SCIRun Module Generation

SCIRun 5.0 Documentation

Center for Integrative Biomedical Computing
Scientific Computing & Imaging Institute
University of Utah

SCIRun software download:

http://software.sci.utah.edu

Center for Integrative Biomedical Computing:

http://www.sci.utah.edu/cibc

This project was supported by grants from the National Center for Research Resources **(5P41RR012553-14)** and the National Institute of General Medical Sciences **(8 P41 GM103545-14)** from the National Institutes of Health.

Author(s):
Jess Tate

Contents

Chapter 1

SCIRun Overview

This tutorial demonstrates how to create new modules in SCIRun 5.0 . It will walk through all the files needed and the basic module structure used by modules. These instructions assume a basic understanding in C++ coding and other basic programing skills

1.1 Software requirements

1.1.1 SCIRun 5.0

Download SCIRun version 5.0 from the SCI software portal. Make sure to update to the most up-to-date version of the source code available, which will include the latest bug fixes. Alternatively, use git to clone the SCIRun repository (https://github.com/SCIInstitute/SCIRun.git). We suggest creating a fork of the repository so that you can track your changes and create pull requests to the SCIRun repository (Section 1.1.3).

1.1.2 Compilers, Dependencies Development Tools

SCIRun will need to be built from the source code in order to test and use any modules written. Make sure that qt 4.8, git, cmake, and the latest c++ compilers for the operating system are installed.

1.1.3 Creating Your SCIRun Fork

With your own github account, go to the SCIRun github page. Click the fork button on the upper right side of the page. It will ask you where to move the fork to, chose your own account. Once the repository is forked, clone it to your local machine with the following command.

```
$ git clone https://github.com/[yourgithubaccount]/SCIRun.git
```

After the the code is cloned, navigate to the repository directory and add the upstream path to the original SCIRun repository.

```
$ git remote add upstream https://github.com/SCIInstitute/SCIRun.git
```

You should be able to see both your and the original repositories when you use the command:

```
$ git remote -v
```

The fork is good to go, but you will need to sync the fork occasionally to keep up with the changes in the main repository. To sync your fork, use the following commands:

```
$ git fetch upstream
$ git checkout master
$ git merge upstream/master
```

You should sync and merge your fork before you start a new module and before you create a pull request. It is a good practice to create a new branch in your fork for every module you will be adding. The command to create a new branch is:

```
$ git checkout -b [branch_name]
```

Please see the github help page for more information.

Chapter 2

Files Needed for a New Module

This chapter will describe the files need to create a module in SCIRun. Each file will be described and a template example will be provided. These template files are all included in the source code in the template directories.

Scope: Overview of Files Needed for each Module - Module Configuration File - Module Source Code - Module UI Code - Algorithm Code

2.1 Overview of Files Needed for each Module

There are only three files required to create a module, though more may be needed depending on the function of the module. In addition to the required module source code and header files (*modulename.cc* and *modulename.h*), a module configuration file is needed. The module configuration file (*modulename.module*) contains a description of the module and its state and names all the files needed for the module to be included in SCIRun.

Simple modules without user interfaces (UIs) can be created with the previously list three files alone. However, if the module function needs a UI, there are three additional files needed. SCIRun can generate a UI for a module without these, but the functionality will be very limited to nonexistent. The qt ui file (*modulenameDialog.ui*) describes the graphics and hooks of the UI can be created using the qt UI editor. Module UIs also require a code and header file (*modulenameDialog.cc* and *modulenameDialog.h*.

Most modules, especially those requiring more than minimal code, should also have algorithm code to allow for greater portability and code control. This algorithm code and header file (*modulenameAlgo.cc* and *modulenameAlgo.h*) should contain all the computation of the module. Though it is possible to build modules without these algorithm files, it is considered good practice to do so.

It is worth noting that each of the *CMakeLists.txt* files in the directories of all of the files (except the module config file). See the examples in the following chapters for details.

2.2 Module Configuration File

The module configuration file contains all the information need for the module factory to create necessary linkage and helper files for module to be properly included into SCIRun. Module configuration files should be located in *src/Modules/Factory/Config/*. It is a text file that describes fields specific to the module delimited by curly brackets. There are three

fields “module”, “algorithm”, and “UI” and within each field are subfields “name” and “header”, and others depending on the field. The following is an example that reflects the template files included in the source code.

```
{
  "module": {
    "name": "@ModuleName@",
    "namespace": "Fields",
    "status": "description of status",
    "description": "description of module",
    "header": "Modules/Template/ModuleTemplate.h"
  },
  "algorithm": {
    "name": "@AlgorithmName@Algo",
    "namespace": "Fields",
    "header": "Core/Algorithms/Template/AlgorithmTemplate.h"
  },
  "UI": {
    "name": "@ModuleName@Dialog",
    "header": "Interface/Modules/Template/ModuleDialog.h"
  }
}
```

This config file example would not build. We will include specific examples that will build and work in following chapters of this tutorial (Chapters—).

As mentioned before, the UI and algorithm files are not required to generated a module, therefore the subfields for the “algorithm” or “UI” fields can changed to “N/A” to indicate that these files do not exist. Please refer to Section **??** for an example.

2.3 Module Source Code

The Module source code consist of a .cc and .h file that code the actual function of the module. However, since most modules use an algorithm file, these files can also be considered the the code that pulls all the relevant information from the algorithm, the UI, and other modules in order to achieve it’s proper function. These files should be located in the proper directory with the *src/Modules/* directory. For example purposes, we will show and discuss the template files included in the *src/Modules/Template/* directory.

2.3.1 Module Header File

The module header file functions as a typical C++ header file, containing code establishing the module object and structure. The *ModuleTemplate.h* file found in *src/Modules/Template/* provides an example of the kind of coding needed for a module header. The relevant functions are included here, with annotated comments:

```
// makes sure that headers aren't loaded multiple times.
// This requires the string to be unique to this file.
// standard convention incorporates the file path and filename.
```

```
#ifndef MODULES_FIELDS_@ModuleName@_H__
#define MODULES_FIELDS_@ModuleName@_H__

#include <Dataflow/Network/Module.h>
#include <Modules/Fields/share.h>
// share.h must be the last include, or it will not build on windows systems.

namespace SCIRun {
namespace Modules {
namespace Fields {
// this final namespace needs to match the .module file
// in src/Modules/Factory/Config/

  // define module ports.
  // Can have any number of ports (including none), and dynamic ports.
  class SCISHARE @ModuleName@ : public SCIRun::Dataflow::Networks::Module,
    public Has1InputPort<FieldPortTag>,
    public Has1OutputPort<FieldPortTag>
  {
  public:
    // these functions are required for all modules
    @ModuleName@();
    virtual void execute();
    virtual void setStateDefaults();

    //name the ports and datatype.
    INPUT_PORT(0, InputField, Field);
    OUTPUT_PORT(0, OutputField, Field);

    // this is needed for the module factory
    static const Dataflow::Networks::ModuleLookupInfo staticInfo_;
  };
}}}
#endif
```

One of the key functions of this header file is the definition of the ports used by the module. This template uses one input and output, but any number can be used by changing the number and defining all the port types. To use two inputs and outputs:

```
public Has2InputPorts<FieldPortTag,FieldPortTag>,
public Has2OutputPorts<FieldPortTag,FieldPortTag>
```

If no there are no input or output ports, the commands are:

```
public HasNoInputPorts,
public HasNoOutputPorts
```

Dynamic ports are also possible for the inputs. Dynamic ports are essentially a vector of ports, and are counted as a single port in the header. For a single dynamic port, then a static port and a dynamic port:

```
public Has1InputPort<DynamicPortTag<FieldPortTag>>,
public Has2InputPorts<FieldPortTag,DynamicPortTag<FieldPortTag>>
```

Here is a list of port tags that can be used in SCIRun:

- MatrixPortTag
- ScalarPortTag
- StringPortTag
- FieldPortTag
- GeometryPortTag
- ColorMapPortTag
- BundlePortTag
- NrrdPortTag
- DatatypePortTag

The header is also where the ports are named and the datatype is declared. It is import for the name of each port to be unique, including all the inputs and outputs. When the port is declared and named, the datatype is also specified. This declares the datatype expected by the port and can be a subset of the port tag type, e.g., DenseMatrix instead of Matrix. However, it can be better to do this within the module to control the exception.

If there is a UI with the module in question, the state variables may be needed to pass data between the module and the UI. State variables will need to be declared here as public (see Section 4.3 for an example). The 'setStateDefault' function is how the default state variables are set. If there is no UI and therefore no state variables, this function can be set to empty in this file and omitted from the .cc file.

