy `marsyas.dll from marsyas_path\build\bin\Release\` to SonicVisualiser main directory.
3. **(Optional) Testing Plugin with vamp-plugin-tester:**
   - `$ vamp-plugin-tester.exe -a`
4. **(Optional) Testing Plugin with vamp-simple-host:**

- List the plugin libraries and Vamp plugins in the library search path: `$ vamp-simple-host.exe -l`

- Run the plugin: `$ vamp-simple-host.exe [-s] pluginlibrary[.dylib]:plugin file.wav [outputno] [-o out.txt]`
  (check pluginlibrary and plugin name with `-l` above)

9.5.6.2 Writing Plugin

Check tutorial on http://www.vamp-plugins.org/develop.html

9.6 Using and Extending the Scheduler

Writing applications using the scheduler may require writing additional timer and event types. This section will try to explain how to customize the scheduler for your own applications.

9.6.1 Using the Scheduler

In the example below an event is created and posted to the network to set the gain control to 0 (silence) two seconds after the network starts processing. This is accomplished by creating a new EvValUpd event which performs a `setctrl` call when dispatched.

The EvValUpd event requires a MarSystem pointer to act on - we use the topmost `series` object because we have a pointer to it. The event also requires a control path referenced to the supplied MarSystem pointer. Finally, it requires a MarControlValue to set the control value to on dispatch. This value should have the same type as the one specified in the control path otherwise the Marsyas system will report an error at dispatch time.

To post the event to a timer the MarSystem `updctrl` call is used. The first parameter to `updctrl` is the dispatch time. This indicates a scheduled event call to the MarSystem and it is passed on to the scheduler. In this case the `TmTime` class is supplied the name of the timer that the event is to be posted on along with the time of event dispatch. The default timer for every MarSystem is a `TmSampleCount` timer, which counts the number of samples processed, with the name Virtual. The TmSampleCount timer understands the units `us, ms, s, m, h` corresponding to microseconds, milliseconds, seconds, minutes, hours, respectively. The supplied time is converted to samples by calling the static method `mrs_natural Marsyas::time2samples(string time, mrs_real srate)` in the '`Conversions.cpp`'.

```
MarSystemManager mng;

MarSystem* series = mng.create("Series", "series");
series->addMarSystem(mng.create("SineSource", "src"));
series->addMarSystem(mng.create("Gain", "g"));
series->addMarSystem(mng.create("AudioSink", "snk"));

series->updctrl("AudioSink/snk/mrs_bool/initAudio", true);
series->updctrl("SineSource/src/mrs_real/frequency",440.0);
series->updctrl("Gain/g/mrs_real/gain",1.0);

EvValUpd* ev = new EvValUpd(series,"Gain/g/mrs_real/gain",0.0);
series->updctrl(TmTime("TmSampleCount/Virtual","2s"),ev);

while(true) series->tick();
```

Figure 9.1: Program using the scheduler to set gain to 0 after two seconds.

9.6.1.1 Repeating Events

We can repeat events using the event `set_repeat` method of the event. This method takes a `Repeat` value which is essentially a (rate,count) tuple. In the example below, two events are posted. One sets the volume to 0, the other to 1. By staggering their dispatch we can achieve a gating effect. Here we repeat the events forever by specifying the repeat rate only as in `Repeat("1s")`. To specify a finite repeat count we could set the repeat count to five as in `Repeat("1s",5)`. Gaze wonderously at the example.

```
MarSystemManager mng;

MarSystem* series = mng.create("Series", "series");
series->addMarSystem(mng.create("SineSource", "src"));
series->addMarSystem(mng.create("Gain", "g"));
series->addMarSystem(mng.create("AudioSink", "snk"));

series->updctrl("AudioSink/snk/mrs_bool/initAudio", true);
series->updctrl("SineSource/src/mrs_real/frequency",440.0);
series->updctrl("Gain/g/mrs_real/gain",1.0);

EvValUpd* ev_off = new EvValUpd(series,"Gain/g/mrs_real/gain",0.0);
ev_off->set_repeat(Repeat("1s")); // repeat forever
//ev_off->set_repeat(Repeat("1s",4)); // repeat four times

EvValUpd* ev_on = new EvValUpd(series,"Gain/g/mrs_real/gain",1.0);
ev_on->set_repeat(Repeat("1s"));

// stagger the dispatch of the events, off by 1 second
series->updctrl(TmTime("TmSampleCount/Virtual","1s"),ev_off);
series->updctrl(TmTime("TmSampleCount/Virtual","2s"),ev_on);

while(true) series->tick();
```

Figure 9.2: Program using two events that toggle the gain between 0 and 1 every second.

But how does this work under the hood, uhh, so to speak, you ask. For a pending event (whose time is now), the scheduler will remove it from the queue, call its dispatch method, check whether it should be repeated by calling its `repeat` method. If it is to be repeated, the rate will be read by calling the `repeat_interval()` method which reads the rate from the `Repeat` value originally supplied through the `set_repeat` method. A calculation of the next dispatch time is made and the event is reposted to the queue. See the `void TmTimer::dispatch()` method of '`TmTimer.cpp`' for the exact method.

9.6.2 Writing a new Timer

Timers control the scheduling of events with respect to some control rate. When writing new timers we don't want to have to worry about this scheduling activity. Instead we wish to define new control rates. No problem. All we have to do is inherit from the TmTimer class. For lack of a more creative idea this section will explain a slightly modified TmSampleCount timer.

When creating new timers we first create a class that inherits from TmTimer. This class is placed in the src/marsyas directory and added to the build system. Our example, requires a constructor that takes a MarSystem to read the time from. Our timer will read the "mrs_natural/onSamples" control to find out how long the interval of time is, in samples, between successive ticks. This will be used to advance our timer. On construction we call setReadCtrl which gets the control as a MarControlPtr for faster access than calling getctrl on each tick.

Whenever the scheduler is ticked, it will tick each of the timers it controls. These timers will call their `readTimerSrc()` method to advance their clocks. Our `readTimerSrc` method will read the onSamples control and return this value (elapsed time since last tick). Our timer is now operational.

