Marsyas Developers Manual

For version 0.2

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

Graham Percival and George Tzanetakis

Table of Contents

1 Contributing to Marsyas

This chapter explains how to integrate your code in Marsyas so that others may use it.

1.1 Contributing documentation

The documentation for Marsyas is still a work in progress; we can use all the help we can get. Don't say "oh, I don't know enough" or "I'm not good at writing English." The question is not "could anybody create something better than my suggestion?" – the question is "is this better than nothing?" Remember the most important thing about documentation:

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

1.1.1 Manual

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

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

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

1.1.2 Examples

Small, easy-to-understand examples are also great. If you have some source code that illustrates something, we can add it to '`examples/`'. We use these examples to generate the Section "Example programs" in *Marsyas User Manual*. You don't have to write any English at all!

1.2 Contributing source documentation

In your `.h` file, just after the `namespace Marsyas \n {` line, please include a short documentation snippet:

```
namespace Marsyas
{
/**
    \ingroup Processing
    \brief Multiply input realvec with gain

Simple MarSystem example.  Just multiply the values of the input
realvec with gain and put them in the output vector.  This object can be
used as a prototype template for building more complicated MarSystems.
```

```
    Controls:
    - \b mrs_real/gain [w] : adjust the gain multiplier.
*/

class Gain: public MarSystem
{
```

Warning: the old convention included a `\class Gain`; this was a mistake in our understanding of Doxygen. Please **do not** include an explicit class name; just make sure that the doc snippet occurs above the class.

The `[w]` indicates that the control should be written and not (usefully) read. Valid options are `[r]`, `[w]`, and `[rw]`.

The `\ingroup GROUP` will generally be `Processing`, `Analysis`, or `Synthesis`. For details about these categories, see Section "MarSystem reference" in *Marsyas User Manual*. For a complete list of all available groups, see the file '`marsyas/groups.doxy`'.

There is one special group: `Basic` Processing. This is a subset of the `Processing` group. No MarSystem should only be in the `Basic` group; it should be placed in both groups, with

```
\ingroup Processing Basic
```

Code which is not a MarSystem should be placed in the `Notmar` group.

A complete list of groups can be found in the '`marsyas/groups.doxy`' file. The main groups are `Composites`, `Basic`, `IO`, `Processing`, `Analysis`, and `Synthesis`.

1.3 Code style

- We use the Allman code style (also known as ANSI or BSD code style) with tabs as indentation. Code style is of course somewhat personal and we can almost guarantee that everybody will hate *some* aspect of the style, but having a uniform style makes it much easier to understand other people's code and fix bugs.

  If possible, configure your editor/IDE to use this code style. Otherwise, you can use the `astyle` tool (Artistic Style) (with the `ansi` style) for automated clean up of code style issues (braces, indentation, whitespace). For ease of use, we have a file '`misc/astylerc`' with predefined options. To use it, call

  ```
  astyle --options=/path/to/marsyas_topdir/misc/astylerc filename.cpp
  ```

  '`filename.cpp`' can include wildcards, like '`*.h`' and '`*.cpp`'.

  You do not need to format your source code according to this style, but don't complain if someone else modifies/cleans up your file accordingly.

  Unfortunately, due to historical reasons the code style is not consistant within the Marsyas source code itself.

- Files should use unix line endings.

- We strongly encourage the use of underscores_ after member variables ("member variables" are variables which can be used throughout the class). In other words, in your '`.h`' files use:

  ```
  class MyMar: public MarSystem
  {
  ```

```
private:
mrs_real varname_;  // don't use a plain "varname"
...
}
```

- By convention, a program (not a MarSystem) should return `exit(0)` upon successful completion. If any problems arise, the program should return a number greater than 0.

1.4 Playing in the mudbox

The easiest way to add code to Marsyas is to add a test to '`apps/mudbox/mudbox.cpp`'. This file is a huge mess of short examples, many out of date and no longer working, but it seems popular.

To add your own test, follow the general pattern of other tests. You will need to modify

- `void printHelp(string progName)` : display the argument which calls your test.
- `int main(int argc, const char **argv)` : call your test function. Yes, that's a 100-line collection of `else if` statements. (see "huge mess" , above)
- `void test_myTestName()` : your actual code.

1.5 Contributing applications

The source code for applications is in the '`apps/`' directory. The easiest way to get started is probably to copy everything from an existing directory, then modify the files accordingly.

You should update all these files:

- '`src/apps/MYDIR/CMakeLists.txt`'
- '`src/apps/CMakeLists.txt`'

By convention, your program should return `exit(0)` upon successful completion. If any problems arise, the program should return a number greater than 0.

