C++ GUI Programming with Qt

Documentation

By Mingwei Hu

Table of Content

ABSTRACT

This documentation presents a common method of using C++ programming language to design a user interface in Qt creator. The building process of this interface includes creating widgets on the designer interface and coding in the editor interface. Widgets, which include input dialogs, checkboxes, and pushbuttons, enable users to run the program in an efficient way without the need to type command in the linux terminal. In the designer interface, widgets are dragged and dropped onto the main window; this process simultaneously declares corresponding QObjects on the user interface. The widgets built on the main window act as a “bridge” through which users can change any member variables declared inside the C++ classes; this process is aided by a “Signal and Slot” mechanism which generates signals that tell the program to implement the changes the user wants. It turns out that this user interface makes it more convenient to run the SIF2XMGRACE program; however, there are also some potential improvements that can be further implemented.

Background Information

Qt: Qt is a cross-platform application framework developed by Digia. It is widely used for developing application software with a graphical user interface (GUI).

Qt Creator : Qt Creator is a cross-platform C++ integrated development environment which is part of the Qt SDK. It includes a visual debugger and an integrated GUI layout and forms designer. The editor's features include syntax highlighting and auto-completion. Qt Creator uses the C++ compiler from the GNU Compiler Collection on Linux and FreeBSD.

Qt creator can be downloaded from http://qt-project.org/downloads

Qt tutorial can be found at http://qt-project.org/doc/qt-5.0/qtdoc/qtexamplesandtutorials.html

A Qt Development Environment

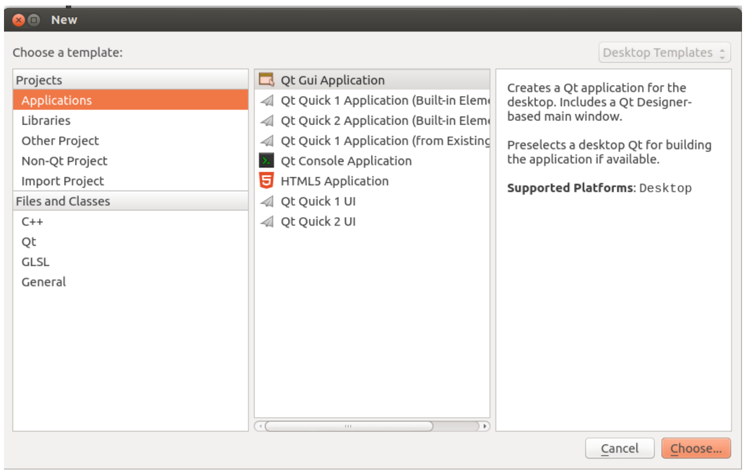
Most of the time we use Qt creator to do the following things: 1) Creating Qt Projects ; 2) Building and Running Applications in Qt Creator; and 3) Debugging applications. All of these tasks can be done conveniently either within the editor environment or designer environment.

Qt project management

Qt provides a user-friendly environment for managing projects. First, creating a Qt project enables user to group files together and add customized build steps. It takes several steps to build a Qt application:

Step#1:

Open Qt creator by clicking the button; click File→New File or Project and we will be directed to the following dialog box:



Creates a Qt application for the desktop. Includes a Qt Designer-based main window.

Preselects a desktop Qt for building the application if available.

Supported Platforms: `Desktop`

Where in the upper-left column we can then select the type of project we want to create. Typically for creating a simple user interface the "Application" is the right option. Then you need to choose a typical type of application from the middle column. Specifically, a Qt GUI Application uses Qt Designer forms to design a Qt widget based user interface for the desktop, which is the case for my project. And a Mobile Qt Application uses Qt Designer forms to design a Qt widget based user interface for mobile devices (for more information see http://doc.qt.digia.com/qtcreator/creator-project-creating.html).

Step#2

After selecting the application type, click next and name your project and then choose the location to place your project. Click next again and you will be asked to select a Kit, for this part we use its default setting (check all the three options). Click next.

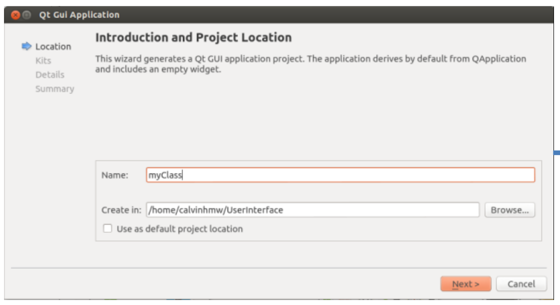
Step#3

Name your class and specify its base class (the class from which your class inherits public members). The three available base classes are: QMainWindow, QWidget, and QDialog. They are not at the same base class level; in fact, they have a parent-child relation (similar to derived-based relation in C++) in which QWidget inherits QObject which is the base class of all Qt objects. But QWidgets is the base class of QMainWindow and QDialog. The programmer chooses the base class based on what functionality he/she wants the user interface to perform. For example, one may choose QMainWindow as the base class if he/she wants to display some menu or show some animation. In our case, we are asking users for their input data so the best choice is to select QDialog as the base class.

Step#4

Choose a version control system to which you can commit your code. Leave "none" if you don't have any version control system available.

Flowchart for creating a project



Introduction and Project Location

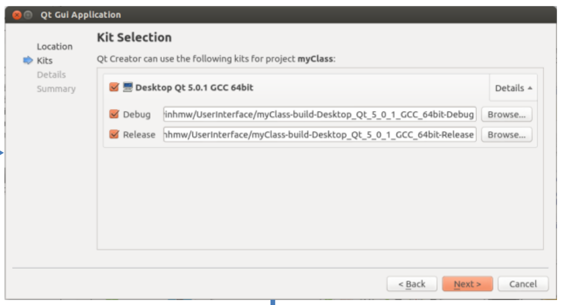
This wizard generates a Qt GUI application project. The application derives by default from QApplication and includes an empty widget.