Timers may also require the definition of special time units. In the case of real time we may want to define what milliseconds or seconds mean with respect to sample count. To do this we must override the `mrs_natural intervalsize(string interval)` method. For our timer, we will simply call the static method `time2samples(string)` defined in '`Conversions`' and pass it the string and the current sample rate. Now our timer supports time defined in samples, microseconds, milliseconds, seconds, minutes, and hours.

```
#ifndef MARSYAS_TM_SAMPLE_COUNT_H
#define MARSYAS_TM_SAMPLE_COUNT_H

#include "TmTimer.h"
#include "MarControlValue.h"
#include "MarSystem.h"

namespace Marsyas
{
  // forward declaration of MarSystem allows Scheduler.getctrl("insamples")
  // for scheduler count
  class MarSystem; // forward declaration

  class TmSampleCount : public TmTimer {
  protected:
    MarSystem* read_src_;
    MarControlPtr read_ctrl_;

  public:
    // Constructors
    TmSampleCount(MarSystem*);
    TmSampleCount(const TmSampleCount& s);
    virtual ~TmSampleCount();
    TmTimer* clone();

    void setReadCtrl(MarSystem* ms);
    mrs_natural readTimeSrc();
    mrs_natural intervalsize(std::string interval);

    virtual void updtimer(std::string cname, TmControlValue value);
};

}//namespace Marsyas

#endif
```

Figure 9.3: TmSampleCount header file example.

```
#include "TmSampleCount.h"
#include "MarSystem.h"
#include "Scheduler.h"

using namespace std;
using namespace Marsyas;

TmSampleCount::TmSampleCount(MarSystem* ms) : TmTimer("TmSampleCount","Virtual")
{
    setReadCtrl(ms);
}
TmSampleCount::TmSampleCount(const TmSampleCount& s) : TmTimer(s)
{
    setReadCtrl(s.read_src_);
}
TmSampleCount::~TmSampleCount(){ }

TmTimer* TmSampleCount::clone()
{
  return new TmSampleCount(*this);
}
void TmSampleCount::setReadCtrl(MarSystem* ms)
{
    read_src_=ms;
    if (read_src_!=NULL) read_ctrl_=read_src_->getctrl("mrs_natural/onSamples");
}

mrs_natural TmSampleCount::readTimeSrc()
{
    if (read_src_==NULL) {
        MRSWARN("TmSampleCount::readTimeSrc()  time source is NULL");
        return 0;
    }
    mrs_natural m = read_ctrl_->to<mrs_natural>();
    return m;
}
mrs_natural TmSampleCount::intervalsize(string interval)
{
    return (read_src_==NULL) ? 0 : time2samples(interval,read_src_->getctrl("mrs_real/isr
}
void
TmSampleCount::updtimer(std::string cname, TmControlValue value)
{
    bool type_error=false;
    if (cname=="MarSystem/source") {
        if (value.getType()==tmcv_marsystem) { setReadCtrl(value.toMarSystem()); }
        else type_error=true;
    }
    else
        MRSWARN("TmSampleCount::updtimer(string,TmControlValue)  unsupported contro
    if (type_error)
        MRSWARN("TmSampleCount::updtimer(string,TmControlValue)  wrong type to "+cn
```

9.6.2.1 Updating timers at run-time

The TmTimer class also supports communication through the updtimer method. An example of this is shown in the TmSampleCount timer above. This is not necessary for the operation of our timer but we might want to support the changing of timer parameters at run-time through the updtimer interface. To do this we simply override the `void updtimer(std::string cname, TmControlValue value)` method. We can now parse the supplied timer control path and set the appropriate value.

Timer control paths have the same format as MarSystem controls. For example, our timer could be accessed through:

```
marsys->updtimer("TmSampleCount/Virtual/MarSystem/source",marsys);
```

Figure 9.5: Setting timer parameters using the updtimer call.

The TmControlValue defines the allowable values that may be passed to timers. These values are limited to: `float, double, int, long, std::string, const char*, bool, MarSystem*`. However, one could always modify the TmControlValue class to add additional types. Be careful of values clashing such as NULL pointers and the integer value 0.

9.6.2.2 Timer Factory

New timers can be added to the Timer Factory by modifying '`TmTimerManager`'. Doing so allows the use of the addTimer method in '`MarSystem`' by simply specifying the type/name of the timer as opposed to creating a new timer.

```
net->addTimer("TmSampleCount","counter");
```

Figure 9.6: Adding a new timer to MarSystem net using the timer factory.

The instructions for adding new timers to the factory are contained in '`TmTimerManager`' and repeated here. Basically, a map is created from "TimerName"=>TimerConstructorObject. This makes it possible to use a map for fast access to specific timers and it prevents having to instantiate each Timer type at startup. The constructor object simply wraps the new operator so that it constructs objects only when requested.

- Add the timer's header file to the top of '`TmTimerManager.cpp`' as an #include.
- Underneath the includes, wrap the timer name in the macro `TimerCreateWrapper`.
- In the addTimers function wrap the timer name in the macro `registerTimer`.

9.6.3 Writing a new Event

Suppose we want to fade the volume down to silence using a Gain MarSystem. We could accomplish this using the scheduler and several EvValUpd events. Assuming the gain control is at 1.0 to begin we just issue 10 EvValUpd events each with a progressively lower volume amount: 0.9, 0.8, 0.7, ... and each at a time that is a little bit later than the previous. This is messy and repetitive. Why not make a new event, called EvRampCtrl, that encapsulates this behaviour.

We start by defining a new EvRampCtrl class that inherits from the '`EvEvent`' class. We define a constructor that takes a MarSystem to act upon, a control to modify, a starting

value, an ending value, and a step amount. The implicit assumption is that this event will only work on values of type `mrs_real` so that values we supply must all be of the correct type. The header and cpp files are supplied below.

We will need an additional variable to save the current adjustment value. Also, the event will need to repeat so we will maintain a repeat flag that is true until we pass the end value. We override the `repeat()` method of '`EvEvent`' to return the current value of the repeat flag. The required variables are shown in the header file below.

```
#ifndef MARSYAS_EV_RAMPCTRL_H
#define MARSYAS_EV_RAMPCTRL_H

#include <string>

#include "MarControl.h"
#include "EvEvent.h"
#include "TmControlValue.h"

namespace Marsyas
{
  class MarSystem; // forward declaration

  class EvRampCtrl : public EvEvent {
  protected:
    MarSystem* target_;
    std::string cname_;
    double start_, end_, value_, rate_;
    bool repeat_flag_;

  public:
    // Constructors
    EvRampCtrl(MarSystem* m, std::string cname, double start, double end, double rate)
    EvRampCtrl(EvRampCtrl& e);
    virtual ~EvRampCtrl();

    virtual EvRampCtrl* clone();

    // Set/Get methods
    void set(MarSystem* ms, std::string cname, double s, double e, double r);

    bool repeat() { return repeat_flag_; };

    // Event dispatch
    void dispatch();

  };

}//namespace Marsyas

#endif
```