2.3.2 Module Code File

The module header file functions as a typical C++ file, containing the functions for the module. The *ModuleTemplate.cc* file found in *src/Modules/Template/* provides an example of the kind of coding needed for a module .cc file. The relevant functions are included here, with annotated comments:

```
#include <Modules/Fields/@ModuleName@.h>
#include <Core/Datatypes/Legacy/Field/Field.h>
#include <Core/Algorithms/Field/@ModuleName@Algo.h>

using namespace SCIRun::Modules::Fields;
using namespace SCIRun::Core::Datatypes;
using namespace SCIRun::Dataflow::Networks;
using namespace SCIRun::Core::Algorithms::Fields;

// this defines the location of the module in the module list.
// "NewField" is the catagory and "SCIRun" is the package.
```

```
const ModuleLookupInfo @ModuleName@::staticInfo_("@ModuleName@",
          "NewField", "SCIRun");

@ModuleName@::@ModuleName@() : Module(staticInfo_)
{
  //initialize all ports.
  INITIALIZE_PORT(InputField);
  INITIALIZE_PORT(OutputField);
}

void @ModuleName@::setStateDefaults()
{
  auto state = get_state();
  setStateBoolFromAlgo(Parameters::Knob1);
  setStateDoubleFromAlgo(Parameters::Knob2);
}

void @ModuleName@::execute()
{
  // get input from ports
  auto field = getRequiredInput(InputField);
  // get parameters from UI
  setAlgoBoolFromState(Parameters::Knob1);
  setAlgoDoubleFromState(Parameters::Knob2);
  // run algorithm code.
  auto output = algo().run(withInputData((InputField, field)));
  //send to output port
  sendOutputFromAlgorithm(OutputField, output);
}
```

As shown in this template example, the model.cc file contains mostly constructors and sends the inputs to the SCIRun algorithm. Most modules should follow this practice, which allows for easier maintenance of common algorithms.

2.4 Module UI Code

There are three files needed to set up a UI for a module, a design file, a header file, and a .cc file. These files should all be located in the same directory within the *src/Interface/Modules/*. We will show the examples located in *src/Interface/Modules/Template* as examples of the core functions needed.

2.4.1 Module Design File

The module design file is an xml file that describes the UI structure. This file can be created and edited in the Qt editor. Figure 2.1 shows the example template *ModuleDesignerFile.ui* within the Qt editor. As shown, the user can interactively modify the placement of the widgets in the window. The Widget Box window allows the user to choose and place new

objects within the window. The Property Editor allows for the modification of properties of the various objects and widgets within the UI, including size, type of input, names, etc. With the Object Inspector window, the hierarchy and organization of the UI can be changed.

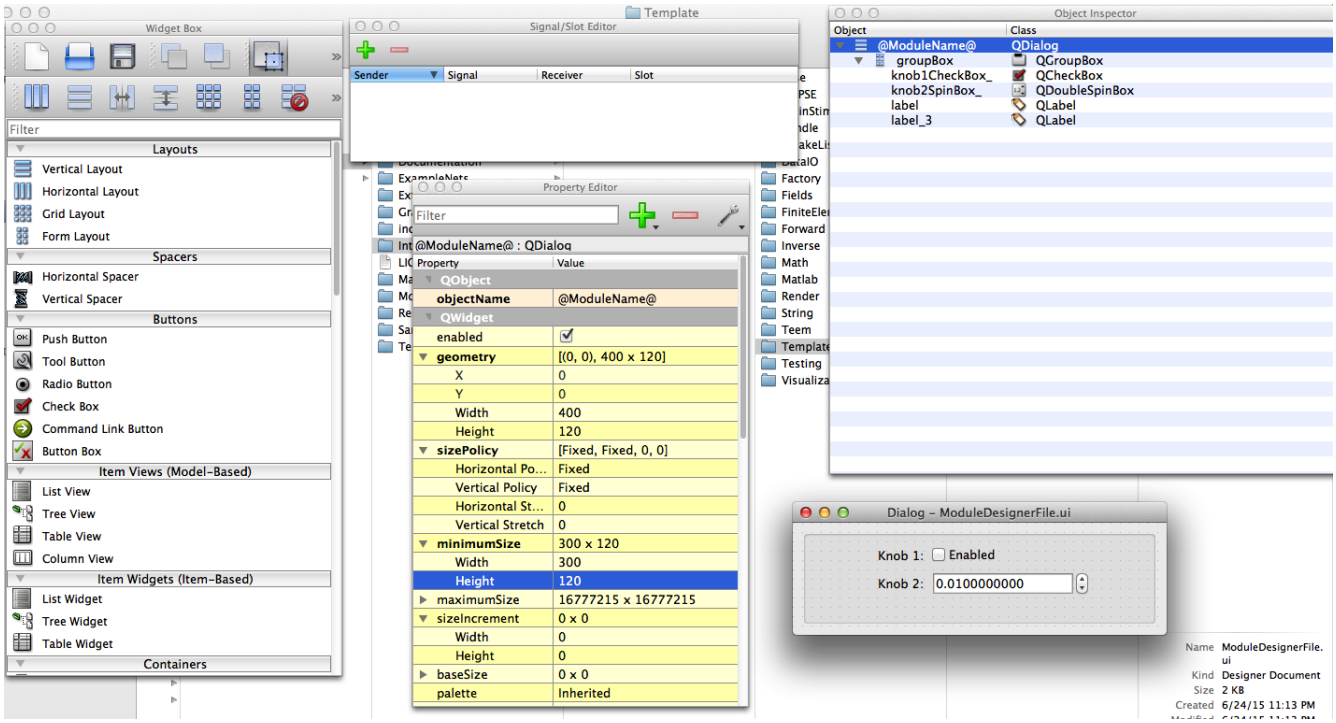

Figure 2.1. Template module interface design file as seen in the Qt editor.

When using the editor to make a module UI, there are a few things to consider. First, make sure all the relevant objects, including the name of UI (QDialog) is consistent with module dialog code. You can change the size and placement of objects with the property manager, but make sure that you leave some buffer space as some OS will interpret the file slightly differently. The structure of the UI can be changed or destroyed, look at some of the existing modules for examples.

2.4.2 Module Dialog Header

The module dialog header performs as a traditional C++ header for the module dialog code. Shown here is the example *ModuleDialog.h* in the *src/Interface/Modules/Template* folder.

```
#ifndef INTERFACE_MODULES_@ModuleName@DIALOG_H
#define INTERFACE_MODULES_@ModuleName@DIALOG_H

//This file is created from the @ModuleName@Dialog.ui in the module factory.
#include <Interface/Modules/Fields/ui_@ModuleName@Dialog.h>
#include <boost/shared_ptr.hpp>
#include <Interface/Modules/Base/ModuleDialogGeneric.h>
#include <Interface/Modules/Fields/share.h>
```

```
namespace SCIRun {
namespace Gui {

class SCISHARE @ModuleName@Dialog : public ModuleDialogGeneric,
  public Ui::@ModuleName@
{
Q_OBJECT

public:
  @ModuleName@Dialog(const std::string& name,
    SCIRun::Dataflow::Networks::ModuleStateHandle state,
    QWidget* parent = 0);
    //this function would be from pulling data from module,
    // usually to change the UI.
  virtual void pull() override;
};
}}
#endif
```

The module dialog header file will be very similar for each module, with only the names of the module and a few different functions declared here.

2.4.3 Module Dialog Code

The module dialog cc file is used with Qt to establish the functionality of module UI. Shown here is the example *ModuleDialog.cc* in the *src/Interface/Modules/Template* folder.

```
#include <Interface/Modules/Fields/@ModuleName@Dialog.h>
#include <Core/Algorithms/Field/@ModuleName@Algo.h>

using namespace SCIRun::Gui;
using namespace SCIRun::Dataflow::Networks;
using namespace SCIRun::Core::Algorithms::Fields;

@ModuleName@Dialog::@ModuleName@Dialog(const std::string& name,
  ModuleStateHandle state,
  QWidget* parent /* = 0 */)
  : ModuleDialogGeneric(state, parent)
{
  setupUi(this);
  setWindowTitle(QString::fromStdString(name));
  fixSize();

//get values from UI and send to algorithm
  addCheckBoxManager(knob1CheckBox_, Parameters::Knob1);
  addDoubleSpinBoxManager(knob2SpinBox_, Parameters::Knob2);
}
```

```
void @ModuleName@Dialog::pull()
{
// pull the code from the module and set in the dialog.
// make changes necessary.
  pull_newVersionToReplaceOld();
}
```

The module dialog code is mostly for passing data between the module and the UI and changing the UI when require by the module. Parameters and inputs can be passed straight to the algorithm, as templated in this example. The 'pull' functions are optional to use data from the module and algorithm to either display in the UI or to use to change the options or appearance of the UI. There can be other forms of 'pull', such as 'pullSpecial' A 'push' function can be used in conjunction with 'pull'. These functions, 'pull' and 'push' are automatic functions in the module UI.

2.5 Algorithm Code

The Module algorithm files are where most of the computation code should live. There are two files, a header and a .cc file. Making algorithm files can be tricky for a beginner, because there are several options to use due to the flexible nature of SCIRun. However, the module algorithms have more in common that it first appears. The trick to implementing a new module algorithm (as with most things) is look at several other modules that have similar functions and try to emulate those methods.