Name: myClass

Create in: /home/calvinhmw/UserInterface

Use as default project location

<u>Select file location</u>



Kit Selection

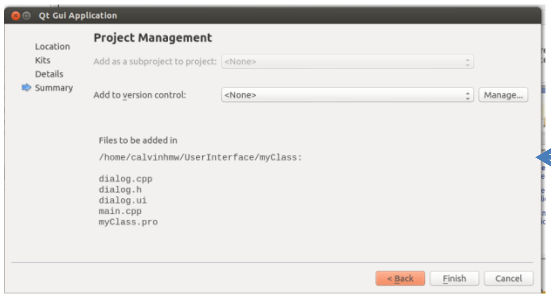
Qt Creator can use the following kits for project **myClass**:

✓ Desktop Qt 5.0.1 GCC 64bit

✓ Debug inhmw/UserInterface/myClass-build-Desktop_Qt_5_0_1_GCC_64bit-Debug

✓ Release hmw/UserInterface/myClass-build-Desktop_Qt_5_0_1_GCC_64bit-Release

<u>Select a kit</u>



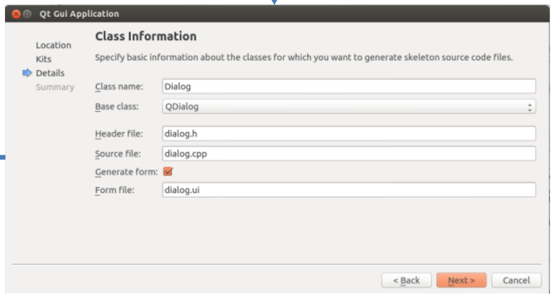
Project Management

Add as a subproject to project: <None>

Add to version control: <None>

```
Files to be added in

/home/calvinhmw/UserInterface/myClass:

dialog.cpp
dialog.h
dialog.ui
main.cpp
myClass.pro
```

<u>Select version control system</u>



Class Information

Specify basic information about the classes for which you want to generate skeleton source code files.

Class name: Dialog

Base class: QDialog

Header file: dialog.h

Source file: dialog.cpp

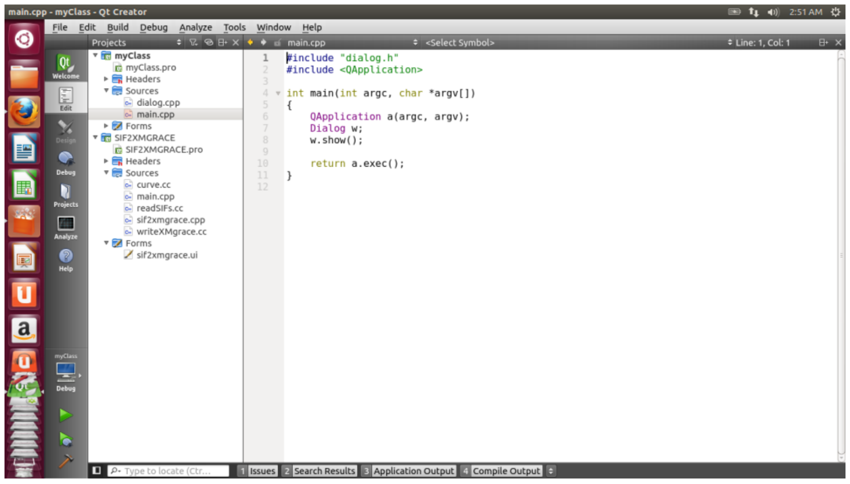
Generate form: ✓

Form file: dialog.ui

<u>Select base class</u>

<u>Editor and Designer interface</u>

After creating you project you will be directed to the editor interface. And the first thing you see is your main fucntion that has already been writen for you:



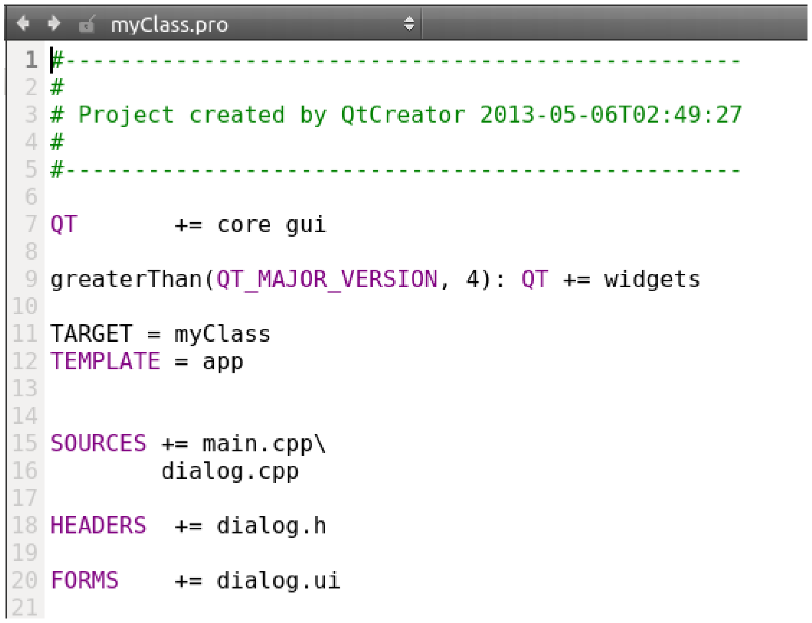
```
myClass
    myClass.pro
    Headers
    Sources
        dialog.cpp
        main.cpp
    Forms
SIF2XMGRACE
    SIF2XMGRACE.pro
    Headers
    Sources
        curve.cc
        main.cpp
        readSIFs.cc
        sif2xmgrace.cpp
        writeXMgrace.cc
    Forms
        sif2xmgrace.ui
```

```
#include "dialog.h"
#include <QApplication>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    Dialog w;
    w.show();

    return a.exec();
}
```

Main function

Where “a” is an object of type “QApplication” which runs the whole program when getting executed. A Dialog object “w” is created then and w.show() simply displays this dialog box when the program runs.

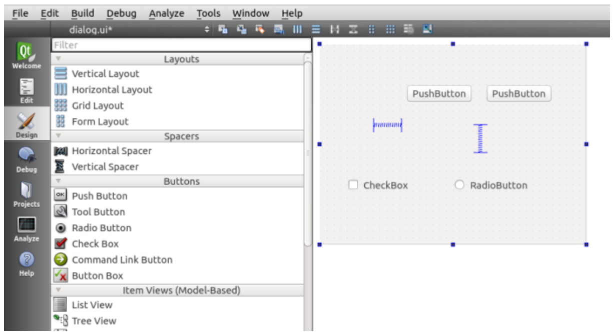
In the left column you can see the files you have. The header files and source files are grouped separately and you can add any other existing files (either .h or .cpp) to you project by right clicking your class name and choose “Add Existing Files”. Notice that there is also a project file (.pro) written for you:



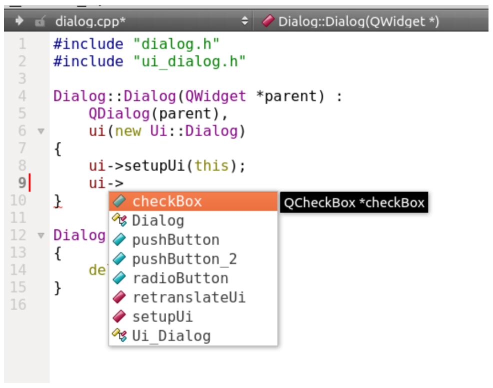
```
#-------------------------------------------------
#
# Project created by QtCreator 2013-05-06T02:49:27
#
#-------------------------------------------------

QT       += core gui

greaterThan(QT_MAJOR_VERSION, 4): QT += widgets

TARGET = myClass
TEMPLATE = app


SOURCES += main.cpp\
        dialog.cpp

HEADERS  += dialog.h

FORMS    += dialog.ui

```

The project file describes all other files where “+=core gui” adds gui library, “SOURCES+=” adds source files, “HEADER+=” adds header files, and “CONFIG+=” linkd relevant libraries. The project file is necessary for QMake to generate Makefile. Note that under few circumstances do users need to write the project file unless they want to manuelly add some external libraries; the project file is automatically generated is able to update with any files you added to the project.