Figure 9.7: EvRampCtrl event header file example.

```
#ifndef MARSYAS_EV_RAMPCTRL_H
#define MARSYAS_EV_RAMPCTRL_H

#include <string>

#include "MarControl.h"
#include "EvEvent.h"
#include "TmControlValue.h"

namespace Marsyas
{
  class MarSystem; // forward declaration

  class EvRampCtrl : public EvEvent {
  protected:
    MarSystem* target_;
    std::string cname_;
    double start_, end_, value_, rate_;
    bool repeat_flag_;

  public:
    // Constructors
    EvRampCtrl(MarSystem* m, std::string cname, double start, double end, double rate)
    EvRampCtrl(EvRampCtrl& e);
    virtual ~EvRampCtrl();

    virtual EvRampCtrl* clone();

    // Set/Get methods
    void set(MarSystem* ms, std::string cname, double s, double e, double r);

    bool repeat() { return repeat_flag_; };

    // Event dispatch
    void dispatch();

  };

}//namespace Marsyas

#endif
```

Figure 9.8: EvRampCtrl event header file example.

The main logic for our event is contained in the dispatch method. Basically, we check to see if we have passed the end value during ramping and if so set the repeat flag to false. The next time that the scheduler checks to see if the event repeats the scheduler will read

the false value and delete the event. If we have not passed the end value then we set the specified control to the current ramp value and decrement the current value by the ramp amount. The scheduler will see that the event is to be repeated, it will read the repeat rate amount, and repost the event to the queue.

```
#include "EvRampCtrl.h"
#include "MarSystem.h"

using namespace std;
using namespace Marsyas;

EvRampCtrl::EvRampCtrl(MarSystem* m, std::string cname, double start, double end, doub
{
    set(m,cname,start,end,rate);
}
EvRampCtrl::EvRampCtrl(EvRampCtrl& e) : EvEvent("EvRampCtrl","rc")
{
    set(e.target_,e.cname_,e.start_,e.end_,e.rate_);
}

EvRampCtrl::~EvRampCtrl() { }

EvRampCtrl*
EvRampCtrl::clone() { return new EvRampCtrl(*this); }

void
EvRampCtrl::set(MarSystem* ms, string cname, double start, double end, double rate)
{
    target_=ms; cname_=cname;
    start_=start; end_=end; rate_=rate;
    value_=start_; repeat_flag_=true;
}

void
EvRampCtrl::dispatch()
{
    if (target_ !=NULL) {
      cout << "target_->updctrl(" << cname_ << ", " << value_ << ")\n";
      if(value_<end_) repeat_flag_=false;
      else {
        target_->updctrl(cname_,value_);
        value_ = value_ - rate_;
      }
    }
}
```

Figure 9.9: EvRampCtrl event C++ source file example.

The `fade1` method shows our EvRampCtrl event in action. We set the repeat rate of the event to 0.2 seconds.

```
void fade1() {
  MarSystemManager mng;

  MarSystem* series = mng.create("Series", "series");
  series->addMarSystem(mng.create("SineSource", "src"));
  MarSystem* gain = mng.create("Gain", "g");
  series->addMarSystem(gain);
  series->addMarSystem(mng.create("AudioSink", "snk"));
  series->updctrl("AudioSink/snk/mrs_bool/initAudio", true);
  series->updctrl("SineSource/src/mrs_real/frequency",440.0);
  series->updctrl("Gain/g/mrs_real/gain",1.0);

  EvRampCtrl* ev = new EvRampCtrl(gain,"mrs_real/gain",1.0,0.0,0.05);
  ev->set_repeat(Repeat("0.2s"));

  series->updctrl(TmTime("TmSampleCount/Virtual","2s"),ev);

  while(true) {
    series->tick();
  }
}
```

Figure 9.10: Program using the EvRampCtrl example event.

9.6.3.1 Expression Events

For a large number of events there is commonality. The Expression syntax was developed to allow the creation of events without having to code new custom event classes. In order to accomplish this there is a built in compiler for the syntax that is invoked when supplying the expression as a string to the '`Ex`' class.

```
#include <stdio.h>
#include "MarSystemManager.h"
#include "EvExpr.h"

using namespace std;
using namespace Marsyas;

void sched1() {
  MarSystemManager mng;

  MarSystem* fanin = mng.create("Fanin", "fanin");
  fanin->addMarSystem(mng.create("SineSource", "src1"));
  fanin->addMarSystem(mng.create("SineSource", "src2"));
  fanin->updctrl("SineSource/src1/mrs_real/frequency",3000.0);
  fanin->updctrl("SineSource/src2/mrs_real/frequency",1000.0);

  MarSystem* series = mng.create("Series", "series");
  series->addMarSystem(fanin);

  series->addMarSystem(mng.create("AudioSink", "dest"));
  series->updctrl("AudioSink/dest/mrs_bool/initAudio", true);

#if 0
  // using aliases makes this a little more readable, see the next bit
  EvExpr* e = new EvExpr(series,
    Ex("Fanin/fanin/SineSource/src1/mrs_real/frequency << 120. + 3000. * R.rand(), \
        Fanin/fanin/SineSource/src2/mrs_real/frequency << 120. + 800. * R.rand(),"
       "'src1='+Fanin/fanin/SineSource/src1/mrs_real/frequency+ \
        ' src2='+Fanin/fanin/SineSource/src2/mrs_real/frequency+'\n'>>Stream.op"),
    Rp("true"));

#else
  EvExpr* e = new EvExpr(series,
       // First line to Ex will be the init expression, run once, when event is poste
    Ex("Fanin/fanin/SineSource/src1/mrs_real/frequency >> @freq1, \
        Fanin/fanin/SineSource/src2/mrs_real/frequency >> @freq2 ",
       // Second line to Ex is the expression, repeated each time the event is posted
       "freq1 << 120. + 3000. * R.rand(),\
        freq2 << 120. + 800. * R.rand(),\
        'src1=' + freq1 + ' src2=' + freq2 + '\n' >> Stream.op"),
    Rp("true"));
#endif

  // set event to repeat every 1/4 second
  e->set_repeat(Repeat("0.25s"));
  // post the event
  series->updctrl(TmTime("TmSampleCount/Virtual","0s"), e);

  while(true) series->tick();

  delete series;
}
```

9.6.4 Marsyas Expression Syntax

Expressing instantaneous change in Marsyas.