Module algorithm code belongs in the relevant directory within the *src/Core/Algorithms/* directory. Provided in this section are some template examples, found in *src/Core/Algorithms/Template/*

2.5.1 Module Algorithm Header

The module algorithm header performs as a traditional C++ header for the module algorithm code. Shown here is the example *AlgorithmTemplate.h* in the *src/Core/Algorithm/Template/* folder.

```
#ifndef CORE_ALGORITHMS_FIELDS_@AlgorithmName@_H
#define CORE_ALGORITHMS_FIELDS_@AlgorithmName@_H

#include <Core/Algorithms/Base/AlgorithmBase.h>
#include <Core/Algorithms/Field/share.h>

namespace SCIRun {
  namespace Core {
    namespace Algorithms {
      namespace Fields {
// declare parametes and options in header when not part of standard names.
        ALGORITHM_PARAMETER_DECL(Knob1);
        ALGORITHM_PARAMETER_DECL(Knob2);

        class SCISHARE @AlgorithmName@Algo : public AlgorithmBase
        {
```

```
        public:
          @AlgorithmName@Algo();
          virtual AlgorithmOutput run_generic(const AlgorithmInput& input) const;
        };
      }}}}
#endif
```

Key difference that may occur in the algorithm header file are the function and variable declarations. Parameters and options for the algorithm may need to be declared here if they are not included in the recognized list (listed in *Core/Algorithms/Base/AlgorithmVariableNames.h*).

2.5.2 Module Algorithm Code

The module algorithm cc file should contain the majority of the computation necessary for the module. Shown here is the example *AlgorithmTemplate.h* in the *src/Core/Algorithm/Template/* folder.

```
#include <Core/Algorithms/Field/@AlgorithmName@Algo.h>
#include <Core/Algorithms/Base/AlgorithmVariableNames.h>
#include <Core/Algorithms/Base/AlgorithmPreconditions.h>
#include <Core/Datatypes/Legacy/Field/FieldInformation.h>
#include <Core/Datatypes/Legacy/Field/VField.h>
#include <Core/Datatypes/Legacy/Field/VMesh.h>
#include <Core/Logging/Log.h>

using namespace SCIRun;
using namespace SCIRun::Core::Datatypes;
using namespace SCIRun::Core::Algorithms;
using namespace SCIRun::Core::Algorithms::Fields;

//  this function is for setting defaults for state variables.
//  Mostly for UI variables.
@AlgorithmName@Algo::@AlgorithmName@Algo()
{
  using namespace Parameters;
  addParameter(Knob1, false);
  addParameter(Knob2, 1.0);
}

//main algorithm function
AlgorithmOutput @AlgorithmName@Algo::run_generic(const
     AlgorithmInput& input) const
{
  auto inputField = input.get<Field>(Variables::InputField);

  FieldHandle outputField(inputField->deep_clone());
  double knob2 = get(Parameters::Knob2).toDouble();
  if (get(Parameters::Knob1).getBool())
```

```
  {
    // do something
  }
  else
  {
    // do something else
  }

  AlgorithmOutput output;
  output[Variables::OutputField] = outputField;
  return output;
}
```

This template shows the algorithm using some of the defined names from *Core/Algorithms/Base/AlgorithmV* using the Variable namespace. This allows for easy use of common inputs and output to the algorithm. If other or more values are needed, they can be declared in the header. Also of note, the default values for the UI are set in the '@AlgorithmName@Algo()' function, and then in the module code, the 'setStateDefault ' function pulls the values from the algorithm. This only works if the algorithm files are linked in the module configuration file.

There are several algorithms already implemented in SCIRun. If there there are modules that have similar functionality, you may be able to use some of the functionality already implemented. The module may still need it's own algorithm file.

Chapter 3

Example: Simple Module Without UI

This chapter describes how to create a very simple module in SCIRun. We will show how to make a simple module that outputs a simple string. This example will show the basics of the functions and code used by SCIRun to create and run modules.

Scope: Module Config File - Module Header File - Module Source Code - Building and Testing

3.1 Module Config File

If you have created a fork from the SCIRun git repository, begin by creating a new branch in your repository. Be sure to commit your changes to your repository often, as this can help you and the developers fix and make improvements to the code. It is often easiest to modify existing code to fit your purposes than create new code, so determine a module that has similar functionality or structure to the new module. If desired, there is also the template files described in Chapter 2 to use as a basis. In this example, we will provide the code needed, so it is not necessary to copy another module.

Begin with the module config file. Create a new text file in module factory configuration directory (*src/Modules/Factory/Config/*) for the new module. I should be named *TestModuleSimple.module* or something similar. The text of the file should be:

```
{
  "module": {
    "name": "TestModuleSimple",
    "namespace": "StringManip",
    "status": "new module",
    "description": "This is a simple module to show how to make new modules.",
    "header": "Modules/String/TestModuleSimple.h"
  },
  "algorithm": {
    "name": "N/A",
    "namespace": "N/A",
    "header": "N/A"
  },
  "UI": {
    "name": "N/A",
    "header": "N/A"
```

```
  }
}
```

The exact text of the status and description can be whatever the creator desires. The names of the module and filenames can also be different, but they must match the module code.

3.2 Module Header File

Now we move to the module code. The module will need to be placed in one of the directories in *src/Modules/*, so choose the directory that fits the modules use the best (do not place the module code in *Factory* or *Template*, and *Legacy* is generally for converted modules from earlier versions of SCIRun) Since this module will be only have a string output, we will place the module code in *src/Modules/String/*. In this directory create a file called *TestModuleSimple.h*. This file will be very similar to the *ModuleTemplate.h* file shown earlier. In addition to the SCIRun license information, the content of the header file should be:

```
#ifndef MODULES_STRING_TestModuleSimple_H
#define MODULES_STRING_TestModuleSimple_H

#include <Dataflow/Network/Module.h>
#include <Modules/Fields/share.h>

namespace SCIRun {
namespace Modules {
namespace StringManip {

class SCISHARE TestModuleSimple : public SCIRun::Dataflow::Networks::Module,
public HasNoInputPorts,
public Has1OutputPort<StringPortTag>
{
public:
  TestModuleSimple();
  virtual void execute();
  virtual void setStateDefaults() {};

  OUTPUT_PORT(0, OutputString, String);

  static const Dataflow::Networks::ModuleLookupInfo staticInfo_;
};
}}}
#endif
```

As mentioned in Section 2.2, the header files for most modules do not vary significantly. This example in particular contains only elements common to most other modules. The key to creating the header files is to ensure that the module name is correct in every place it occurs, that the namespace (StringManip) matches the module config file and that the ports are numbered and labeled correctly.

3.3 Module Source Code

The final file needed for this module is the source code file (*ModuleTemplate.cc*). The functionality used in this module is minimal to show how the essential functions. With the license and other comments, the file should contain:

```
#include <Modules/String/TestModuleSimple.h>
#include <Core/Datatypes/String.h>

using namespace SCIRun;
using namespace SCIRun::Modules::StringManip;
using namespace SCIRun::Core::Datatypes;
using namespace SCIRun::Dataflow::Networks;

/// @class TestModuleSimple
/// @brief This module splits out a string.

const ModuleLookupInfo TestModuleSimple::staticInfo_("TestModuleSimple",
       "String", "SCIRun");

TestModuleSimple::TestModuleSimple() : Module(staticInfo_,false)
{
  INITIALIZE_PORT(OutputString);
}

void
TestModuleSimple::execute()
{
  std::string message_string;

  message_string = "[Personalize your  message here.]";

  StringHandle msH(new String(message_string));
  sendOutput(OutputString, msH);
}
```

3.4 Building and Testing

After these files are modified correctly, the only step remaining before building is adding the module code and header to the compiler list. Open the *src/Modules/String/CMakeLists.txt* file. Add *TestModuleSimple.cc* and *TestModuleSimple.h* to the respective list. There will more to the file, but the relevant sections should look something like this:

```
SET(Modules_String_SRCS
  CreateString.cc
  NetworkNotes.cc
  TestModuleSimple.cc
)
```

```
SET(Modules_String_HEADERS
  CreateString.h
  NetworkNotes.h
  share.h
  TestModuleSimple.h
)
```

After changing the CMakeList.txt file, build SCIRun using the build script, or if you have already built SCIRun recently, go to the *SCIRun_root/bin/SCIRun* directory and run make. Take note of any build errors, if there is a problem in the with any module factory files, make sure that there are no mistakes in the the module configuration file and build again. Check out the common build errors in section [TODO].

After SCIRun builds completely, Launch SCIRun and test the module. You can use the PrintDatatype module to view the string that this module outputs. Other modules will require more testing, but due to the very simple nature of the module you can know that if the messages matches what you expect, then it is working properly.

Chapter 4

Example: Simple Module With UI

In this chapter, we will build off the module that we described in the previous chapter to show how to add a UI and an input port. This module will print a message that comes from either the input port or the UI. We will show how to add a UI incrementally to help convey the principles that the software is based upon. This incremental approach allows the user to copy this approach with more complicated module as it provides sanity checks for the user.

Scope: Duplicate the Previous Module - Creating a Custom UI - Connecting UI to the Module - Adding an Input Port - Finished Code

4.1 Duplicate the Previous Module

To begin, copy the *TestModuleSimple.module* in the *src/Modules/Factory/Config/* and name the copy *TestModuleSimpleUI.module*. Change the name and header field to reflect the new name of the module, as shown here:

```
"module": {
  "name": "TestModuleSimpleUI",
  "namespace": "StringManip",
  "status": "new module",
  "description": "This is a simple module to show how to make new modules.",
  "header": "Modules/String/TestModuleSimpleUI.h"
},
```

For now, leave the rest of the fields as 'N/A'; we will come back to those.