Under “form”, click “dialog.ui” and you will be directed to the design interface. This is the place where you want to start creating your application. The left column has a list QWidgets objects that you can simply drag and drop them on to the dialog box. One best thing is that no matter how many QWidget objects you create the editor interface will keep updating the object list; that is, you don’t need to declare you object after you create them on the designer interface; they are automatically “known” by the user interface:



Randomly drop some widgets on the dialog



```
#include "dialog.h"
#include "ui_dialog.h"

Dialog::Dialog(QWidget *parent) :
    QDialog(parent),
    ui(new Ui::Dialog)
{
    ui->setupUi(this);
    ui->
}

Dialog
{
    de
}
```

Widgets are known by the user interface

Here please note that the QDialog object, the checkbox object, and pushButton object, etc are all implicitly declared inside the User Interface class (here we call it “ui”).

Also notice that you can indirectly modify the dialog object by writing some code inside the constructor (Dialog :: Dialog(QWidget * parent) : QDialog(parent),ui(new UI::Dialog) {……}). For example, you can call the setGeometry() function to adjust the position and size of a widget. This line:

ui->checkbox->setGeometry(200,200,50,10);

sets the position of the checkbox to be (200,200) and the size to be (50,50).

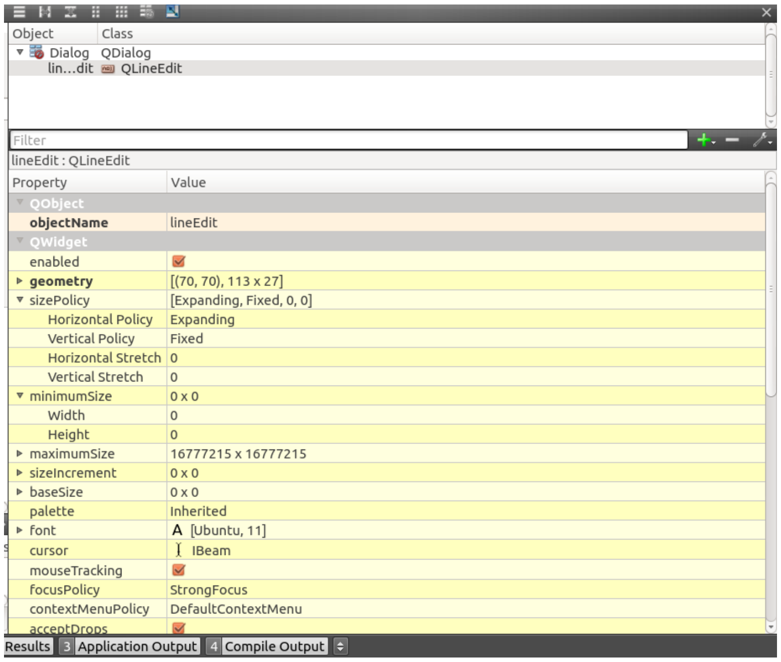
And this line:

ui->checkbox->setText(“title”)

changes the text of the checkbox to “title”.

Also, often we want to read the user’s input from the interface, and we need to create a QLineEdit object for achieving this. A lineEdit object can be found in the “input widget” list. Simply drag it and drop it onto the main dialog; and then after the user inputs a value (can be a sentence), you can extract the user’s input by simply calling “lineEdit->text()” (suppose lineEdit is the name of your QLineEdit object).

Another important thing is that after creating widgets on the designer interface you may want to change its name, position, or other properties. A recognizable name allows you to quickly find the widget you want to modify in the editor interface. This process can be easily done in the right side column in the designer interface:


Object	Class
Dialog	QDialog
lin...dit	QLineEdit

lineEdit : QLineEdit

Property	Value
QObject	
objectName	lineEdit
QWidget	
enabled	✓
geometry	[(70, 70), 113 x 27]
sizePolicy	[Expanding, Fixed, 0, 0]
Horizontal Policy	Expanding
Vertical Policy	Fixed
Horizontal Stretch	0
Vertical Stretch	0
minimumSize	0 x 0
Width	0
Height	0
maximumSize	16777215 x 16777215
sizeIncrement	0 x 0
baseSize	0 x 0
palette	Inherited
font	A [Ubuntu, 11]
cursor	IBeam
mouseTracking	✓
focusPolicy	StrongFocus
contextMenuPolicy	DefaultContextMenu
acceptDrops	✓

This column contains all the common properties of an object you create. They include the object name, object size, and the front of the text (for QLineEdit object). You can easily modify them by changing their value under the value column.

User Interface Design for SIF2XMGRACE program

<u>Background information for SIF2XMGRACE</u>

To be able to design an interface for plotting XMGRACE data, first we have to have a deep understanding of the original C++ code as well. The existing C++ codes include the following files:

`curve.h` and `curve.cc`: store curve data and curve properties (radius and extraction method)

`readSIFs.h` and `readSIFs.cc`: read a .sif or .crp data file; extract and build different types of curves (propagation curves, raw global curves, etc.)

`writeXMgrace.h` and `writeXMgrace.cc`: take the curves data (constructed by readSIFs) and plot them in XMGRACE.

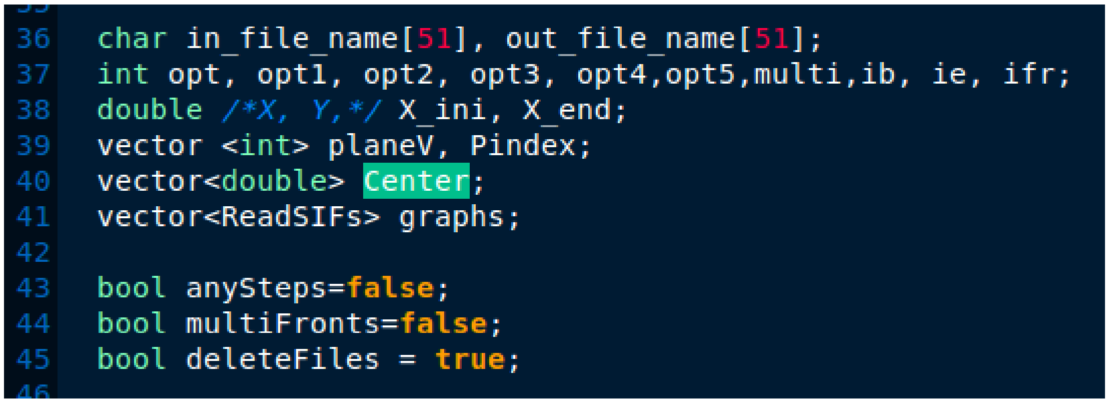
`sif2grace.cc`: contains the main function for the whole program; gets user's input and passes the input to writeXMgrace.cc.