Still in development, but what software project isn't..

The motivation for an expression syntax within the scheduler is purely one of convenience. While the Marsyas scheduler is highly configurable, allowing the programmer to add their own classes for their own event requirements, eyes tend to glaze over when C++ programming is mentioned. This expression syntax allows a programmer to define an expression that can be fed to an event and evaluated once or any number of times within the scheduler. While many functions are available, more can be added with a little ... C++ programming. Read on fearless marsyasian.

9.6.4.1 Type System

- Basic Types
  - Bool - can be true and false or shorthand $t and $f.
  - Natural - a number without a decimal point.
  - Real - floating point number (equivalent of a C double) can be written with leading or trailing decimal point - .123, 123., and 123.456.
  - String - a string of characters delimeted by single quotes - 'hello' - which may contain newline \n or tab \t characters - '1\t2\t3\n'. Strings are also sequence types and can be modified using the sequence iterators.
- Type coercion

  There are a few instances where type coercion may take place. Binary operators such as + require two values of the same type. Natural numbers are therefore promoted to reals in such a situation (1 + 1.0 := ((real)1) + 1.0). Real numbers are not converted to naturals in any case. For string concatenation using the + operator, any type concatenated to a string is converted to a string type (1+'2' := '12').

  Type coercion also affects function parameters. For instance, the function Real.cos(real) when used as Real.cos(1) would result in a type error as 1 is interpreted to be a natural value. Instead the function call is converted to Real.cos((real)1).

  Note that there is a performance penalty for coercion as slight as it may be.
- Type terms used in this document
  - bool
  - natural
  - real
  - string
  - num_t := natural | real
  - basic_t := natural | real | string | bool

9.6.4.2 Operators

- String
  - Binary (+)
    - Concatenation (+).

        - string + basic_t := string ~ 'go ' + 2 := 'go 2'
        - basic_t + string := string ~ 2 + ' much' := '2 much'
        - string + string := string ~ 'too' + ' much' := 'too much'
- Numbers
    - Unary (-)
        - Negation (-) - overloaded for Real, Natural.
            - - natural := natural ~ - 2 := -4
            - - real := real ~ - 2.1 := -2.1
    - Binary (+ - * / %)
      Operations involving both a Real and Natural type will result in a Real type.
        - Addition (+) - overloaded for Real, Natural, and String types.
            - natural + natural := natural ~ 2 + 2 := 4
            - num_t + real := real ~ 2 + 2.3 := 4.3
            - real + num_t := real ~ 2.3 + 2 := 4.3
            - string + basic_t := string ~ 'go ' + 2 := 'go 2'
            - basic_t + string := string ~ 2 + ' much' := '2 much'
        - Subtraction (-) - overloaded for Real and Natural types.
            - natural - natural := natural ~ 2 - 2 := 0
            - num_t - real := real ~ 2 - 2.3 := 0.3
            - real - num_t := real ~ 2.3 - 2 := 0.3
        - Multiplication (*) - overloaded for Real and Natural types.
            - natural * natural := natural ~ 2 * 2 := 4
            - num_t * real := real ~ 2 * 2.3 := 4.6
            - real * num_t := real ~ 2.3 * 2 := 4.6
        - Division (/) - overloaded for Real and Natural types.
          The division operator is also used to delimit control name paths so this a/b or will be interpreted as a path and a /b will be interpreted as variable a followed by path /b. Division involving variables must be written with the division separated by spaces as in: a / b or even a/ b.
            - natural / natural := natural ~ 2 / 2 := 1
            - num_t / real := real ~ 2 / 2.3 := 0.8695
            - real / num_t := real ~ 2.3 / 2 := 1.15
        - Remainder (%) - overloaded for Real and Natural types.
            - natural % natural := natural ~ 3 % 2 := 1
            - num_t % real := real ~ 4 % 1.5 := 1.0
            - real % num_t := real ~ 4.5 % 2 := 0.5
- Boolean
    - Negation (!)
        - ! bool := bool ~ ! $t := $f

- Relational (& |)
  - And (&) - comparison of bool types.
    - bool & bool := bool ~ $t & $f := $t
  - Or (|) - comparison of bool types.
    - bool | bool := bool ~ $t | $f := $t
- Comparison (= != > >= < <=)
  Comparisons may only be made between the same types.
  - Equal (=) - comparison of types.
    - basic_t = basic_t := bool ~ 3 = 2.1 := $f
  - Not Equal (!=) - comparison of types.
    - basic_t != basic_t := bool ~ 3 != 2.1 := $t
  - Greater Than (>) - comparison of types.
    - basic_t > basic_t := bool ~ 3 > 2.1 := $t
  - Greater Equal (>=) - comparison of types.
    - basic_t >= basic_t := bool ~ 3 >= 2.1 := $t
  - Less Than (<) - comparison of types.
    - basic_t < basic_t := bool ~ 3 < 2.1 := $f
  - Less Equal (<=) - comparison of types.
    - basic_t <= basic_t := bool ~ 3 <= 2.1 := $f

9.6.4.3 Variables

Variable names must start with a letter but may be followed by letters or numbers. hello and hi8 are valid names wheras 2much is not. The type of a variable is defined through assignment. 12 >> x defines variable x of type natural. Once a variable type is defined it cannot be changed, 12 >> x, 'hello' >> x results in a type mismatch error. There is no notion of scope except that all variables are in the same scope so that variables defined within a conditional statement will exist outside the conditional and after it has been evaluated.

Control Names (/absolute/path or relative/path)

Control names are written using the path notation, /Gain/gain/mrs_real/gain.

Aliases (@name)

Control names can be rather long. Aliases can be used to shorten expressions containing control names. Making an alias is done through assignment to a name preceded by the symbol: /Gain/gain/mrs_real/gain >> @gain. After assignment, the alias can be used as any other variable name: gain << 0.5.

9.6.4.4 Assignment (<< >>)

Assignment is performed using the assignment operators. Left assignment x << 5 and right assignment 5 >> x. These operators are equivalent. Assignment of a value to a variable defines the variable type. Once a variable is 'typed' it cannot be assigned another type. Note there is no scope, all variables are at the 'top level.'

Assignment Sugar (<<+ +>> etc...)

Note: left and right assignment do not necessarily yield the same results.