Next, copy the module code files *TestModuleSimple.h* and *TestModuleSimple.cc* in the *src/Modules/String/* directory and rename them appropriately (*TestModuleSimpleUI.h* and *TestModuleSimpleUI.cc*). In these new files, change all the references of the modules name to TestModuleSimpleUI. A find and replace function will manage most instances, but make sure that all of them are changed. There are 4 lines in each of the two files that need to be changed, with more than one change in some lines. The changes in *TestModuleSimpleUI.h* are these lines:

```
#ifndef MODULES_STRING_TestModuleSimpleUI_H
#define MODULES_STRING_TestModuleSimpleUI_H

...
```

```
class SCISHARE TestModuleSimpleUI : public SCIRun::Dataflow::Networks::Module,

...

public:
  TestModuleSimpleUI();

...
```

And for the *TestModuleSimpleUI.cc* file:

```
#include <Modules/String/TestModuleSimpleUI.h>



const ModuleLookupInfo TestModuleSimpleUI::staticInfo_("TestModuleSimpleUI",
       "String", "SCIRun");

TestModuleSimpleUI::TestModuleSimpleUI() : Module(staticInfo_)



void
TestModuleSimpleUI::execute()
{
```

Another change you may notice is to remove the 'false' input in the constructor line:

```
TestModuleSimpleUI::TestModuleSimpleUI() : Module(staticInfo_)
```

The 'false' options means that there is no module. Removing the options changes the input to 'true', which allows for a module UI. If no UI file is found, a default UI will be used.

With these changes, we should try to build. Make sure the files are added to the CMakeList.txt file in *src/Modules/String/* as we showed in the previous chapter. If there are build errors, check for spelling mismatches. Also, check out the common build errors in section [TODO]. Once SCIRun is built, you can try to add the new module to the workspace. SCIRun will give you a warning dialogue about not finding a UI file, so it will create a default one. This UI is not connected to anything, so it won't affect the module at all, but you should be able to open the UI and see it (a slider and two buttons). Check to make sure that the output is still the string that you expected. If everything is working properly, we can move onto the next step of adding our own module.

4.2 Creating a Custom UI

To create a new UI, we need to add three new files: a design file, and a cc and header file for the UI. We will need these files linked to the other module code, so we will modify the module config file again to add the name of the UI and the path to the header file. The naming convention often used is to add 'Dialog' to the end of the module name for the name of the UI and the names of the files.

```
"UI": {
    "name": "TestModuleSimpleUIDialog",
    "header": "Interface/Modules/String/TestModuleSimpleUIDialog.h"
  }
```

Next, we need to use the QT editor to design a module UI. Copy the QT module file from *src/Interface/Modules/Template/ModuleDesignerFile.ui* to *src/Interface/Modules/String/TestModuleSimp* Open the *TestModuleSimpleUIDialog.ui* file in the Qt editor, which provides a graphic method for modifying and compiling the design file. First, delete the check box and input scroll wheel widgets and delete the 'knob 2' label. Next, add a *line edit widget* by finding it in the *Widgdet Box* window (in the *Input Widget* section) and clicking and dragging it next to the remaining label. Change the text of the remaining label to 'My String:' or something similar. Finally, change the name of the of the *line edit widget* to *inputstring_* and the name of the *QDialog* object to *TestModuleSimpleUIDialog*. This can be done in the *Object Inspector* or in the *Property Editor* when the appropriate object is clicked. Figure 4.1 shows what the module should look like in the Qt editor.

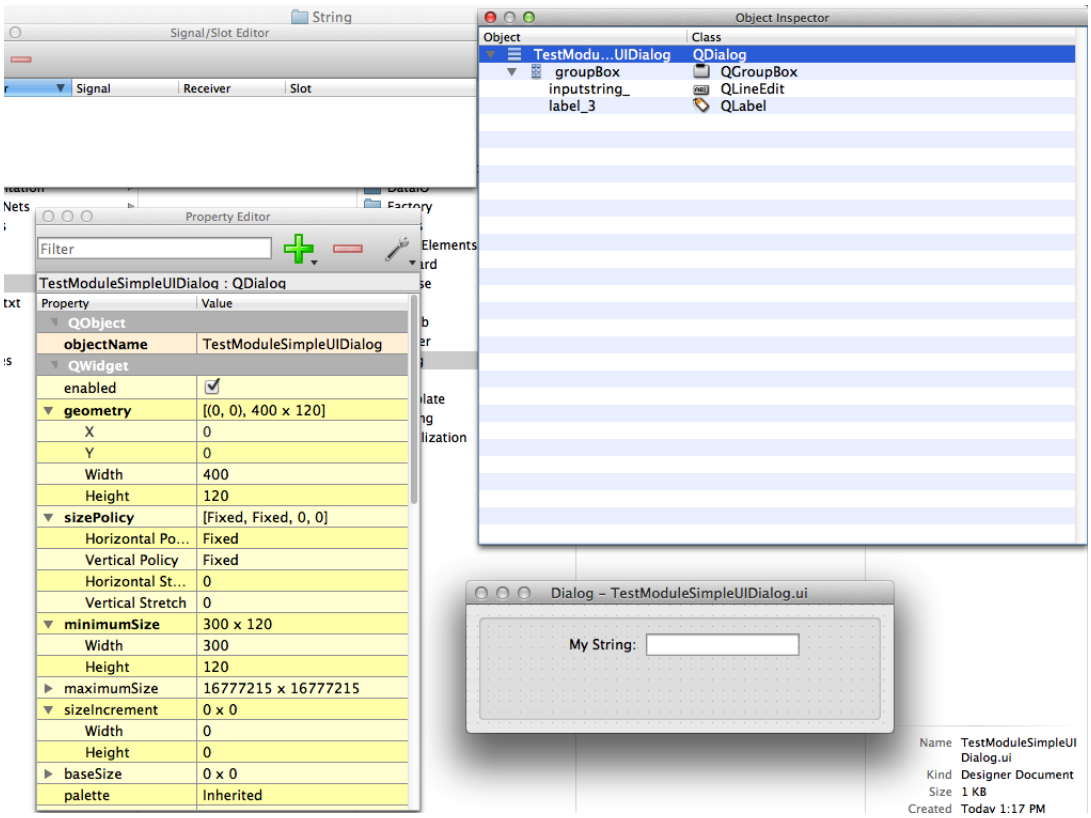

Figure 4.1. Module interface design file for the TestModuleSimpleUI module as seen in the Qt editor.

Now that the module UI is designed, we need to link it to the module with the module dialog code. Copy the *ModuleDialog.cc* and the *ModuleDialog.h* from the *src/Interface/Modules/Template/* directory to the *src/Interface/Modules/String/* directory, with the appropriate names (*TestModuleSimpleUIDialog.cc* and *TestModuleSimpleUIDialog.h*). For the *TestModuleSimpleUIDialog.h*, change the module name reference to the correct module name and delete the 'virtual void pull()' function. The code should be very similar to the following:

```
#ifndef INTERFACE_MODULES_STRING_TestModuleSimpleUIDialog_H
#define INTERFACE_MODULES_STRING_TestModuleSimpleUIDialog_H

#include <Interface/Modules/String/ui_TestModuleSimpleUIDialog.h>
#include <Interface/Modules/Base/ModuleDialogGeneric.h>
#include <Interface/Modules/String/share.h>

namespace SCIRun {
namespace Gui {

class SCISHARE TestModuleSimpleUIDialog : public ModuleDialogGeneric,
  public Ui::TestModuleSimpleUIDialog
{
Q_OBJECT
```

```
public:
  TestModuleSimpleUIDialog(const std::string& name,
    SCIRun::Dataflow::Networks::ModuleStateHandle state,
    QWidget* parent = 0);
};
}}
#endif
```

The *TestModuleSimpleUIDialog.cc* requires similar treatment, but will require the addition of few more changes. Add and include for the module header file, change the namespace from 'Field' to 'StringManip', and delete the last two lines from the main function. The code should be:

```
#include <Interface/Modules/String/TestModuleSimpleUIDialog.h>
#include <Modules/String/TestModuleSimpleUI.h>

using namespace SCIRun::Gui;
using namespace SCIRun::Dataflow::Networks;
using namespace SCIRun::Modules::StringManip;

TestModuleSimpleUIDialog::TestModuleSimpleUIDialog(const std::string& name,
  ModuleStateHandle state,
  QWidget* parent /* = 0 */)
  : ModuleDialogGeneric(state, parent)
{
  setupUi(this);
  setWindowTitle(QString::fromStdString(name));
  fixSize();
}
```

This should be enough to create a UI for the TestModuleSimpleUI module, but it will not be able interact yet. We will need to modify this file later to connect all the required inputs. For now, we can build SCIRun to test the UI design. Make sure that these three new files are added to the *CMakeList.txt* in the *src/Interface/Modules/String/* directory:

```
SET(Interface_Modules_String_FORMS

  ...

  TestModuleSimpleUIDialog.ui
)

SET(Interface_Modules_String_HEADERS
```

```
  TestModuleSimpleUIDialog.h
)

SET(Interface_Modules_String_SOURCES

  ...

  TestModuleSimpleUIDialog.cc
)
```

Once these files are added, SCIRun should build. Load SCIRun and place the TestModuleSimpleUI module. Open the UI for the module and make sure that looks correct.