This program originally runs on linux terminal and it asks user's input options over and over again before plotting data. The user cannot proceed to the next question until he/he answers the current one; and if the user realizes that he/she input a wrong value previously he/she has to rerun the program in order to change the value. These make the plotting process quite tedious and inconvenient. Therefore, our desired interface is one that enables users to input different options at the same time and to change an input value in a quicker way.

<u>Private member variables of SIF2XMGRACE class</u>

It can be clearly seen that the sif2grace.cc acts as an interface between users and the program. Therefore, our interface should implement the functionality of sif2grace.cc, but in a different way. It should be able to collect all the user inputs and process them all at once and give warning messages for invalid inputs.

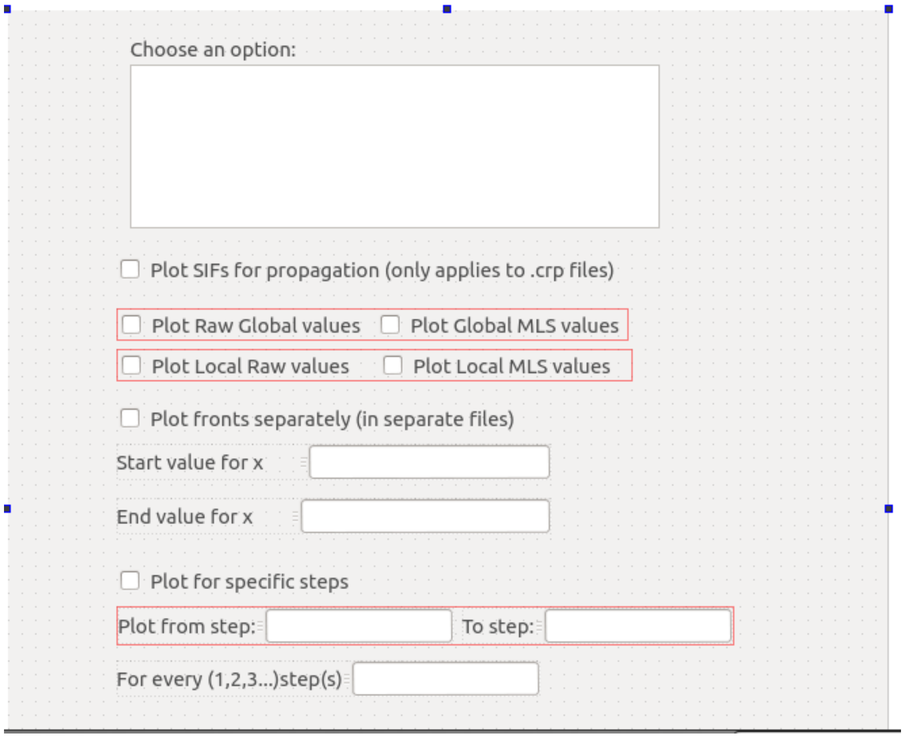
In order to achieve these functionalities, we have to first understand how the program stores the users' input options. The sif2grace.cc defines several local variables in the main function as follows:



```
char in_file_name[51], out_file_name[51];
int opt, opt1, opt2, opt3, opt4,opt5,multi,ib, ie, ifr;
double /*X, Y,*/ X_ini, X_end;
vector <int> planeV, Pindex;
vector<double> Center;
vector<ReadSIFs> graphs;

bool anySteps=false;
bool multiFronts=false;
bool deleteFiles = true;
```

where *in_file_name* and *out_file_name* respectively stores the name of input file and the name of output file respectively. *Opt1—opt5* are integers and they record which plot option the user selects. multi is an integer and it records the number of front a crack surface has. *MultiFronts* is a Boolean variable and it stores whether the user wants to plot fronts separately (for data files having more than one fronts). *Ib* is the beginning step and *ie* is the end step and the *ifr* is how many other steps are there between two selected steps. *X_ini* is the starting value of x coordinate and *Y_ini* is the starting value of y coordinate. *Pindex* is a vector that stores the step values the user wants to plot. *anySteps* records whether the user wants to plot specific steps and *deleteFiles* record whether to delete output file after plotting. In addition, in sif2grace.cc there are also some Boolean variables specifying whether to plot a certain type of data.

These local variables are then initialized by the user and passed to initialize the outFile of the WriteXMgrace class. It should be easy to imagine that, in the case of designing a user interface, we have to enter some values into a dialog box and let this dialog pass these values to the program. Therefore, we create a Qt Gui application and choose the base class to be QDialog. We name our QDialog class to be “SIF2XMGRACE”. In the designer interface a “blank” dialog is created (meanwhile in main.cpp an object of SIF2XMGRACE is declared). Next, we can put some widgets on the dialog according to what variables we have in sif2grace.cc:



As can be seen above, these widgets include a listWidget (the first big box) which is used for displaying the plot options (plot KI,KII,KIII,G or all), and several checkboxes. These checkboxes corresponds to the Boolean variables of whether to plot a certain type of values, such as global values and local values. It also includes some lineEdits which ask for user input.

After creating the interface in the designer interface, the next thing we need is to go to the editor interface and extract the user input so that we can initialize relevant variables (the same variables declared in if2grace.cc).

