



**KTH Computer Science
and Communication**

SimpleGraphPlotter v1.6

Programkonstruktion för F, DD1342
Laboration 4A

JIM HOLMSTRÖM
JIMHO@KTH.SE

Teacher: Ann Bengtson

Contents

1	Introduction	1
1.1	Requirements	1
1.2	Scope	2
1.3	Assistance	2
1.4	Compile and Run	2
1.4.1	Compile	2
1.4.2	Run	2
2	Structure	3
2.1	Parser	5
2.1.1	interface iparser	5
2.1.2	class parser	5
2.1.3	interface iexpression	7
2.1.4	class constant	7
2.1.5	class variable	8
2.1.6	class unary_operation	8
2.1.7	class binary_operation	9
2.2	Plotter	9
2.2.1	class function	11
2.2.2	class plot_drawingarea	11
2.2.3	class function_list_controller	11
3	Results and Discussion	15
3.1	Results	15
3.2	Discussion	16
3.2.1	Problems	16
3.2.2	Reflections	16

Chapter 1

Introduction

In the following part firstly the problem will be explained and secondly the requirements for a basic plotter will be listed. A plotter is a program that can plot functions from strings which defines the functions by ordinary math syntax. The project uses C++ programming language and the `gtkmm`¹ wrapper for the `GTK+`² toolkit to generate the graphical user interface. Also `Cairo`³ is used for the raw-rendering. It is compiled with the GNU `gcc-4.6.1` compiler and the repository for the project can be found at `github.com/Jim-Holmstroem/SimpleGraphPlotter` and are available under the GNU GPL license.

1.1 Requirements

A few basic things is needed to have a functioning math plotter:

1. Define a function given ordinary math syntax.
2. Parse the inputed function and plot it accordingly.
3. Add/Remove functions from plotarea.
4. Plotarea should be scrollable both vertical and horizontal.
5. Range should be fixed to the unit-cube.⁴
6. Display axis of the plot.
7. Parser must be properly tested.

¹Documentation, binaries and source can be found at: `www.gtkmm.org`

²Documentation, binaries and source can be found at: `www.gtk.org`

³Documentation, binaries and source can be found at: `www.cairographics.org`

⁴This restriction will be handled in section 1.2

1.2 Scope

The functionality that is possible to put in a system like this is almost endless so a few delimitations has to be made in order to complete the project. The currently biggest restriction to the plotter is the lack of ability to zoom or change the range from the unit-cube. No support for neither parametric nor complex functions.⁵ Also no fileIO support.

1.3 Assistance

Besides the reference manuals for `Cairo`, `gtkmm` and `C++`, no external help for this project was received.

1.4 Compile and Run

To code has been tested to be compiled and executed with Ubuntu 11.10. The program has one non-trivial dependency `gtkmm-3.0` which can be installed from the package manager Synaptic under `libgtkmm-3.0-dev` (compile) and `libgtkmm-3.0-1` (run).

1.4.1 Compile

Under the folder `/src` run

```
make
```

The parser can also be compiled individually by under the folder `/src/parser` run

```
make
```

1.4.2 Run

To run the program

```
./simplegraphplotter
```

The parser can also be executed individually by

```
./parser/parser_test "-(x+sin(1+x))" 2
```

⁵ Since no native support in `C++` for complex numbers which means all the basic math functions would have to be rewritten in order for this to work.

Chapter 2

Structure

A basic overview of the structure can be seen in figure 2.1, all public non-self-explanatory parts will then be listed and explained in a `javadoc` like manner. In the actual code the definition and implementation was separated into `.h` and `.cpp`-files respectively as far as possible,¹ and the code mostly follows Google's style guide for `C++`.² The main goal of the structure for this project is to have as flexible code as possible.

¹Some small trivial methods were left out from this distinction as well as a few things that is hard or impossible to separate in `C++`.