1.6 Contributing system code

1.6.1 Adding/removing to/from the build system

To add files to the Marsyas build system, use

```
addMarsystem MyMarSystemName
addClass MyClassName
```

> **Warning:** Remember to do `svn add`! Adding files to the build system does not add them to svn.

For text files (including source code `.h` and `.cpp`), please set

```
svn propset svn:eol-style native FILE
    for example
svn propset svn:eol-style native *.h *.cpp
```

It would be nice if svn did this automatically, but unfortunately it does not. :(

To remove files from the build system, use

```
removeMarsystem MyMarSystemName
removeClass MyClassName
```

Warning: Remember to do `svn rm`! Removing files from the build system does not remove them from svn.

1.6.2 Contributing non-MarSystem system code

If you have created a useful library or set of functions which you want to share with others, you may add it to the Marsyas source code. The new files should be placed in the '`src/marsyas/`' directory, and added to the build process.

Automatically

There is a Python script which automates this process; please see Section 1.6.1 [Adding/removing to/from the build system], page 3. If you are a masochist and wish to do this manually, see below.

Manually

The easiest way is to look for any appearances of `realvec` in the below files, and duplicate these entries using your '`myfile`'.

- '`src/marsyas/CMakeLists.txt`'

1.6.3 Contributing MarSystems

If you have created a useful MarSystem which you want to share with others, you may add it to the Marsyas source code. **THIS IS NOT REQUIRED FOR BUILDING YOUR OWN APPLICATONS!!!** See Section "Compiling and using a new MarSystem" in *Marsyas User Manual*.

Automatically

There is a Python script which automates this process; please see Section 1.6.1 [Adding/removing to/from the build system], page 3. If you are a masochist and wish to do this manually, see below.

Manually

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

- '`src/marsyas/CMakeLists.txt`'

1.7 Sending a patch

Checklist

- Does Marsyas still compile?
- Does Marsyas still pass `make test`?
- Did you follow Section 1.2 [Contributing source documentation], page 1 (if applicable)?

- Did you add your file(s) to the build process?

If the answer to all these questions is *yes*, then proceed.

Producing the patch

To produce a patch with svn, simply type

```
svn diff > mypatch.diff
```

and then send the resulting file to <marsyas-developers@lists.sourceforge.net>.

If you have SVN write access, you may simply type

```
svn ci
```

1.8 Building documentation

This manual is built with texinfo, and the source code documentation is created with doxygen. These software packages may be installed from

- texinfo
- doxygen
- graphviz

The latest version of the manual is in the SVN tree; create a new directory `doc-build`, run cmake, and make.

Source-highlighted examples are stored in the '`doc/source-doc/`' directory. These may be built with this additional program and the following command:

- GNU/source-highlight

```
scripts/generate-source-docs.sh
```

1.9 Build system details

The main file is src/CMakeLists.txt. Additional modules are in cmake-modules/.

1.10 Command-line arguments

We're trying to avoid have the same command-line arguments meaning different things in different programs. All arguments should have a short form (1-2 characters) and a long form. Here is a list of commands with specific meanings; if you want to do something different, then find a different pair of letters for the short form.

```
(null)              : same as -u
-u --usage
-h --help
-v --verbose
-s --silent
-q --quiet

-o --output     : output to a file

-g --gain
```

```
-ws --windowsize
-hs --hopsize
-ms --memorysize

-sa --start
-ln --length
-pl --plugin

-sr --samplerate
-ch --channels

-co --collection
```

We're still in the process of renaming arguments, so there may be inconsistencies between actual program behaviour and this list.

1.11 Making a release

Checklist:

1. From the top source dir, build in RELEASE mode and test:

   ```
   mkdir -p build-release/
   cd build-release/
   cmake ../src/
   make clean ; make -j3
   make test
   ```

2. Build the docs from the top source dir,

   ```
   mkdir -p build-doc
   cd build-doc
   cmake ../doc/
   make clean ; make -j3
   ```

3. In the top source dir, create tarball (adjust number as necessary):

   ```
   svn export . /tmp/marsyas-0.4.8/
   cd /tmp && tar -czf marsyas-0.4.8.tar.gz marsyas-0.4.8/
   ```

4. Upload the tarball. Either use the web interface, or command-line. Update the filename and `gperciva@` as needed.

   ```
   rsync /tmp/marsyas-0.4.8.tar.gz \
     gperciva@frs.sourceforge.net:/home/frs/projects/marsyas/marsyas/
   ```

   RSYNC CURRENTLY UNTESTED!

5. Set the default download to the new file. To my knowledge, this must be done with the online sourceforge interface.

6. Upload the docs from the top source dir,

   ```
   scripts/upload-docs.sh
   ```

   That script might require directory names to be adjusted.

2 Automatic testing

In an attempt to reduce the number of times we break working code, we have added some testing mechanisms to Marsyas.