However, a problem is that where should we declare these variables for the case of designing an interface? The original sif2grace.cc declares these variables inside the main function, but in the editor interface a main.cpp already exists. The main.cpp creates a SIF2XMGRACE object, which is completely different from what the main function in sif2grace.cc is doing. Therefore, we have to declare these variables else where. One good way to do this is let these variables to be private member variables of the SIF2XMGRACE class. In this way we force these variables to be some properties of the dialog box. And after users input some values on the interface we should be able to use these values to initialize these member variables and change the SIF2XMGRACE object. Therefore, we EXPLICITLY put these member variables in sif2xmgrace.h as follow:

```
private:
    Ui::SIF2XMGRACE *ui;
    string in_file_name;
    string out_file_name;
    string input;
    int opt, multi , ib, ie, ifr;

    bool anySteps;
    bool multiFronts;

    bool isPropFile;
    bool PROP;
    bool PGL;
    bool PGLMLS;
    bool PL;
    bool PLMLS;
    bool deleteFiles;
    bool plotAgain;


    double  X_ini, X_end;
    vector <int> planeV, Pindex;
    vector<double> Center;
```

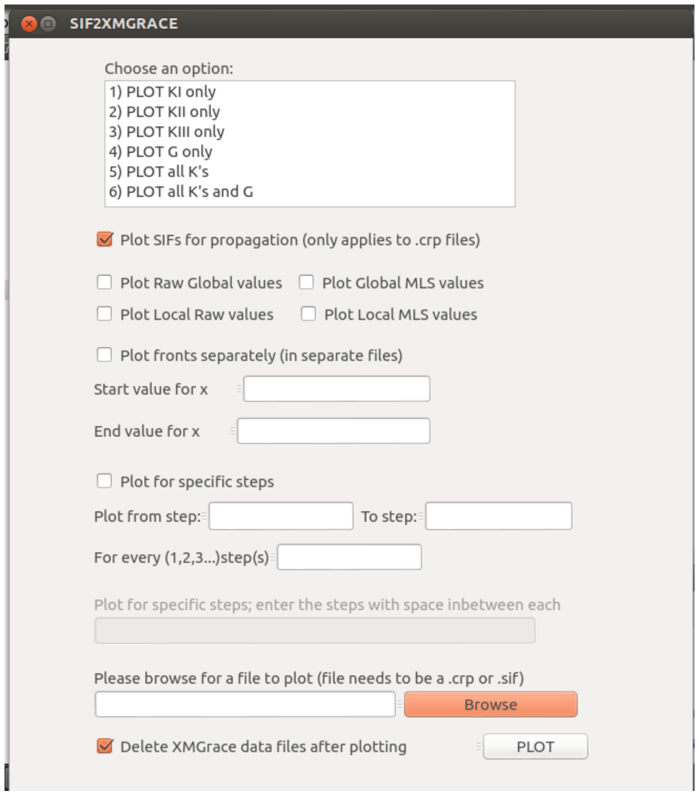
Now, since we have our member variables declared, the next thing it to initialize them. Clearly this is done in the constructor in sif2xmgrace.cpp. In addition, since we have created some widgets on the interface, these widgets belong to the user interface (ui) and they should also be initialized in SIF2XMGRACE's constructor. Therefore, the constructor is written as:

```
SIF2XMGRACE::SIF2XMGRACE(QWidget *parent) :
                              QDialog(parent),
                              (new Ui::SIF2XMGRACE)
{
         ui->setupUi(this);

         anySteps=false;
         multiFronts=false;
         isPropFile=false;

         PROP=false;
         PGL=false;
         PGLMLS=false;
         PL=false;
         PLMLS=false;
         deleteFiles=false;
         plotAgain=false;

         X_ini=0;
         X_end=0;

        ui->listWidget->addItem("1) PLOT KI only");
        ui->listWidget->addItem("2) PLOT KII only");
        ui->listWidget->addItem("3) PLOT KIII only");
        ui->listWidget->addItem("4) PLOT G only");
        ui->listWidget->addItem("5) PLOT all K's");
        ui->listWidget->addItem("6) PLOT all K's and G ");
        ui->checkBox_PROP->setChecked(true);

         ui->checkBoxAnySteps->setChecked(false);
         ui->lineEdit_specSteps->setEnabled(false);
         ui->label_anySteps->setEnabled(false);
        ui->checkBoxDelete->setChecked(true);
        ui->checkBoxDelete->setCheckable(true);

        connect(ui->checkBoxAnySteps,SIGNAL(stateChanged(int) ), this , SLOT(newSteps()) );

}
```

Note that the lines "`ui->listWidget->addItem("XXX");`" adds an item to the listWidget object and set its title to "XXX". The lines `ui->checkBoxXXX->setChecked(bool)` set the correspoding checkbox to be checked or unchecked. Note that the line "`ui->checkBoxDelete->setChecked(true)`" sets the "Delete file" checkbox to be checked since deleting files after plotting is a common option.

The connect() call at the end of the constructor employs a signal-slot-machinism which will be discussed later.

Now, if we run the program, an interface will pop up:


SIF2XMGRACE

Choose an option:

1) PLOT KI only
2) PLOT KII only
3) PLOT KIII only
4) PLOT G only
5) PLOT all K's
6) PLOT all K's and G

[✓] Plot SIFs for propagation (only applies to .crp files)

[ ] Plot Raw Global values [ ] Plot Global MLS values

[ ] Plot Local Raw values [ ] Plot Local MLS values

[ ] Plot fronts separately (in separate files)

Start value for x

End value for x

[ ] Plot for specific steps

Plot from step: To step:

For every (1,2,3...)step(s)

Plot for specific steps; enter the steps with space inbetween each

Please browse for a file to plot (file needs to be a .crp or .sif)

Browse

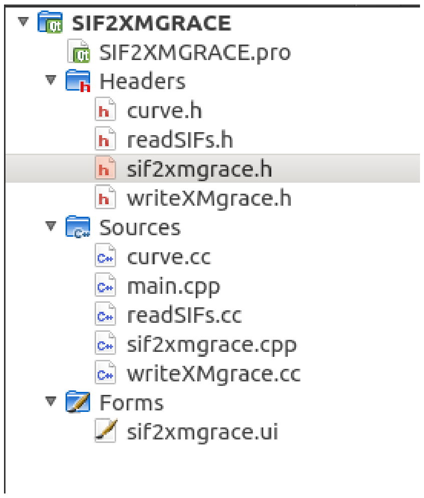
[✓] Delete XMGrace data files after plotting

PLOT

<u>Adding existing files</u>

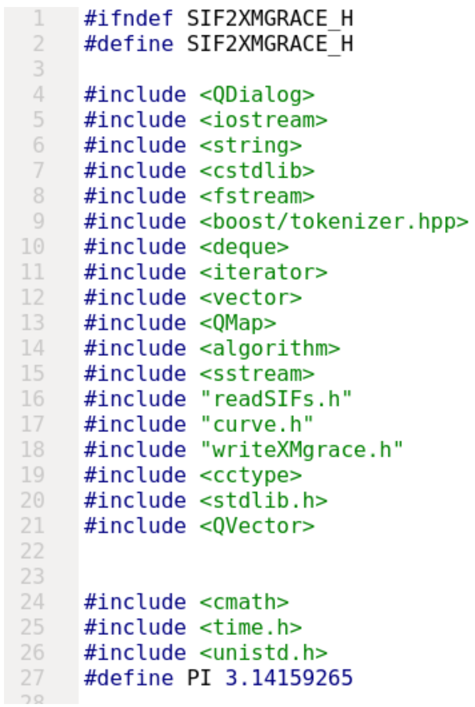
Now if you clicked one of the buttons on the interface above nothing happens because we haven't built a "bridge" between this interface and the C++ code; that is, we are not able to extract the user's input to modify corresponding member variables in sif2xmgrace.h, and we have imported the plotting functionality of the origninal program.