4.3 Connecting UI to the Module

Now we will work on connecting the input from the UI to the code in the module. Begin by modifying the *TestModuleSimpleUIDialog.cc* to include a line that reads the input field and assigns it to a variable. This line needs to go near the end in the main function of the module dialog code

```
addLineEditManager(inputstring_,TestModuleSimpleUI::FormatString);
```

This function will read the value of 'inputstring_' and set it to 'FormatString', which we have included as if it was part of the 'TestModuleSimpleUI' namespace. We will need to include it as such, by adding it as a public function in the *TestModuleSimpleUI.h* file.

```
static Core::Algorithms::AlgorithmParameterName FormatString;
```

This should be the final declaration in the public list (after the 'staticInfo_' declaration). Another change in this file is to modify the 'setStateDefault' function so that it is not empty. Remove the curly brackets from this:

```
virtual void setStateDefaults() {};
```

so that it is:

```
virtual void setStateDefaults();
```

We need a couple more additions to make the value from the UI available for use in the main function code. In the *TestModuleSimpleUI.cc* file, add the following line to the before the main execute function, i.e., right after declaring the namespaces.

```
SCIRun::Core::Algorithms::AlgorithmParameterName
     PrintStringIntoString::FormatString("FormatString");
```

Next we need to be able to set the state defaults by creating context for the 'setStateDefault' function we just exposed. Add this function just before the execute function.

```
void TestModuleSimpleUI::setStateDefaults()
{
  auto state = get_state();
  state->setValue(FormatString,std::string ("[Insert message here]"));
}
```

With these three additions, the code should build. If you load the module, you should she the default message ("[Insert message here]") in the input field. Changing this will still not affect the output because the execute function is still hard coded for a specific message.

Now we need to change the execute function to use the UI inputs, which is very simple. Simply get the state of the module (`auto state = get_state();`), then assign the output string variable to `state -> getValue(FormatString).toString();` so that the function is as follows:

```
void
TestModuleSimpleUI::execute()
{
  std::string message_string;
  auto state = get_state();
  message_string = state -> getValue(FormatString).toString();
  StringHandle msH(new String(message_string));
  sendOutput(OutputString, msH);
}
```

After building the software, you should now see that the output of module will be the same as the string that is put in the input field in the module UI.

4.4 Adding an Input Port

With the UI implemented and working, we will now add an optional input port to the module. This functionality is fairly simple in SCIRun5. We need to add the port in the *TestModuleSimpleUI.h* file. First, replace the line:

```
public HasNoInputPorts,
```

with:

```
public Has1InputPort<StringPortTag>,
```

Next, we need to name the port in the list of public objects. Add:

```
INPUT_PORT(0, InputString, String);
```

near the output port declaration. These changes are all that are needed in the header file, but we need to initialize the port in the .cc file. In the *TestModuleSimpleUI.cc*, add the initializing line to the module constructor function, which should be then:

```
TestModuleSimpleUI::TestModuleSimpleUI() : Module(staticInfo_)
{
  INITIALIZE_PORT(InputString);
  INITIALIZE_PORT(OutputString);
}
```

These changes should allow the code to build with an input port, yet the input will not affect the output of the module.

In the main execute function in *TestModuleSimpleUI.cc*, we need to read whether there is an input, then use that input or the UI input if there is none. This is straight forward in SCIRun, the execute function will be:

```
void
TestModuleSimpleUI::execute()
{
  std::string message_string;
  auto  stringH = getOptionalInput(InputString);
  auto state = get_state();

  if (stringH && *stringH)
  {
    state -> setValue(FormatString, (*stringH) -> value());
  }

  message_string = state -> getValue(FormatString).toString();
  StringHandle msH(new String(message_string));
  sendOutput(OutputString, msH);
}
```

This code reads an optional input, checks if it is not empty, and if so then changes the state variable to the input. By changing the state variable before assigning it to the output, it changes the UI input string also.

This should be all the changes necessary to add inputs to this module. Build SCIRun, then test the module using the CreateString and PrintDatatype modules. When there is no input, the value in the UI field is the output. When there is an output, the input port is the same as the output port, and the UI input field is set to the input string. This prevents the user from changing the input string while there is a string in the input port.

For a slightly more complicated, yet much more useful module as an example, check out PrintStringIntoString. The setup code is mostly the same, except there are dynamic ports, so much of the code will look very similar.

4.5 Finished Code

Chapter 5

Example: Simple Module With Algorithm

In this chapter, we will show how to build a module with a simple algorithm and a simple UI. This chapter will build off the principles established in the previous examples. We will use SCIRun to create a module that will perform a simple sorting algorithm on a matrix. This example will show how to use module algorithm files with a module UI to implement simple algorithms into modules. We will build off some of the principles of the previous examples.

Scope: Module Overview - Module Configuration File - Module Code - Module UI Code - Module Algorithm Code - Building and Testing

5.1 Module Overview

As mention in the chapter introduction, we are going to create a module called SortMatrix that will sort the entries of a matrix in ascending or descending order. This module will use a simple simple quick sort algorithm with a Lomuto partition scheme. There are some implementations for vector sorting in the STL algorithm library, but this implementation works more generally on matrices and will hopefully be helpful in showing how to implement an algorithm from scratch.

There are eight files needed in total for this module: a module configuration file, module code and header file, a UI design file with UI code and header files, and algorithm code and header files. The first six were used in the previous example, but this chapter will show how to incorporate the algorithm code and how it interacts with the module and UI code. Each file is also describe in the general in Chapter 2 with templates.

5.2 Module Configuration File

As with the other examples, we need a module configuration file for this module. This file will need every field filled. Create a *SortMatrix.module* file in the *src/Modules/Factory/Config/* directory containing the following:

```
{
  "module": {
    "name": "SortMatrix",
    "namespace": "Math",
```

```
    "status": "new module.",
    "description": "sorts a matrix.",
    "header": "Modules/Math/SortMatrix.h"
  },
  "algorithm": {
    "name": "SortMatrixAlgo",
    "namespace": "Math",
    "header": "Core/Algorithms/Math/SortMatrixAlgo.h"
  },
  "UI": {
    "name": "SortMatrixDialog",
    "header": "Interface/Modules/Math/SortMatrixDialog.h"
  }
}
```

If you copy another module config file, make sure all the names are correct and that the namespace is set to Math.

5.3 Module Code

The next files needed for this module are the module code (*SortMatrix.cc*) and the header (*SortMatrix.h*) files. These files should be located in *src/Modules/Math/*.

The header (*SortMatrix.h*) file is not much different from the other two examples, as shown here:

```
#ifndef MODULES_MATH_SortMatrix_H
#define MODULES_MATH_SortMatrix_H

#include <Dataflow/Network/Module.h>
#include <Modules/Math/share.h>

namespace SCIRun {
namespace Modules {
namespace Math {

  class SCISHARE SortMatrix : public SCIRun::Dataflow::Networks::Module,
    public Has1InputPort<MatrixPortTag>,
    public Has1OutputPort<MatrixPortTag>
  {
  public:
    SortMatrix();
    virtual void execute();
    virtual void setStateDefaults();
    static const Dataflow::Networks::ModuleLookupInfo staticInfo_;

    INPUT_PORT(0, InputMatrix, Matrix);
    OUTPUT_PORT(0, OutputMatrix, Matrix);
```

```
  };
}}}
#endif
```

The import differences in this example is that the namespace and type of ports are different, and to leave the *setStateDefaults()* function without brackets so it can be set in the cc file.

The *SortMatrix.cc* file is a bit different from the previous examples. First, since most of the functionality of the module is in the algorithm files, this file can be very short, yet still have a powerful module. This file, along with the header, is mostly the code that pulls the code from the UI and algorithm together and interacts with SCIRun.

```
#include <Modules/Math/SortMatrix.h>
#include <Core/Datatypes/Matrix.h>
#include <Dataflow/Network/Module.h>
#include <Core/Algorithms/Math/SortMatrixAlgo.h>

using namespace SCIRun::Modules::Math;
using namespace SCIRun::Dataflow::Networks;
using namespace SCIRun::Core::Algorithms;
using namespace SCIRun::Core::Datatypes;

/// @class SortMatrix
/// @brief This module sorts the matrix entries
/// into ascending or descending order.

const ModuleLookupInfo SortMatrix::staticInfo_("SortMatrix", "Math",
    "SCIRun");

SortMatrix::SortMatrix() : Module(staticInfo_)
{
  INITIALIZE_PORT(InputMatrix);
  INITIALIZE_PORT(OutputMatrix);
}

void SortMatrix::setStateDefaults()
{
  setStateIntFromAlgo(Variables::Method);
}

void
SortMatrix::execute()
{
  auto input = getRequiredInput(InputMatrix);
  if (needToExecute())
  {
    setAlgoIntFromState(Variables::Method);
    auto output = algo().run_generic(withInputData((InputMatrix, input)));
    sendOutputFromAlgorithm(OutputMatrix, output);
```

```
  }
}
```

Notice that the algorithm file header is included. Also, most of the code in this file links the algorithm code directly to either the UI (with *setStateIntFromAlgo*, and *setAlgoIntFromState*) or SCIRun (*sendOutputFromAlgorithm*). This allows most of the code to reside in the algorithm code and makes SCIRun more modular.