- Add Assign (<<+, +>>) - left assignment is x <<+ 5 is x << x + 5 whereas right assignment is 5 +>> x is 5 + x >> x.
- Subtract Assign (<<-, ->>) - left assignment is x <<- 5 is x << x - 5 whereas right assignment is 5 ->> x is 5 - x >> x. Left and right assignment are not the same here.
- Multiply Assign (<<*, *>>) - left assignment is x <<* 5 is x << x * 5 whereas right assignment is 5 *>> x is 5 * x >> x.
- Divide Assign (<</, />>) - left assignment is x <</ 5 is x << x / 5 whereas right assignment is 5 />> x is 5 / x >> x. Left and right assignment are not the same here.
- Remainder Assign (<<%, %>>) - left assignment is x <<% 5 is x << x % 5 whereas right assignment is 5 %>> x is 5 % x >> x. Left and right assignment are not the same here.

9.6.4.5 Links (-> <-)

Controls may be linked using the link operators. Left and right operators perform the same actions. There is no bidirectional link since this would require defining a link order. This order is left to the user who will have to explicity define the order by making two links.

9.6.4.6 Conditional Statements ({? cond_expr : exprs1 : exprs2 })

The conditional statement is used for making decisions. The cond term is the decision and must result in a boolean. If cond is true then the then_expr is executed otherwise the else_expr is executed. Each of these expressions may be a list of expressions, even the cond_expr so long as cond_expr results in a bool type and both expr1 and expr2 result in the same type. Example: {? 5<3 : 1 : 2}.

9.6.4.7 Multiple Expressions (exprs := expr1 , expr2 , expr3)

Multiple expressions may be executed by separating them with commas as in t << 5, t << t - 2.

9.6.4.8 Properties (ie Real.pi or 'hello'.2 etc.)

Properties are simply a way of associating methods or values with types. As an example, the String module supports a number of methods that can accessed using the property notation. The expression String.len('hello') results in the natural value of 5. This expression could also be written as 'hello'.len() since the value 'hello' is known to be a string. This is just sugar for String.len('hello'). Here the first parameter to String.len is filled with the value. This can be used for variables of the right type as in s << 'hello', s.len() is similar to the previous except that variable s is defined.

All basic types or things that evaluate to basic types can respond to property calls. For example the conditional operator {?cond:expr1:expr2} evaluates to a single type so can take on a property as in {?true:'123':'12'}.len. To take this further a function call resulting in a basic type can take on properties as well and so on: '123'.len.max(4).

9.6.4.9 Sequences (Lists)

Sequence types are those data types that have a sequence of elements. Falling into this category are lists and strings not natural or real numbers.

Lists are denoted using the square brackets as in [1, 2, 3]. Lists can contain other lists as in [[1,2], [3,4]]. However, all elements of a list must be of the same type. This [1, '2'] and this [1, [2, 3]] are not valid lists.

Iterators

There are four different iterators for working with sequence types: map, iter, for, rfor. There are subtle differences.

- map {map x in xs: <expr>}
  The map iterator works in a similar fashion to the functional map. Map iterates from left to right across each element and replaces each element with a new one of the same or different type. Map returns a new list without modifying the original. An example: xs<<[1,2,3], b<<{map x in xs: x+1}.
- iter {iter x in xs: <expr>}

  The iter iterator works like map except that it modifies the original list in place. Iter returns unit. An example: xs<<[1,2,3], {iter x in xs: x+1}.
- for {for x in xs: <expr>}

  The for iterator iterates across the list but does not modify the original list and does not create a new list. for returns unit. An example: xs<<[1,2,3], sum<<0, {for x in xs: sum <<+ x}.
- rfor {rfor x in xs: <expr>}

  The rfor iterator works in the same way as for but from right to left. An example: xs<<[1,2,3], b<<{rfor x in xs: Stream.opn << x}.

Element Access

Elements of sequences may be accessed using the standard array notation as in a<<[1,2,3], one<<a[0], three<<a[2]. Note that the first element is position 0. Don't let me catch you using an index that's too big.

Slices

Slices are portions of sequences. A range takes two positions in the list - start position (included) and end position (not included) separated by a colon [start:end]. If the start position is left out then the first element is assumed. If the last element is left out then the end is assumed. The start and end positions may be adjusted to avoid errors, ie negative start becomes the beginning of the list.

We can take the head and tail of a list using slices: a<<[1,2,3], hd<<a[0], tl<<a[1:].

Concatenation, Joining

Sequences may be joined using the concatentation operator otherwise known as +. [1,2,3]+[4] joins the two lists into [1,2,3,4].

9.6.4.10 Function Libraries

In most cases library names can be shortened to reduce typing. For example the Real module can be reduced to R as in R.cos(1.57).

- Real (R)
  - **mrs_real** *Real*.e
    e = ~2.718281828
  - **mrs_real** *Real*.pi
    pi = ~3.141592654
  - **mrs_real** *Real*.pi2
    pi/2 = ~1.570796327
  - **mrs_real** *Real*.pi4
    pi/4 = ~0.785398163
  - **mrs_real** *Real*.rpd
    Radians per Degree = ~0.017453292
  - **mrs_real** *Real*.dpr
    Degrees per Radian = ~57.29577951
  - **mrs_real** *Real*.sqrt2
    Square Root of 2 = ~1.414213562
  - **mrs_real** *Real*.cos(**mrs_real**)
    Trigonometric cosine in radians.
  - **mrs_real** *Real*.acos(**mrs_real**)
    Trigonometric arc cosine in radians.
  - **mrs_real** *Real*.cosh(**mrs_real**)
    Hyperbolic cosine in radians.
  - **mrs_real** *Real*.sin(**mrs_real**)
    Trigonometric sine in radians.
  - **mrs_real** *Real*.asin(**mrs_real**)
    Trigonometric arc sine in radians.
  - **mrs_real** *Real*.sinh(**mrs_real**)
    Hyperbolic cosine in radians.
  - **mrs_real** *Real*.tan(**mrs_real**)
    Trigonometric tangent in radians.
  - **mrs_real** *Real*.atan(**mrs_real**)
    Trigonometric arc tangent in radians.
  - **mrs_real** *Real*.ln(**mrs_real**), **mrs_real** *Real*.log(**mrs_real**)
    Natural logarithm, base e.
  - **mrs_real** *Real*.log2(**mrs_real**)
    Logarithm base 2.
  - **mrs_real** *Real*.log10(**mrs_real**)
    Logarithm base 10.
  - **mrs_real** *Real*.sqrt(**mrs_real**)
    Square root.