Therefore, the next thing to do is adding files that are responsible for reading and plotting data files to the program. Add the files curve.h, curve.cc, writeXMgrace.h, writeXMgrace.cc, readSIFs.h, and readSIFs.cc so that the the files are grouped like below:



```
SIF2XMGRACE
    SIF2XMGRACE.pro
    Headers
        curve.h
        readSIFs.h
        sif2xmgrace.h
        writeXMgrace.h
    Sources
        curve.cc
        main.cpp
        readSIFs.cc
        sif2xmgrace.cpp
        writeXMgrace.cc
    Forms
        sif2xmgrace.ui
```

Next we need to update the sif2xmgrace.h. Add necessary "#include <>" statements to include useful libraries and other head files:



```
#ifndef SIF2XMGRACE_H
#define SIF2XMGRACE_H

#include <QDialog>
#include <iostream>
#include <string>
#include <cstdlib>
#include <fstream>
#include <boost/tokenizer.hpp>
#include <deque>
#include <iterator>
#include <vector>
#include <QMap>
#include <algorithm>
#include <sstream>
#include "readSIFs.h"
#include "curve.h"
#include "writeXMgrace.h"
#include <cctype>
#include <stdlib.h>
#include <QVector>


#include <cmath>
#include <time.h>
#include <unistd.h>
#define PI 3.14159265
```

<u>Signal and Slot</u>

The code itself now has the functionality to plot XMGRACE data. However, we need somehow to “trigger” the plot command so that the users input can be passed into the program and the readSIFs and writeSMgrace program will then run. This process can be done using a “Signal & Slot” mechanism. It is used for communication between objects. Signals are emitted by objects when they change their state in a way that may be interesting to other objects. Slots are used for receiving signals; they are often in the form of member functions.

Objects emitting signals are independent from slots receiving it. An object does not know which slot receives its signal and a slot does not know which signal is connected to it. Rather, we sometimes use a “connect” command to “trigger” the function call in the slot when a signal is detected. The connect function has the following form:

```
connect(object#1,signal,object#2,slot)
```

where object#1 emits a signal and the slot belonging to object#2 receives the signal and the function defined in the object#2’s slot is called. In order to use connect command, we have to declare the “slot functions” inside the class header file like:

```
private slots:
      function#1;
      function#2;
      ……
```

And then we need to write the function implementation in the class cpp file for function declared in private slots (or public slots).
Another way to implement the signal and slot mechanism, especially for bushbutton object, is using the "Go To Slot" command on the designer interface. In this way, the slot functions are automatically declared. We don't even need to write the "connect" command and all we need to do is implement the slot functions.

For more information about Signal & Slot, please see: http://qt-project.org/doc/qt-4.8/signalsandslots.html

Back to our user interface, notice we have two pushButtons. By clicking the "Browse" pushButton we want to be directed to a file directory so that we can select a data file to plot. And by clicking the "PLOT" button we want the program to get executed. Right click the "Browse" button in the designer interface; click "**Go to slot**" and select **clicked()**. Hit **OK** and you will end up inside a function implementation called "`void SIF2XMGRACE::on_pushButton_clicked()`". This is the slot function belonging to the object of SIF2XMGRACE.

Now begin implementing this slot function. The way it should work is that we will be directed to a file location and the name of the file we choose will be displayed in the lineEdit box on the left of the Browse button. Therefore, we need to have another function directs user to a file directory and change the text of the lineEditName to be the name of the file we choose. Therefore, we implement a function called "getPath()" and call the QFileDialog's existing function "getOpenFileName()" (don't forget to declare the getPath() in sif2xmgrace.h !):

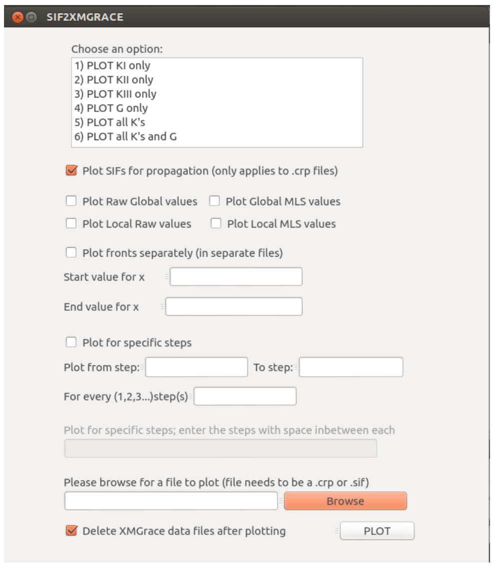
```
void SIF2XMGRACE::getPath(){
        QString path;
        path = QFileDialog::getOpenFileName( this,"Choose a file to open",
QString::null, QString::null);
        ui->lineEditName->setText(path);
}
```

And then we can call this function inside the `on_pushButton_clicked()` function:

```
 void SIF2XMGRACE::on_pushButton_clicked()
{
      getPath();
      QString fileName=ui->lineEditName->text();
      if(fileName[fileName.size()-1]=='p')
                  isPropFile=true;
            in_file_name=fileName.toStdString();

      warning();
}
```

Note: warning() is a function written for handling the case where a non-propagation file is selected (see the source code for detail).

Now the “Browse” button can work the way we want:


SIF2XMGRACE

Choose an option:
1) PLOT KI only
2) PLOT KII only
3) PLOT KIII only
4) PLOT G only
5) PLOT all K's
6) PLOT all K's and G

[✓] Plot SIFs for propagation (only applies to .crp files)

[ ] Plot Raw Global values [ ] Plot Global MLS values
[ ] Plot Local Raw values [ ] Plot Local MLS values

[ ] Plot fronts separately (in separate files)
Start value for x
End value for x

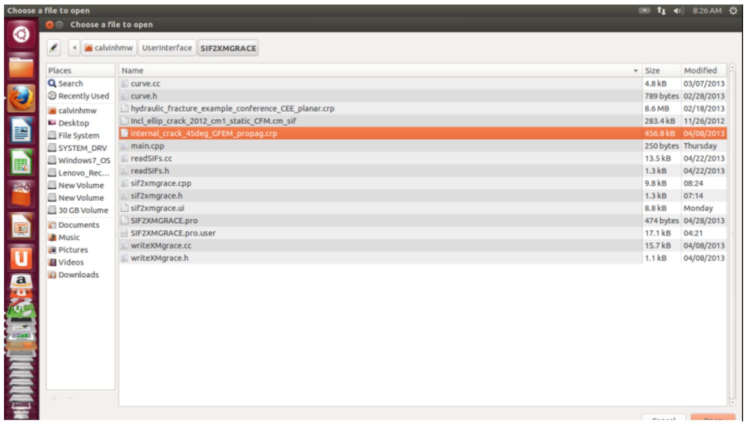
[ ] Plot for specific steps
Plot from step: To step:
For every (1,2,3...)step(s)