5.4 Module UI Code

We will create a simple module UI for the SortMatrix module. The UI will consist of a toggle switch to choose between ascending and descending sorting. As before, we will need three files: *SortMatrixDialog.ui*, *SortMatrixDialog.h*, and *SortMatrixDialog.cc*, all need to be in *src/Interfac/Modules/Math/*. The process and code for this example is very similar to the previous example.

We will make the *SortMatrixDialog.ui* in the Qt editor as we did previously (Chapter 4). It may be easier to copy a previously created UI file and modify it that to create one from scratch. Copy the *src/Interface/Template/ModuleDesignerFile.ui* file and rename it. Delete the widgets in the UI and add two radial button widgets (drag from the 'Widget Box' window). Change the labels to 'ascending' and 'descending', and the names (in the 'Object Inpector' window) to 'ascendButton_' and 'descendButton_'. You will also need to make sure to change the name of the QDialog to 'SortMatrixDialog' (also in the 'Object Inpector'). The placement of the buttons and the size of the window and boxes can be adjusted in the 'Property Editor' window. The UI should look similar to Figure 5.1.

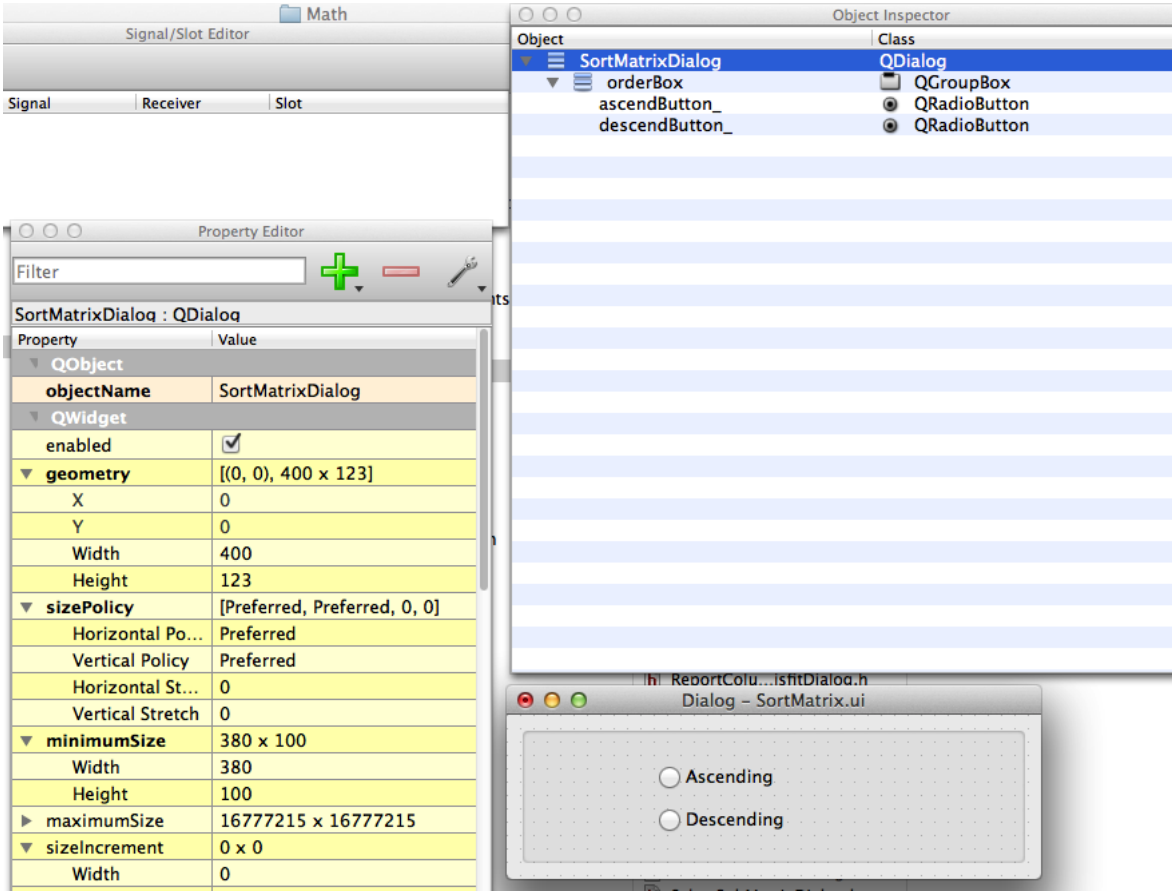

Figure 5.1. Module interface design file for the SortMatrix module as seen in the Qt editor.

The *SortMatrixDialog.h* is virtually identical to the header in the previous example (Chapter 4), except for the names, as shown here:

```
#ifndef INTERFACE_MODULES_MATH_SortMatrixDIALOG_H
#define INTERFACE_MODULES_MATH_SortMatrixDIALOG_H 1

#include "Interface/Modules/Math/ui_SortMatrixDialog.h"
#include <Interface/Modules/Base/ModuleDialogGeneric.h>
#include <Interface/Modules/Math/share.h>

namespace SCIRun {
namespace Gui {
class SCISHARE SortMatrixDialog : public ModuleDialogGeneric,
public Ui::SortMatrixDialog
{
Q_OBJECT

public:
SortMatrixDialog(const std::string& name,
```

```
SCIRun::Dataflow::Networks::ModuleStateHandle state,
QWidget* parent = 0);
};
}}
#endif
```

SortMatrixDialog.cc is also very similar to the dialog cc file in the previous example (Chapter 4):

```
#include <Interface/Modules/Math/SortMatrixDialog.h>
#include <Core/Algorithms/Base/AlgorithmVariableNames.h>
#include <QtGui>

using namespace SCIRun::Gui;
using namespace SCIRun::Dataflow::Networks;
using namespace SCIRun::Core::Algorithms;

SortMatrixDialog::SortMatrixDialog(const std::string& name,
                ModuleStateHandle state,
         QWidget* parent/* = 0*/)
         : ModuleDialogGeneric(state, parent)
{
         setupUi(this);
setWindowTitle(QString::fromStdString(name));
fixSize();

  addRadioButtonGroupManager({ ascendButton_, descendButton_ },
   Variables::Method);
}
```

The key difference in this file is the line that pulls the inputs from the UI using *addRadioButtonGroupManager*. This function pulls a set of radio buttons and creates a toggle switch to assign it to a variable. The order of the button names is important, as that determines the integer values of the variable when it is pressed, i.e., in the order shown, ascending will be 0 and descending will be 1.

5.5 Module Algorithm Code

The final step in creating this SortMatrix module is to create the module algorithm code. Two files are needed for the algorithm, *SortMatrixAlgo.h* and *SortMatrixAlgo.h*, which should be located in *src/Core/Algorithms/Math/*. The header file contains all the declarations for the functions needed in the algorithm and should be as follows:

```
#ifndef CORE_ALGORITHMS_MATH_SortMatrixALGO_H
#define CORE_ALGORITHMS_MATH_SortMatrixALGO_H

#include <Core/Datatypes/Matrix.h>
#include <Core/Datatypes/DenseMatrix.h>
```

```
#include <Core/Datatypes/DenseColumnMatrix.h>


#include <string>
#include <sstream>
#include <vector>
#include <algorithm>

#include <Core/Algorithms/Base/AlgorithmVariableNames.h>
#include <Core/Algorithms/Base/AlgorithmBase.h>
#include <Core/Algorithms/Math/share.h>

namespace SCIRun {
namespace Core {
namespace Algorithms {
namespace Math {

class SCISHARE SortMatrixAlgo : public AlgorithmBase
{

  public:
    SortMatrixAlgo();
    AlgorithmOutput run_generic(const AlgorithmInput& input) const;

    bool Sort(Datatypes::DenseMatrixHandle input,
        Datatypes::DenseMatrixHandle& output, int method) const;

    bool Quicksort(double* input, index_type lo, index_type hi) const;
    index_type Partition(double* input, index_type lo, index_type hi) const;
};
      }}}}
#endif
```

In this Algorithm, we have three functions to help implement the sorting algorithm, which will be called in the *run_generic* function.