- Natural (N)
  - **mrs_natural** *Natural*.randmax
    Constant for the maximum random number.
  - **mrs_natural** *Natural*.abs(**mrs_natural**)
    Absolute value.
  - **mrs_natural** *Natural*.rand()
    Generate a random number.
  - **mrs_natural** *Natural*.srand(**mrs_natural**)
    Generate a random number using the given seed.
  - **mrs_natural** *Natural*.min(**mrs_natural**, **mrs_natural**)
    Return the minimum of two numbers.
  - **mrs_natural** *Natural*.max(**mrs_natural**, **mrs_natural**)
    Return the maximum of two numbers.
- String (S)
  - **mrs_natural** *String*.len(**mrs_string**)
    String length.
- Stream
  - **outstream** op
    Stream.op and Stream.opn support assignment. Data can be written to stdout by assigning to this object as in 3.1415 >> Stream.op or 'hello' >> Stream.op.
  - **outstream** *Stream*.op(**mrs_real**)
    write a mrs_real number to stdout.
  - **outstream** *Stream*.op(**mrs_natural**)
    write a mrs_natural number to stdout.
  - **outstream** *Stream*.op(**mrs_bool**)
    write a mrs_bool number to stdout.
  - **outstream** *Stream*.op(**mrs_string**)
    write a mrs_string number to stdout.
  - **outstream** *Stream*.opn(**mrs_real**)
    write a mrs_real number to stdout followed by a newline.
  - **outstream** *Stream*.opn(**mrs_natural**)
    write a mrs_natural number to stdout followed by a newline.
  - **outstream** *Stream*.opn(**mrs_bool**)
    write a mrs_bool number to stdout followed by a newline.
  - **outstream** *Stream*.opn(**mrs_string**)
    write a mrs_string number to stdout followed by a newline.
- Timer (Tmr)
  - **mrs_timer** *Timer*.cur
    the current timer that the event expression is posted on.
  - **mrs_string** *Timer*.name(**mrs_timer**)
    the name of the timer
  - **mrs_string** *Timer*.type(**mrs_timer**)
    the type of the timer

- **mrs_string** *Timer*.prefix(**mrs_timer**)
  the prefix of the timer which is type/name
- **mrs_natural** *Timer*.time(**mrs_timer**)
  returns the current time of the timer as a count. This is not in the form of the specific representation for the particular timer.
- **mrs_bool** *Timer*.upd(**mrs_timer**, **mrs_string**, **mrs_bool**)
  updates a timer control value where the second parameter is the string path and the third parameter is the value
- **mrs_bool** *Timer*.upd(**mrs_timer**, **mrs_string**, **mrs_real**)
  updates a timer control value where the second parameter is the string path and the third parameter is the value
- **mrs_bool** *Timer*.upd(**mrs_timer**, **mrs_string**, **mrs_natural**)
  updates a timer control value where the second parameter is the string path and the third parameter is the value
- **mrs_bool** *Timer*.upd(**mrs_timer**, **mrs_string**, **mrs_string**)
  updates a timer control value where the second parameter is the string path and the third parameter is the value
- **mrs_natural** *Timer*.ival(**mrs_timer**, **mrs_string**)
  returns a natural value corresponding to the string representation of a time value in the specific format for the timer. ie "1s" may represent 1 second for a timer.

9.6.4.11 Using

Two examples show simple usage. These examples are in src/scheduler.cpp.

There are a few ways to specify an expression to be parsed.

- **new** EvExpr(marsym, Ex("*x* << *3*","*'hello'* + *x* >> *Stream.opn*"),Rp("*false*"));
  specifies a primary expression within the Ex constructor that is to be executed on every repetition of the event. Values to be used in the expression may be initialised in the first string, then used in the second string. The first string is evaluated as soon as the expression event is posted on some timer. The second string is executed on each repetition. The init string is optional within the Ex constructor (in fact the Rp exprsession may also have an init expression though it can be combined with the ExInit expression). However the expression inside the Rp Constructor specifies a repeat expression that determines if the event is to be repeated. This expression must have a type boolean.
- **new** EvExpr(marsym, ExFile("*filename.expr*"));
  the expressions for Ex and Rp can be read in from a file. This eliminates the need to recompile each time an expression needs to be adjusted. The file format is text where the lines expected by Ex and Rp are separated by titles:
  - #ExInit: - specify an init variables expression.
  - #ExExpr: - specify the expression to be repeated.
  - #RpInit: - specify an init variables expression for the repeated expression. The repeat expression may share variables with the primary Ex expression so anything specified here can be specified in #ExInit:

- #RpExpr: - specify a boolean expression that determines if the expression event is to be repeated after each repetition. This expression is evaluated after the primary expression.
- #RpRate: - (Not implemented yet, pending) specify an expression that evalutes to a mrs_string that determines how far into the future the event is to be posted after it is evaluated. The result must have meaning in the current timer timebase.

9.6.4.12 Extending

There are two ways to extend the libraries with your own functions. The hard way is to hand code the function as a class in C++ then modify a corresponding loadlib function in ExNode.cpp. The other way is to use the code generation script in tools/scheduler.

Useful conversion functions

There are a few built in functions for converting to and from the std::string type. These functions can be used in your new functions.

- ltos(v) - converts v from a mrs_natural value to a std::string
- dtos(v) - converts v from a mrs_real value to a std::string
- btos(v) - converts v from a mrs_bool value to a std::string
- stol(v) - converts v from a std::string value to a mrs_natural

Adding new libraries or library functions (C++ way)

Library functions are added by creating a new ExFun class. The first step is creating a new ExFun class for the new function. The second step is adding it to the loadlib functions in ExNode.cpp.

Creating a new ExNode

The ExFun class derives from the ExNode which is an expression tree node. The example here is a function for extracting a substring from a given string.

Functions taken a set number of parameters. The function class is supplied information on the number of parameters and their types through the setSignature function when the function class is instantiated in the symbol table. For now we assume this set of parameters (mrs_string,mrs_natural,mrs_natural) for the original string, the start index, and the end index.

Creating an Exfun class requires the definition of the constructor, a calc() method that returns the result of the function call, and a copy method that returns a copy of the function without its parameters. The example shows the

```
class ExFun_StrSub : public ExFun { public:
    ExFun_StrSub() : ExFun("mrs_string","String.sub(mrs_string,mrs_natural,mrs_natural
    virtual ExVal calc() {
        std::string str = params[0]->eval().toString();
        int s = params[1]->eval().toNatural();
        int e = params[2]->eval().toNatural();
        int l = str.length();
        if (s<0) { s=0; } else if (s>l) { s=l; }
        if (e<s) { e=s; } else if (e>l-s) { e=l-s; }
        return str.substr(s,e);
    }
    ExFun* copy() { return new ExFun_StrSub(); }
};
```

Figure 9.12: String Substring ExFun class.