Plot for specific steps; enter the steps with space inbetween each

Please browse for a file to plot (file needs to be a .crp or .sif)

[✓] Delete XMGrace data files after plotting

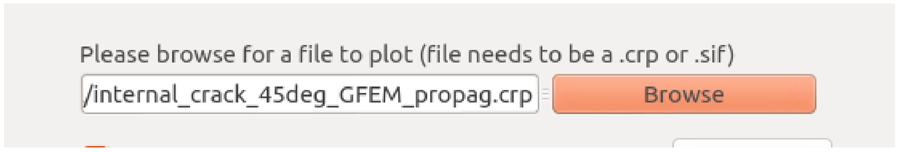
Click “Browse”


Choose a file to open

calvinhmw / UserInterface / SIF2XMGRACE

Places	Name	Size	Modified
Search	curve.cc	4.8 kB	03/07/2013
Recently Used	curve.h	789 bytes	02/28/2013
calvinhmw	hydraulic_fracture_example_conference_CEE_planar.crp	8.6 MB	02/18/2013
Desktop	Incl_ellip_crack_2012_cm1_static_CFM.cm_sif	283.4 kB	11/26/2012
File System	internal_crack_45deg_GFEM_propag.crp	456.8 kB	04/08/2013
SYSTEM_DRV	main.cpp	250 bytes	Thursday
Windows7_OS	readSIFs.cc	13.5 kB	04/22/2013
Lenovo_Rec...	readSIFs.h	1.3 kB	04/22/2013
New Volume	sif2xmgrace.cpp	9.8 kB	08:24
New Volume	sif2xmgrace.h	1.3 kB	07:14
30 GB Volume	sif2xmgrace.ui	8.8 kB	Monday
Documents	SIF2XMGRACE.pro	474 bytes	04/28/2013
Music	SIF2XMGRACE.pro.user	17.1 kB	04:21
Pictures	writeXMgrace.cc	15.7 kB	04/08/2013
Videos	writeXMgrace.h	1.1 kB	04/08/2013
Downloads			

Select a data file


Please browse for a file to plot (file needs to be a .crp or .sif)

/internal_crack_45deg_GFEM_propag.crp

Name of the data file is copy into the lineEdit box

Finishing implementing Signal & Slot for the browse, we need to do the same thing for the PLOT button. When we click “PLOT”, the program has to immediately initialize or change the value of relevant member variables according to user’s selection. For example, if the user checks the option “Plot Raw Global Values”, then the Boolean member variable PGL should be set to “true”. Therefore, inside the on_pushButton_plot_clicked() function, we can write several “if—else” statements to check user input and then change the corresponding member variable.

The whole slot function “on_pushButton_plot_clicked()” is written as following:

```
void SIF2XMGRACE::on_pushButton_plot_clicked()
{

          if(!isPropFile) {
                 ui->checkBoxMulti->setChecked(false);
         }

         QChar option=(ui->listWidget->currentItem()->text())[0];

          if(option=='1')opt=1;
         else if(option=='2')opt=2;
         else if(option=='3')opt=3;
         else if(option=='4')opt=4;
         else if(option=='5')opt=5;
         else opt=6;

          if(ui->checkBoxMulti->isChecked()) multiFronts=true;

         if (ui->checkBox_PROP->isChecked()) PROP=true;
         if (ui->checkBox_PGL->isChecked()) PGL=true;
        if (ui->checkBox_PGLMLS->isChecked()) PGLMLS=true;
         if (ui->checkBox_PL->isChecked()) PL=true;
         if (ui->checkBox_PLMLS->isChecked()) PLMLS=true;

           X_ini=ui->lineEdit_startX->text().toDouble();
          X_end=ui->lineEdit_EndX->text().toDouble();

          if(!ui->checkBoxAnySteps->isChecked()) {
                        anySteps=false;
                  ib=ui->lineEdit_fromStep->text().toInt();
                ie=ui->lineEdit_toStep->text().toInt();
                 ifr=ui->lineEditSteps->text().toInt();

                 for(int pos=ib;pos<=ie;pos=pos+ifr)
                         Pindex.push_back(pos);
     }

         else {
                        anySteps=true;
                  input=ui->lineEdit_specSteps->text().toStdString();
                 for(int i=0;i<(int)input.size();i++){

//************THIS PART DEALS WITH THE CASE FOR ENTERING SPECIFIC STEPS****************//
                if (isdigit(input[i])){
                      string specificSteps="";

                    specificSteps.push_back(input[i]);
                    for(int k=i+1;k<(int)input.size();k++){
                            if (!isdigit(input[k])){ i=k; break;}
                          specificSteps.push_back(input[k]);
                            i=k;
                        }
                int singlestep;
                stringstream(specificSteps) >> singlestep;
                Pindex.push_back(singlestep);
                }
            }
            bool repeat=false;
            bool  negative=false;
            for(int z=0;z<(int)Pindex.size();z++){
                     if(Pindex[z]<0) {
                        negative=true;
                        Pindex.erase(Pindex.begin()+z);z--;
                        }
            }
            for(int x=0;x<(int)Pindex.size();x++){
                for(int y=x+1;y<(int)Pindex.size();y++)
                    if(Pindex[x]==Pindex[y]) {
                    repeat=true;
                    Pindex.erase(Pindex.begin()+y);y--;
                    }
            }
            if(repeat) {
                QMessageBox::warning(this,"warning","warning: you entered at least two same number! We
will only keep one of them",QMessageBox::Ok);
              cout<<"warning: you entered at least two same number! We will only keep one of
them"<<endl;
            }
            if(negative)
              cout<<"warning: you entered at least a negative number! We will remove negative
number"<<endl;
            cout<<"valid steps you entered are: "<<endl;
            for(int num=0;num<(int)Pindex.size();num++){
                if(num==(int)Pindex.size()-1) cout<<Pindex[num]<<endl;
                else cout<<Pindex[num]<<",";
            }
    }
//****************************************************************************//
    if(ui->checkBoxDelete->isChecked()) deleteFiles=true;
    int numFronts=1;
     bool multiExists=false;
     if(isPropFile ==false) { multiFronts=false; }
     else {
       ReadSIFs inFileHelper(in_file_name,1,multiFronts,multiExists);  // constructor, used to
initialze the
//inFile object of ReadSIFs type.
       numFronts=inFileHelper.getNumFronts();
       if(numFronts>1) multiExists=true;
       if(multiFronts==false) numFronts=1;
      }


  //************************THIS PART READS THE DATA FILE*************************//
   for(int i=1;i<=numFronts;i++){
       ReadSIFs inFile(in_file_name,i,multiFronts, multiExists);
       inFile.set_isPropFile(isPropFile);  //use set_isPropFile member function to set variables in
inFile
       inFile.storeCurves();
       //declare variables of vector type using curve template
       vector<Curve> Propcurves;
       vector<Curve> GLcurves;
       vector<Curve> GLMLScurves;
       vector<Curve> Lcurves;
       vector<Curve> LMLScurves;


       Propcurves=inFile.getPropSIFs(); // using getPropSIFS() to read data from inFile object and
extract corrensponding values                        // and use them to initialize Propcurves

       //the following extract member fucntions from ReadSIFs class work the same way and extract
different information and store t
       // them in different curve variables.
       GLcurves=inFile.getGLSIFs();
       GLMLScurves=inFile.getGLMlsSIFs();
       Lcurves=inFile.getLSIFs();
       LMLScurves=inFile.getLMlsSIFs();

       cout<<"Read Process done!"<<endl;
       if(Lcurves.size()==0) PL=false;
       if(LMLScurves.size()==0) PLMLS=false;

       string PCols;

       switch(opt){
            case(1): {   //plot KI only

                    PCols = " -bxy 1:5 ";
                    break;
                }

             case(2): {   // plot KII only
                     PCols = " -bxy 1:6 ";
                    break;
                    }
            case(3):{    //plot KIII only
                    PCols = " -bxy 1:7 ";
                     break;
                }
            case(4):{   // this corrensponds to  "PLOT G only:", which should be added in
curve.cpp!!!!!!
                     PCols = " -bxy 1:8 ";
                     break;
                }
            case(5):{   // plot all K's
                     PCols = " -bxy 1:5 -bxy 1:6 -bxy 1:7 ";
                     break;
                }
            case(6):{   // plot all K's and G

                    PCols = " -bxy 1:5 -bxy 1:6 -bxy 1:7 -bxy 1:8 ";
                    break;
                }
       }
       //this connects to WriteXMgrace class; all the previously initialized variables are passed by
values to the constructor
       //which constructs the outFile object of WriteXMgrace type!!!!!!!!!

       if(!deleteFiles){
            bool ok;
          QString output= QInputDialog::getText(this, "output file name", "Please give the output file
name", QLineEdit::Normal,QDir::home().dirName(), &ok);
           out_file_name=output.toStdString();
       }
//******************************THIS PART WRITES OUTPUT FILE*************************//
       WriteXMgrace outFile(out_file_name, Propcurves, GLcurves,
                    GLMLScurves, Lcurves,
                    LMLScurves, PCols, opt, PROP, PGL,
                    PGLMLS, PL, PLMLS, /*planeV,*/
                    /*Center,*/ X_ini, X_end,
                    Pindex, deleteFiles, isPropFile);


       cout << endl << endl<< "Done!!!!"<<endl;


       }
//********************THIS PART ASKS IF USERS WANT TO PLOT AGAIN*****************//
     QMessageBox::StandardButton reply;
     QMessageBox::information(this,"Result","XMGrace data plotted successfully !");
     reply=QMessageBox::question(this,"Question","do you want to continue plotting ?", QMessageBox::Yes
| QMessageBox::No);

     if(reply==QMessageBox::Yes) plotAgain=true;
     this->close();

}
//END OF on_pushButton_clicked()
```