2.1 Unit tests

These tests are highly focused on specific portions of code.

2.1.1 What are unit tests?

Unit tests are simple, short tests that test the functionality of individual modules in your source code. A module can be a method of a class, or can be smaller or larger parts of your code. Unit tests are usually developed at the same time code is written, independently testing each small component of your algorithm is on it's own.

The main goal of unit testing is to isolate each part of your code and ensure that each part is correct. This allows you to refactor your code at a later time with the confidence that your refactoring gives the same behaviour as the original code. This allows you to quickly write code that works, and then later come and refactor your code to make it run faster.

Unit testing also allows you to check whether a particular piece of code still works properly. This is useful in a large software project such as Marsyas which has currently over 180,000 lines of code and many developers on different continents.

Unit tests can also provide a sort of living documentation for the system. Well written unit tests have example code in them that can help new developers understand how to use different MarSystems. Written documentation added to the unit tests also helps in this regard, and often the best place for verbose documentation is in the unit tests rather than in the main functions themselves, where they can occasionally hide the beautiful logic therein.

Test-Driven Development (TDD) is a technique where you first write your tests, and then write the actual that will make the tests pass. In TDD you have short iterative development cycles on pre-written test cases that define desired improvements or new functionality. On each iteration of the cycle, you write just the code that you need to make the tests pass. With TDD you write just the code that you need, not the code you think you might need at some point in the future.

Cxxtest unit testing framework that is built into Marsyas lets you write your tests either before, as in standard unit testing, or after, as in TDD.

2.1.2 What testing framework does Marsyas use?

Marsyas uses the GPLed Cxxtest (http://cxxtest.sourceforge.net/) testing framework. Cxxtest is a JUnit/CppUnit/xUnit-like framework for C++. We chose Cxxtest because it is lightweight, easy-to-use, is very portable and is distributed under the GPL licence. It is easier to use than other C++ testing frameworks and features a very rich set of assertions.

An example can be found in marsyas/src/tests/unit_tests/TestSelector.h

```
#include "Selector.h"
using namespace Marsyas;
```

```
class Selector_runner : public CxxTest::TestSuite
{
public:
  void
  setUp()
  {
// ... setup the "in" realvec ...
  }

  void test_all_input_copied_to_output_by_default(void)
  {
  selector->myProcess(in,out);
   TS_ASSERT_EQUALS(out(0,0), 0.1);
  }
}
```

In this example, we setup the input realvec "in" in the function "setup()". We then add a test by creating a new method that begins with the word "test_".

2.1.3 How do I add a new test?

The easiest way to add a new test is to copy the file marsyas/src/tests/unit_tests/TestSample.h to a new file of your choice, for example, to "TestAutoCorrelation.h". "TestSample.h" has sample code in it that will help you get started, and contains copious documentation.

Add new tests for all the parts of your class that you want to test. Remember to start all the new function names with the string "test_" so that CxxTest knows that this function is one that you wish to test.

You then need to add a line to "marsyas/src/tests/unit_tests/CMakeLists.txt" to tell CMake to compile this new file:

```
marsyas_unit_test(AubioYin AubioYin_runner.cpp)
```

In order to generate the AubioYin_runner.cpp file from the TestAubioYin.h file, you then need to run the command:

./scripts/generate-unit-test-cpp-files.py

2.1.4 How do I run the tests?

To run the tests, you first need to enable testing in your build directory. To do this, do something like:

```
cd marsyas
mkdir build-with-tests
cd build-with-tests
ccmake ../src
```

Then type "t" to see the advanced options and turn on the "BUILD_TESTS" option.

The tests will then be automatically built if you then compile Marsyas:

```
make -j3
```

2.2 Black-box tests

These tests are deliberately simple and easy to understand.

2.2.1 What does "black-box" mean?

Definition

The term "black-box testing" seems to mean something different to each project, website, and software engineering academic. So I shall define what we mean by "black-box testing" in the context of Marsyas "black-box testing" :

> Does it work the way it used to?

This is checked as follows:

1. Pick a program to test. Find some input for the program (generally a sound file). Save the input.
2. Run the program on the saved input. Save the output (either another sound file, or some text).
3. Wait / change stuff / let other people change stuff.
4. Run the program with the initial input. Compare the current output to the initially stored output. If they are different, the test fails.
5. Goto step 3.

In other words, these tests do not check for *correctness*; they simply check for *consistency*. Fixing a bug could result in a test "failing" . However, this is not a problem: see Section 2.2.4 [When a test fails...], page 10

2.2.2 How do I write a black-box test?

Writing a black-box test is very easy.