Note that the constructor takes two parameters where t is the type that the function evaluates to and n is the signature of the function. These parameters are simply passed on to the ExFun parent constructor along with a third boolean parameter for the pureness of the function. Pureness is a flag that determines whether the function is free of side-effects or not. If the parameters to the function can be determined to be constant then a pure function can be evaluated at parse time to a constant value.

The calc() method uses the three parameters from the params[] array. These parameters are set at parse time and placed in the params[] array. Each param[] is an expression so they are of type ExNode*. Therefore, you need to evaluate each expression prior to using it. To evaluate, call the eval() method of the ExNode not the calc() method. eval() will make sure that each expression in a list of expressions will be evaluated whereas calc() only calculates the current node.

Adding the function to the library

The function can be added to the library by adding a line to the loadlib_String method in ExNode.cpp. The addReserved call made to the symbol table adds a reserved word. There is some flexibility in how the name appears in the symbol table which in turn defines how it may be used in an expression. The path to a function is separated by the '.' symbol. Multiple names for a segment of the path can be defined by separating them with the '|' symbol where the first among several is the 'true' name. For example String|Str|S.sub defines three different leading names where the true name is String.sub but S.sub will get the same thing. The same is possible for parameter tuples but not the individual parameters. For example Real|R.cos(mrs_real)|(mrs_natural) allows the call Real.cos(0.5) as well as Real.cos(1) as 1 is a natural and not a real. This type information is used to type check function calls in the parser.

```
st->addReserved("String|S.sub(mrs_string,mrs_natural,mrs_natural)",
                new ExFun_StrSub("mrs_string",
                                 "String.sub(mrs_string,mrs_natural,mrs_natural)"));
```

Figure 9.13: Adding a function to the library.

The second parameter to the addReserved call is a new instantiation of the function object. Here the return type is the first parameter to the constructor and the 'correct' or 'true' full function call is the second parameter. This information is used for type checking parameters. While type errors are not possible if the first parameter to the addReserved call is correct, the type information in the second parameter to the constructor is actually used for type coercion - promoting naturals to reals, etc.

Adding new libraries or library functions (Code Gen way)

In the tools/scheduler directory is a python script for generating library functions from a simplified source code. The easiest way to explain the process is through an example:

```
 1: lib Foo|F.Bar|B
 2:
 3: pure mrs_string fun|alt(mrs_real a, mrs_natural b)
 4:     mrs_string s="hello";
 5:     mrs_bool x;
 6: {
 7:     mrs_natural z=a+b;
 8:     if (x) { s=s+" "+ltos(z); }
 9:     x = z < 0;
10:     return s;
11: }
```

Figure 9.14: Adding a function to the library.

Though not a useful function it does demonstrate the full extent of the code generation syntax.

Line 1. library definition starts with keyword 'lib' the names following denote a path to the library. The true path is Foo.Bar, all functions defined after this statement until a new lib definition will be in this library. This means that the function fun is called by 'Foo.Bar.fun'. Alternate names or aliases for portions of the path can be defined using the `|` symbol. In the above example F is an alias for Foo so the path to fun could also be written as 'Foo.B.fun' or 'F.B.fun' etc.

Line 3. the function definition may start with 'pure' where pure implies that if the parameters to the function are constants then the function can be evaluated at parse time to a constant, ie no side-effects. If pure isn't specified then the function is not pure. the return type must be a type supported by the ExVal class (names starting with 'mrs_'). The function name can also have aliases divided by the `|` symbol where the first name is the true name. Parameters must be defined using the 'mrs_' names.

Line 4. Normally functions do not have state but as a bonus variables whose values persist may defined after the parameters definition and prior to the opening function body brace. These types can be the 'mrs_' types or valid C++ types.

Line 6. The function body begins with a opening brace `{`.

Line 7-10. The function body contains valid C++ code and will likely use the parameter values defined on line 3.

Line 11. The function body ends with a closing brace `}`.

9.6.4.13 Marsyas Expression Examples

```
#include <stdio.h>
#include "MarSystemManager.h"
#include "EvExpr.h"

using namespace std;
using namespace Marsyas;

void sched1()
{
  MarSystemManager mng;

  MarSystem* fanin = mng.create("Fanin", "fanin");
  fanin->addMarSystem(mng.create("SineSource", "src1"));
  fanin->addMarSystem(mng.create("SineSource", "src2"));
  fanin->updctrl("SineSource/src1/mrs_real/frequency",3000.0);
  fanin->updctrl("SineSource/src2/mrs_real/frequency",1000.0);

  MarSystem* series = mng.create("Series", "series");
  series->addMarSystem(fanin);

  series->addMarSystem(mng.create("AudioSink", "dest"));
  series->updctrl("AudioSink/dest/mrs_bool/initAudio", true);


#if 0
  // using aliases makes this a little more readable, see the next bit
  EvExpr* e = new EvExpr(series,
    Ex("Fanin/fanin/SineSource/src1/mrs_real/frequency << 120. + 3000. * R.rand(), \
        Fanin/fanin/SineSource/src2/mrs_real/frequency << 120. + 800. * R.rand(),"
       "'src1='+Fanin/fanin/SineSource/src1/mrs_real/frequency+ \
        ' src2='+Fanin/fanin/SineSource/src2/mrs_real/frequency+'\n'>>Stream.op"),
    Rp("true"));

#else
  EvExpr* e = new EvExpr(series,
        // First line to Ex will be the init expression, run once, when event is poste
    Ex("Fanin/fanin/SineSource/src1/mrs_real/frequency >> @freq1, \
        Fanin/fanin/SineSource/src2/mrs_real/frequency >> @freq2 ",
       // Second line to Ex is the expression, repeated each time the event is posted
       "freq1 << 120. + 3000. * R.rand(),\
        freq2 << 120. + 800. * R.rand(),\
        'src1=' + freq1 + ' src2=' + freq2 + '\n' >> Stream.op"),
    Rp("true"));

#endif

  e->set_repeat(Repeat("0.25s"));

  series->updctrl(TmTime("TmSampleCount/Virtual","0s"), e);

  for (int i=1;i<100;i++) series->tick();
```

```
void sched2()
{
  MarSystemManager mng;

  MarSystem* fanin = mng.create("Fanin", "fanin");
  fanin->addMarSystem(mng.create("SineSource", "src1"));
  fanin->addMarSystem(mng.create("SineSource", "src2"));
  fanin->updctrl("SineSource/src1/mrs_real/frequency",400.0);  // A

  fanin->updctrl("SineSource/src2/mrs_real/frequency",554.37); // C#
  fanin->updctrl("SineSource/src3/mrs_real/frequency",659.26); // E

  MarSystem* series = mng.create("Series", "series");
  series->addMarSystem(fanin);

  series->addMarSystem(mng.create("AudioSink", "dest"));
  series->updctrl("AudioSink/dest/mrs_bool/initAudio", true);

  EvExpr* e =
    new EvExpr(series,
      Ex("tmr<<Timer.cur","tmr.prefix + ' ' + tmr.time + '\n' >> Stream.op"),
      Rp("true"));

  e->set_repeat(Repeat("0.25s"));

  series->updctrl(TmTime("TmSampleCount/Virtual","0s"), e);

  for (int i=1;i<100;i++) series->tick();

  delete series;
}
```