The last place where the signal & slot is used is on checkbox “Plot for specific steps”. If this is checked, then the next three lineEdit boxes: Plot from step, To step, and For every (1,2,3…) steps should be disabled. Otherwise, these three boxes should be enabled and the “Plot for specific steps” should be disabled. Clearly, the signal is emitted by the checkBoxAnySteps object whenever it is checked or unchecked (in other words, when its state is changed). The slot should belong to the SIF2XMGRACE class. The slot function, as defined as “newSteps()”, should work in a way that it disables and enables objects as described above. Therefore, this plot function can be written as:

```
void SIF2XMGRACE::newSteps(){

    if(ui->checkBoxAnySteps->isChecked()){

        ui->lineEdit_specSteps->setEnabled(true);
        ui->lineEdit_fromStep->setDisabled(true);
        ui->lineEdit_toStep->setDisabled(true);
        ui->lineEditSteps->setDisabled(true);
        ui->label_anySteps->setEnabled(true);
        ui->label_3->setDisabled(true);
        ui->label_4->setDisabled(true);
        ui->label_5->setDisabled(true);
    }
    else{

        ui->lineEdit_fromStep->setDisabled(false);
        ui->lineEdit_toStep->setDisabled(false);
        ui->lineEditSteps->setDisabled(false);
        ui->lineEdit_specSteps->setDisabled(true);
        ui->label_anySteps->setDisabled(true);
        ui->label_3->setDisabled(false);
        ui->label_4->setDisabled(false);
        ui->label_5->setDisabled(false);

    }

}
```

And then we can write a connect command inside the constructor implementation:

```
   Connect(ui->checkBoxAnySteps, SIGNAL(stateChanged(int)), this, SLOT(newSteps()));
```

Where “`this`” represent the SIFWXMGRACE class itself.

Using QMessageBox to generate reminder message

It is necessary for the user interface to give a warning message when the user enters an invalid input value; for example, the user forgets to browse for a data file. This can be done by executing an object of QMessageBox whose text is a warning message. For warning of no data file selected, the code can be written as below:

```
QMessageBox mesg;
Mesg.setText("warning! No file selected. Go back to select a file!");
If(in_file_name== "") {
       Mesg.exec();
       Return;
}
```

//Or to make things easier:

```
If(in_file_name== "") {
       QMessageBox::warning(this,"warning","No file selected, please select a file",
QMessageBox::Ok);
       On_pushButton_clicked();
}
```

Then we can put these above lines at the beginning of the on_pushButton_plot_clicked() function.

There are other places where the QMessageBox is used; please refer to the source code for detail.

Conclusion and Potential Improvement

After finishing implementing the on_pushButton_plot_clicked() function, the data can be successfully plotted with the user interface. In summary, the implementation of the SIF2XMGRACE class proves to be feasible for plotting XMGRACE data. This user interface not only improves the efficiency of plotting XMGRACE data, but it is able to remind users of invalid input so that a successful plot can be made.

However, there remain some potential improvements for this user interface. For example, this user interface is tested using the three given XMGRACE data files. Only because it works for these data file doesn’t mean it works for other files (especially those with different format). It can be improved by adding more conditional statements to take into account all the possible inputs. Perhaps other improvements can be proposed by users who use this interface to plot XMGRACE data on a frequent basis.