The *SortMatrixAlgo.cc* file contains the computation code for he module. There are five functions in it. The first *SortMatrixAlgo()* sets up the defaults for the parameters that are set in the module UI. *run_generic* is the main function of the algorithm and the module. It mostly takes the inputs from the module and sends the data to use in the helper functions. *Sort* is the main helper function, which further processes the data into a format that be quickly sorted. *Quicksort* uses the output of *Partition* to spilt the matrix into smaller and smaller chunks and recursively calls itself until the matrix is sorted. *Partition* properly places the last entry in the matrix subset, with the values larger after and those lower before, and splits the matrix at that new place for the next iteration. The code should be similar to this:

```
#include <Core/Algorithms/Math/SortMatrixAlgo.h>
#include <Core/Datatypes/MatrixTypeConversions.h>
```

```
#include <Core/Math/MiscMath.h>

using namespace SCIRun;
using namespace SCIRun::Core::Datatypes;
using namespace SCIRun::Core::Algorithms;
using namespace SCIRun::Core::Algorithms::Math;

SortMatrixAlgo::SortMatrixAlgo()
{
  //set parameter defaults for UI
  addParameter(Variables::Method, 0);
}


AlgorithmOutput SortMatrixAlgo::run_generic(const AlgorithmInput& input) const
{
  auto input_matrix = input.get<Matrix>(Variables::InputMatrix);
  AlgorithmOutput output;

  //sparse support not fully implemented yet.
  if (!matrix_is::dense(input_matrix))
  {
    //TODO implement something with sparse
    error("SortMatrix: Currently only works with dense matrices");
    output[Variables::OutputMatrix] = 0;
    return output;
  }
  auto mat  = matrix_cast::as_dense (input_matrix);
  DenseMatrixHandle return_matrix;

  //pull parameter from UI
  auto method = get(Variables::Method).toInt();

  Sort(mat,return_matrix,method);
  output[Variables::OutputMatrix] = return_matrix;
  return output;
}


bool
SortMatrixAlgo::Sort(DenseMatrixHandle input, DenseMatrixHandle& output,
            int method) const
{
  if (!input)
  {
    error("SortAscending: no input matrix found");
    return false;
```

```
  }
  //get size of original matrix
  size_type nrows = input->nrows();
  size_type ncols = input->ncols();
  //copy original matrix for processing
  output.reset(new DenseMatrix(*input));
  //pointer to matrix data
  double *data = output->data();

  if (!output)
  {
    error("ApplyRowOperation: could not create output matrix");
    return false;
  }

  size_type n = nrows*ncols;
  //call the sorting functions
  Quicksort(data,0,n-1);

  if (method==1)
  {
    //if set to descending, reverse the order.
    output.reset(new DenseMatrix(output -> reverse()));
  }
  return true;
}

bool
SortMatrixAlgo::Quicksort(double* input, index_type lo, index_type hi) const
{
  //splits matrix based on Partition function
  index_type ind;
  if (lo<hi)
  {
    ind=Partition(input,lo,hi);
    Quicksort(input,lo,ind-1);
    Quicksort(input,ind+1,hi);
  }
  return true;
}

index_type
SortMatrixAlgo::Partition(double* input, index_type lo, index_type hi) const
{
  // places the last entry in its proper place in relation to the other
  // entries, ie, smaller values before and larger values after.
  index_type ind=lo;
```

```
  double pivot = input[hi];
  double tmp;
  for (index_type k=lo;k<hi;k++)
  {
    if (input[k]<=pivot)
    {
      tmp=input[ind];
      input[ind]=input[k];
      input[k]=tmp;
      ind+=1;
    }
  }
  tmp=input[ind];
  input[ind]=input[hi];
  input[hi]=tmp;
  return ind;
}
```

This algorithm uses the common inputs defined in the *AlgorithmVariableNames.h* file with the Variable namespace (Variable:Method, Variable:InputMatrix, and Variable:OutputMatrix. This allows for fewer declarations in the header file and is slightly easier to use. Also of note is that this algorithm is only implemented for dense matrices. This is because some support for sparse matrices hasn't been implemented at the time of writing this tutorial.

5.6 Building and Testing

5.6.1 Building

Once all the files have been created, SCIRun can be built with the new module. Be sure to added all eight files to the appropriate *CMakeList.txt* files in the *src/Modules/Math/*, *src/Interface/Modules/Math/*, and *src/Core/Algorithms/Math/*. Just add each of the filenames to the appropriate lists within the *CMakeList.txt* file, as shown in the previous examples (Sections 3.4, 4.1, & 4.2)

When creating new modules, it can be easier to added the code in a piecemeal fashion. This would entail get SCIRun to build with just the bare minimum of the algorithm code (only *run_generic* with no calls other functions) then to add the other functions in a step by step manner. This allows for easier debugging and a more systematic process to get the module working. Check Section 6.3 for common build problem.

5.6.2 Testing

To make sure that the new SortMatrix module works, create a network with CreateMatrix, SortMatrix, PrintMatrixIntoString, and PrintDatatype as shown in Figure 5.2. Create any matrix in CreateMatrix. In PrintMatrixIntoString, change the input to have the number of columns in your input matrix. In the 4x2 matrix that shown in Figure 5.2, the format string was: `%4.2g %4.2g %4.2g %4.2g \n`. Alternatively, the matrix entries can be printed as a

list with `%4.2g` (make sure there is a space at the beginning or end of the string). This network can be used to see the input and output of the SortMatrix module.

If this or another module is not behaving as expected, change the output of some functions and the modules to be some of the intermediate steps, or use `std::cout<< "message" <<std::endl;` to print values as the code is run. Unit Test can also find some bugs in module code.

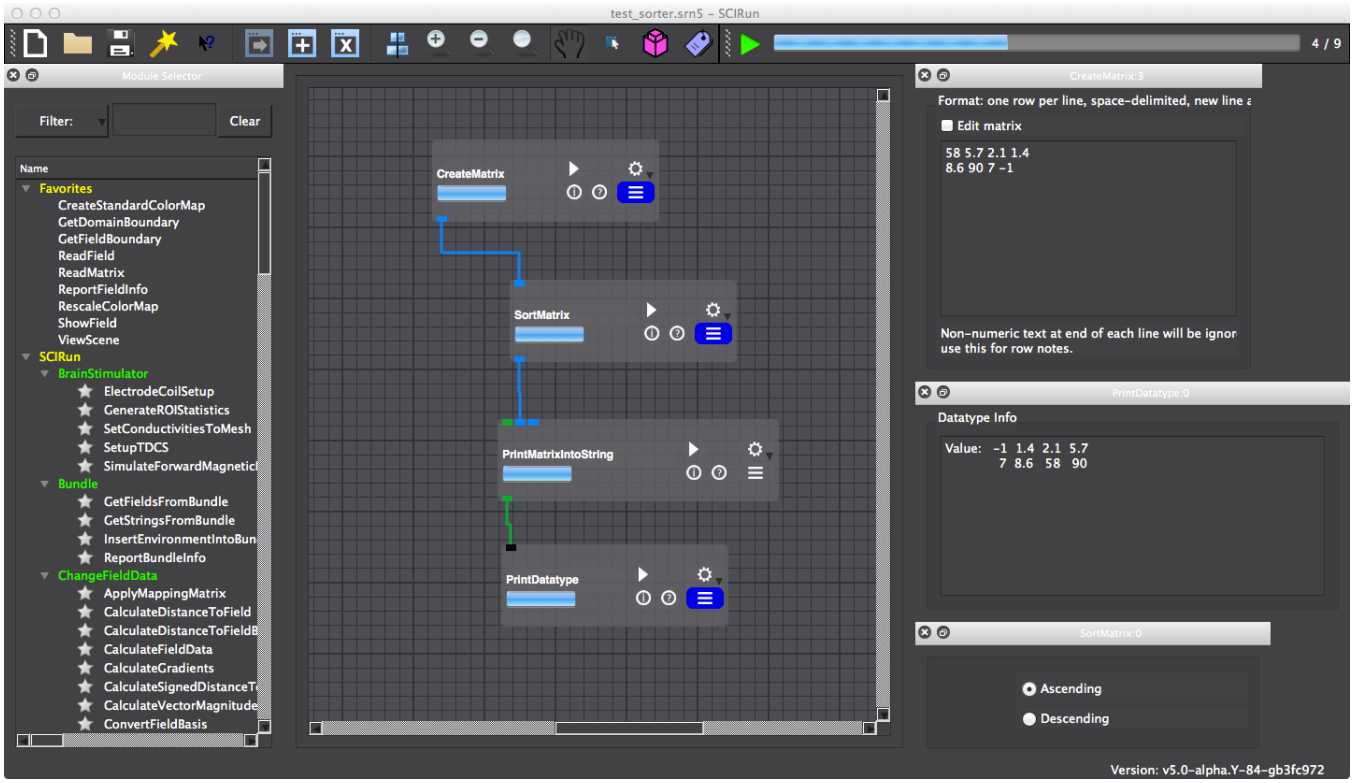

Figure 5.2. Network for running and testing the SortMatrix module.

Chapter 6

Converting Modules from SCIRun 4

This chapter will walk through the steps necessary to convert a module from SCIRun 4 to SCIRun 5. Converting a module is very similar to creating a new module, as expected. However, there are additional considerations that will be described in this chapter, including a list of common build errors and a list of common changes in code needed for the conversion.

Scope: Strategy - Common Function Changes - Common Build Errors

6.1 Strategy

The strategy for converting a module from SCIRun 4 to SCIRun 5 is very similar to the strategy of making a new module, which is to start with the basics, and add all the necessary parts piece by piece. This is demonstrated in our previous examples, especially in Chapters 3 & 4.