1) Run your program with the normal command line flags on an input audio file. I would suggest to use one of the files in the marsyas/src/tests/black-box/input directory if possible. To test the waveform generation functionality of sound2png I ran it with the following flags:

```
sound2png -m waveform marsyas/src/tests/black-box/input/right.wav waveform.png
```

2) Copy the output file to the marsyas/src/tests/black-box/output directory

```
cp waveform.png marsyas/src/tests/black-box/output
```

3) Add a section to marsyas/src/tests/black-box/CMakeLists.txt for your new test:

```
set( ARGUMENTS
-m waveform
)
black-box_explicit(sound2png_waveform right.wav waveform.png sound2png "${ARGUMENTS}")
```

The first section "ARGUMENTS" are where you should put all the command line arguments for your program. "black-box_explicit" is a test MACRO that we've written to help make it easy to write tests like this, and it takes 5 arguments:

```
macro(black-box_audio REG_NAME REG_INPUT REG_OUTPUT REG_COMMAND REG_ARGS)
```

REG_NAME is the name of your test. Each test must be named differently, if not, the duplicate tests won't be run.

REG_INPUT is the input file for your test. In our case this was “right.wav”

REG_OUTPUT is the output file for your test. In our case this was “waveform.png”

REG_COMMAND is the command to run, “sound2png” in this example.

REG_ARGS are the command line arguments to REG_COMMAND.

2.2.3 How do the tests work?

List of tests

Tests are defined in ‘`src/tests/black-box/CMakeLists.txt`’. Each test is split into two steps: creating an audio file, and comparing that audio file to the previous (working) audio file.

Approximate matching (no rounding errors)

When an audio file is specified as an “answer” file,

```
audioCompare phasevoder.au ../output/right-phasevocoder.au
```

we attempt to match the files approximately, to avoid rounding mismatches on different operating systems. Each sample must be very close to the corresponding sample in the answer file, but they need not be exact.

2.2.4 When a test fails...

Don’t panic!

If a test fails as a result of your work, remember that these are *consistency* tests, not *correctness* tests. Do you expect your work to produce any different output for that particular test?

For example, if you discover (and fix) a bug in the inverse FFT, the phasevocoder test will probably “fail” . This is to be expected: the previously-recorded output of the phasevocoder faithfully archived the buggy output, so the bug-free output is detected as different.

On the other hand, suppose you are adding a new classifier for machine learning, and the Windowing test breaks. This is not expected; a new feature should not impact basic functions like taking a Hanning window! In this case, you should investigate before committing your changes to svn.

Updating the test

If you are certain that your patch (and new output file) are good, then you should update the answer file. This is simply a matter of copying your new output file over the old answer file.

The new output file is found in ‘`BUILD_DIR/tests/black-box/`’. You may also create the file manually yourself; the exact command-line arguments used for each test can be seen with:

```
make test ARGS="-V"
```

Please commit the changed ‘`src/tests/black-box/output/<FILE.au>`’ in a separate svn commit, and make sure the log message explains that your new output is superior to the old one.

Temporarily disabling a test

If you are planning on doing a lot of work on part of Marsyas (which would result in tests failing, but having no working output yet), tests may be commented out in the '`src/tests/black-box/CMakeLists.txt`' file. Again, please commit this change in a separate svn commit with a log message that explains this.

2.2.5 Why should I care?

Developers

Think of these tests as a mutual assistance pact: you should care about not breaking other people's code, because other people will care about not breaking *your* code.

Of course, this requires that you create black-box tests for your own code. Due to practicality, we can't check every single case of every single program. So instead, create one or two tests which investigate as many things as possible. For example, instead of simply testing if sfplay can output a sound file, we test changing the gain, starting at a specific time, and only playing for a specific length.

We recommend discussing potential black-box tests on the developer mailist.

New users

These tests are also very useful if you start investigating a new aspect of Marsyas. Currently there are many unmaintained MarSystems, applications, and projects in Marsyas. New users can easily waste hours trying to use part of the codebase which has been broken for months.

Having a testing mechanism means that users know that the code is working – at least, working for the exact command and input that the test uses. If (when?) a user has problems getting something to work in Marsyas, he can turn to the regtests: if a regtest passes but his own code doesn't work, he can compare his code against the regtest code.

Trust me, by now I've probably spent 20 hours trying to make broken code to work – sometimes when the developers already knew it was broken! Quite apart from the waste of time, it's incredibly demoralizing. It was this problem that inspired me to create these tests.

2.3 Daily Auto-tester

We run the script

```
scripts/dailytest.sh
```

every day. This script performs a few tasks:

1. Exports a clean copy of the source tree from svn.
2. Builds Marsyas.
3. Runs the tests.
4. Builds the documentation.

An email is sent to some developers with the results of these tests. If you would like to receive these daily emails, please enquire on the developer mailist.

The Index

(Index is nonexistent)