Figure 9.16: Printing the time of the current timer

10 Programming 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.

10.1 Compiling and using a new MarSystem

Writing a new MarSystem is relatively straightforward if you begin with a working example and modify it for your needs. Fortunately, there is a python script which does precisely this.

10.1.1 Writing your own MarSystems

To create a "blank" MarSystem to begin programming, use

```
path/to/marsyas/scripts/createMarSystem.py MyMar
```

where `MyMar` is the name of your new MarSystem. This script will create '`MyMar.h`' and '`foo.cpp`' in your **current** directory.

If you want to create these new MarSystems in the '`src/marsyas/`' directory, go to that directory and call the script. Relative paths are fine, for example `../../scripts/createMarSystem.py MyMar` (on *nix).

10.1.2 Using your MarSystem

Suppose that you have created a MarSystem called `MyMar`, which implements a filter. To use this MarSystem in a network (see Chapter 9 [Writing applications], page 76), simply register the MarSystem with the manager:

```
MarSystemManager mng;
// add MyMar to MarSystemManager
MarSystem* myMar = new MyMar("hello");
mng.registerPrototype("MyMar", myMar);

// create a network normally
playbacknet = mng.create("Series", "playbacknet");
playbacknet->addMarSystem(mng.create("SoundFileSource", "src"));
playbacknet->addMarSystem(mng.create("MyMar", "mymar"));
playbacknet->addMarSystem(mng.create("Gain", "gain"));
playbacknet->addMarSystem(mng.create("AudioSink", "dest"));
playbacknet->updctrl("AudioSink/dest/mrs_bool/initAudio", true);
```

10.2 Anatomy of a MarSystem

10.2.1 Methods of the object

A MarSystem is an object which contains these methods. In this example, we use a fake MarSystem called `MyName`.

- Constructors / Destructor:

```
MyName::MyName(string name):MarSystem("MyName", name)
MyName::MyName(const MyName& a) : MarSystem(a)
MyName::~MyName()
MarSystem* MyName::clone() const
```

- Handling controls:

```
void MyName::addControls()
void MyName::myUpdate(MarControlPtr sender)
```

- Actual processing method:

```
void MyName::myProcess(realvec& in, realvec& out)
```

Most of the changes that you make to the basic template will be to the '`Handling controls`' methods and the `myProcess` method. For more information, see Section 10.2.3 [Handling controls], page 131 and Section 10.2.4 [myProcess()], page 132.

10.2.2 Constructors / destructor

The first function is the standard C++ constructor; the second function is the copy constructor. The destructor is straightforward.

```
MyName::MyName(string name):MarSystem("MyName", name)
MyName::MyName(const MyName& a) : MarSystem(a)
MyName::~MyName()

MarSystem* MyName::clone() const
```

`clone()` is used to create a new MarSystem; Marsyas stores an instance of every MarSystem at run-time, and future MarSystems are simply `clone()`'d from the initial instance.

Copy constructor

All member pointers to controls **must** be explicitly reassigned in the copy constructor. Otherwise these member points would be invalid, which results in trying to de-allocate them twice! The function should look like this:

```
MyMar::MyMar(const MyMar& a) : MarSystem(a)
{
    ctrl_gain_ = getctrl("mrs_real/gain");
    ctrl_other_ = getctrl("mrs_natural/other");
    ctrl_dothis_ = getctrl("mrs_bool/dothis");
    ...
}
```

10.2.3 Handling controls

`addControls()` defines which controls a MarSystem uses:

```
addctrl("mrs_real/frequency", 1000);
```

```
//setctrlState("mrs_real/frequency", true);
```

The `addctrl()` sets up a control for the MarSystem; this control may be changed by other C++ code by doing

```
MarNetwork->updctrl("MyName/myInstanceName/mrs_real/frequency",
500 );
```

This will change the value of the control and call `MyName::myUpdate()`.

If we called `setctrl` instead of `updctrl`,

```
MarNetwork->setctrl("MyName/myInstanceName/mrs_real/frequency",
500 );
```

Then `myUpdate()` will not be called. If we had set `setctrlState` to *true* (ie uncommented the line in the initial example), then setting this control would automatically call `MyName::myUpdate()`.

10.2.4 `myProcess()`

`myProcess()` is called every time the MarSystem receives a `tick()` (ie all the time the program is running).

Resource-intensive operations (such as changing the buffer size, computing trigonomic functions, etc) that only depend on the controls (not the actual dataflow input) should be performed inside `myUpdate()`. For more information, see Section 10.2.5 [myUpdate() vs. myProcess()], page 132

Most `myProcess()` functions will look like this:

```
void
MyMar::myProcess(realvec& in, realvec& out)
{
// pre-loop initialization
...

// loop over buffer
for (o=0; o < inObservations_; o++)
for (t = 0; t < inSamples_; t++)
// calculate next value
...
out(o,t) = ...;

// post-loop actions
...
}
```

10.2.5 `myUpdate()` vs. `myProcess()`

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);
```

```
// main loop
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()`.

10.2.6 More details about MarSystems

These files have useful comments:

```
marsyas/MarSystemTemplateBasic .h .cpp
marsyas/MarSystemTemplateMedium .h .cpp
marsyas/MarSystemTemplateAdvanced .h .cpp
marsyas/Gain .h .cpp
```

Missing Docs

B

C

D

O

The Index