Another document outlining the steps to convert a module which may be helpful is in the source code: *src/Documentation/Manuals/ModuleConversionSteps.md*. It can also be found on github.

6.1.1 Set up Git Branch

In order for all the hard work of converting a module to be useful for all the users of SCIRun, be sure to use git and github for version control. Make sure that you have your own fork, sync to the SCIRun repository (Section 1.1.3). Create a new branch in your fork for each module conversion (Section 1.1.3) and be sure to commit your changes frequently.

6.1.2 Create a Module Configuration file

Create a module configuration file (Section 2.2) for the module in the *src/Modules/Factory/Config/* directory. I may be easiest to copy an existing file and change it. Be sure that all the names match the appropriate names of other files and fields in other files. It will be easier if the naming convention shown in Section 2.2 (name, nameDialog, nameAlgo, etc.) Leave fields blank with "N/A". For now, it can help to only fill in the first section `"module"`, as we will be adding the UI and the algorithm piece by piece.

6.1.3 Create a Module Header file

Virtually all modules in SCIRun 4 did not have header files for the module code, so one will need to be created for it. Copy the template header file (Section 2.3.1) or another module header file. The header file needs to be in the same location as the module .cc file (next Section). Change all the references to match the name of the module. Change the port names, number, and type. Make sure all functions used in the module code are declared in the header.

6.1.4 Get Module to Build Without Functionality

Getting the module code from SCIRun 4 working in SCIRun 5 can be challenging because there are a number of infrastructure and function changes. For this reason, it can beneficial to start with the basic code and build from there. Most of the SCIRun 4 module code is in the *src/Modules/Legacy/* directory, and can remain there. Make sure the header file is also in this directory.

Once the files are in the correct place, begin by removing all the SCIRun 4 specific code. If desired, the SCIRun 4 code can be commented out for now instead of deleted, but should be cleaned up before submitting a pull request. Remove any port header file includes, such as: `#include <Dataflow/Network/Ports/FieldPort.h>`, and The `DECLARE_MAKER` function. Remove the class declaration, as the header should already contain the essential declarations. We will add more as needed.

Make sure that the namespaces used are correct (at least two,Fields/Math/etc and Networks, are needed, Section 2.3.2). Make sure that the module header is included and the other headers included have the correct path. Change the module constructor (`@modulename@::@modulename@()` in the template example) to match the format shown in Section 2.3.2 with the correct port names. Add `staticInfo_` varible as in Section 2.3.2. Add a blank `setStateDefaults()` function:

```
void @ModuleName@::setStateDefaults()
{

}
```

or, if there is no module UI planned, add empty curly brackets to the header file declaration as discussed in Section 2.3.1.

The only code that will carry over is probably the `execute()` and other functions coded in the module, so everything else can be removed. Any helper functions will be should moved to an algorithm file later, but for now comment them out. Also comment out the content of the executable for now.

In this state, the three module files (config, header, and cc file) should be similar to the simplest example (Chapter 3), but the execute function will empty for now. Add the header and cc files the *CMakeList.txt* file in the directory they are in and try to build SCIRun. If there are build errors, check Section 6.3 for some ideas for how to correct them. The goal of building the code at this point it to make sure that the code which interacts with SCIRun is working properly before there are other mistakes in the code.

Once SCIRun builds, open it and find the new module. There will probably be a warning about creating UI because the code is expecting one and there isn't one. Make sure all the ports are there and are correctly names (hover the mouse over the port).

If there is no UI planned for this module, add a 'false' input to the module constructor (Sections 2.3.2 & 3.3). Otherwise, continue with the next step.

6.1.5 Add Module UI

Once the module is building without functionality, a module UI can be added. Begin by copying the UI files from another module with a similar interface, or the template files in *src/Interface/Modules/Template/*. Rename the three UI files (Section 2.4) and place them in the appropriate file in *src/Interface/Modules/*. The file names and subsequent function and item names in the UI code should be the same as the module with 'Dialog' appended to it.

In the module design file, use the Qt editor to create the UI that is needed by adding and removing widgets as needed (see Sections 4.2 & 5.4). Make sure that the name of the module and the name of the inputs are correctly named.

Next, modify the dialog header file so that the names of the module and dialog are corrected. There usually isn't anything extra needed with the dialog header. Similarly modify the module dialog cc file. Now add code to interpret the inputs from the widgets placed in the UI (Section 2.4). It may be helpful to look at other modules with similar UIs to determine which functions are needed (see Sections 4.2 & 5.4 for simple examples).

Now that the interface files are created, fill out the Interface section of the module configuration file. Make sure the names are consistent across files. Add all three files to *CMakeList.txt* in the directory that the files are in. Try to build SCIRun. If there are any errors, see Section 6.3 for ideas to resolve them. Once SCIRun is built, pull up the module and check the module UI. There will not be any functionality or defaults set, but the look should be correct. If the UI is correct, the UI code should be complete.

6.1.6 Add Module Algorithm Files

Module algorithm code isn't necessary if the module is simple. However using the algorithm class can be an easy way to work with the UI. Therefore, if the module that is being ported does not have algorithm code, consider adding it. If no algorithm code will be added to the module, skip this step.

Copy the algorithm code from SCIRUn 4 to the appropriate directory within *src/Core/Algorithms/* (some algorithms have been copied and not ported to SCIRun 5). If there was no algorithm code in SCIRun 4, copy a similar module algorithm code or the template code found in *src/Core/Algorithms/Template* and modify the names to match the module name with 'Algo' added to it. For now, comment out all the code within each of the functions, except any necessary return commands. Some of the functions may need name changes to match the general format in Section 5.5.

Now fill out the algorithm section of the module configuration file. Add the algorithm code and header files to the *CMakeList.txt* file in the directory that the algorithm code is in (or possibly the parent directory). Include the algorithm header in the module code. Build SCIRun. If there are no build errors, start adding the commented out code as described in the next section.

6.1.7 Add Module and Algorithm Functionality

If all the previous steps are completed, all the files needed for the module have been created and the infrastructure code is working. Next, the functional code of the module will need some work to get working. This step of converting the modules is the most specific to the module and could require the most experience to know the various functions that may have been changed in SCIRun 5. However, there may be some modules that require very little code changes in this step. Section 6.2 provides some examples of commonly used functions in SCIRun 4 and the replacement in SCIRun 5. The key to making the module code functional is similar to the other steps, i.e., expose one small piece at a time.

If there is a UI for the module, the state variables defaults need to be set. These need to be set in the `setStateDefaults` function. If there is algorithm code, these default definitions can be passed to the algorithm code, such as the example in Chapter 5. If there is no algorithm code, the state variables may need to be declared in the module code, as in UI example (Chapter 4). After setting the state defaults (and building the code), the module UI should display the defaults.

When converting the module code, if may be easier to start on the more standardized code, then work toward the more specific code, as we have been doing through this tutorial. For example, it may be easier to get the input and output calls working (as in Chapter 3), and then work on making the output what it needs to. Then, if there is algorithm code for the module, start by getting the algorithm call in the module code working with the algorithm code commented out, then work on the algorithm code. See Sections 6.2 & 6.3 for ideas to convert and fix specific functions and pieces of code.

6.1.8 Module Testing

Testing for a new module should occur intermittently while converting the code to make debugging easier, as we described in earlier steps. As you are trying to convert the module, test the module regularly to make sure that the output of module is as expected. Before finishing and submitting the module, test several types of inputs into make sure that the module behaves as intended. Since the module is converted from SCIRun 4, compare the outputs of the different versions.

In addition to making sure that the module works as expected, the module will need to be tested regularly for regression testing. A regression testing network and unit test code is needed for the module. The testing network should show different uses of the module if there are different function. For unit testing see Chapter 7 for information on creating unit test for the converted module.

6.1.9 Module Documentation

Make sure your module is documented properly in the git commits and code in addition to the module documentation as described in Chapter 8.

6.1.10 Github Pull Request

With the module fully completed, we can now submit it to be included in the main release of SCIRun using a pull request. Since there was a branch created for the new module, there should be regular commits as the module is ported. For the the pull request, make sure all

the changes have been committed to the branch meant for the new module. Now make sure that the branch is up to date with the latest changes in the main branch of SCIRun. To do this, sync your fork and merge the SCIRun master branch as shown in the Section 1.1.3. Make sure the module branch is merge with the master branch and make sure that your local changes are pushed to github. To make a pull request, there is usually a short cut on the main github page of the SCIRun or you can check out the github help page about it. Add some comments to the developers to know what to look for when reviewing the code. If you have changes to make, either that you noticed or requested by the developer, just commit it to the same branch and push it github and the pull request will track the changes until it is merged.

6.2 Common Function Changes

```
if (input.get_rep() == 0)
```

6.3 Common Build Errors

```
ConvertMeshToPointCloudDialog.h:32:10: fatal error:
      'Interface/Modules/Field/ui_ConvertMeshToPointCloudDialog.h' file not found
```

Chapter 7

Creating Unit Tests

You will at least need a testing network.

Chapter 8

Documenting the New Module

You should totally document the modules you add.