²The style guide can be found at: google-styleguide.googlecode.com

CHAPTER 2. STRUCTURE

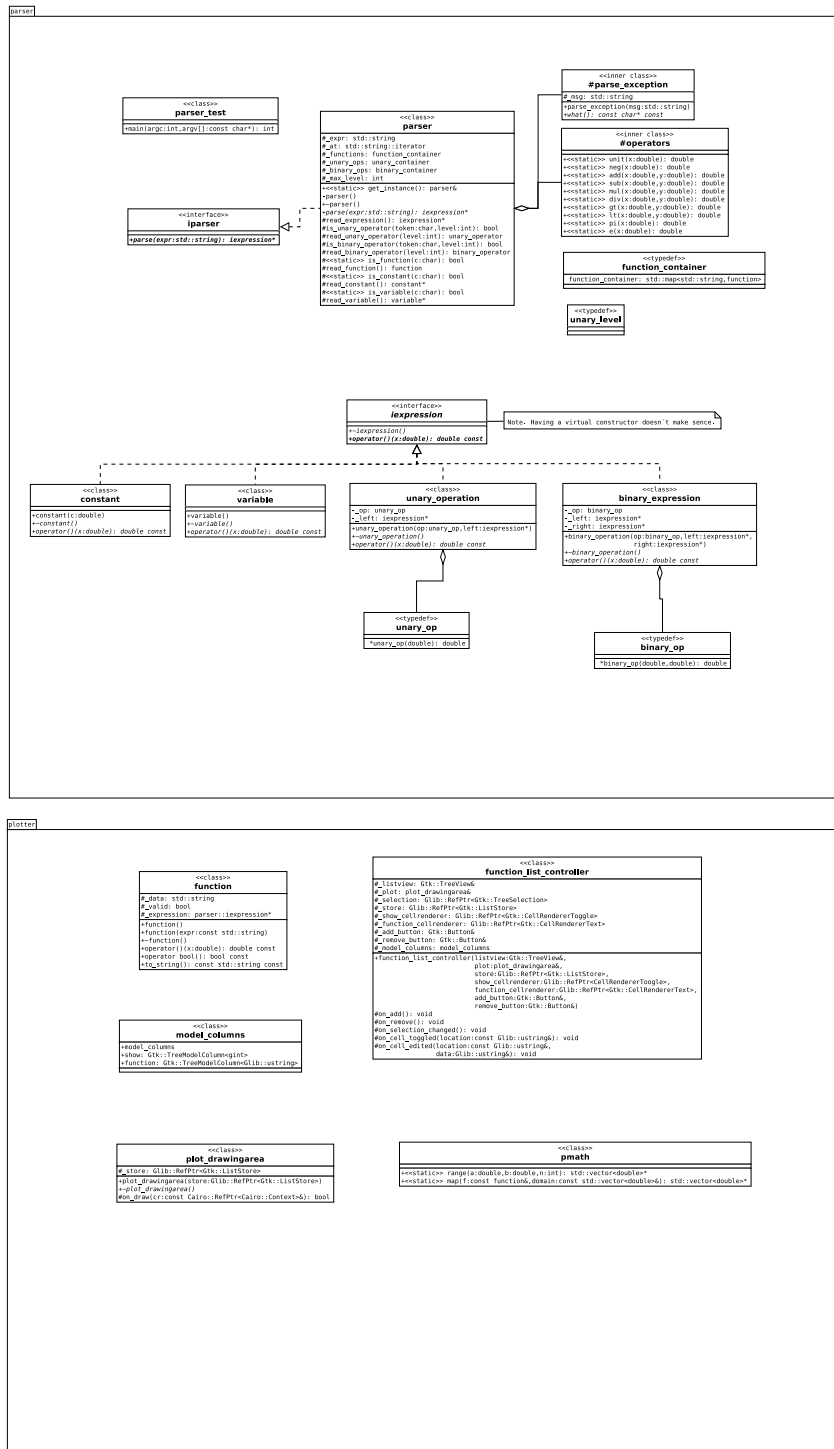


Figure 2.1. An UML showing the structure and the enclosure.

2.1. PARSER

2.1 Parser

The parser can be divided into two parts; the algorithm and the *parse tree* data structure.

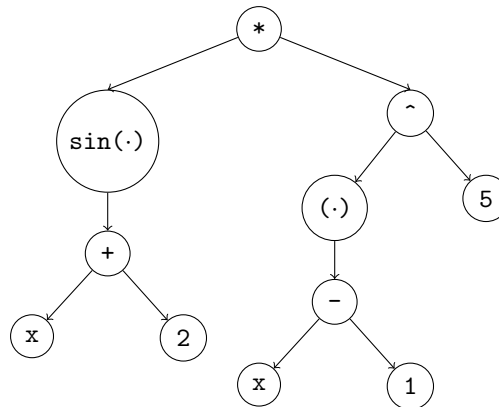


Figure 2.2. An example of the parse tree data structure generated by the parser from the expression `sin(x+2)*(x-1)^5`. Trivial nodes made by the actual implementation where here left out for added clarity.

2.1.1 interface iparser

An *abstract base class* (ABC) that defines the *interface* for what a parser needs to have to be considered a parser, this in case we want to compare different parser implementations or such.

public parse(expr : std::string) Virtual method which handles parsing the string `expr` to generate a parse tree that represents the math expression in `expr`.

Parameters:

`expr` - The string to be parsed.

Returns:

A pointer to the root of the parse tree.

2.1.2 class parser

The parser is a *singleton realization* of `iparser` implementing a *recursive descent parser* algorithm. Two types of methods are used in the parsing, `is-a`³ and `read-it`⁴. The `is-a` is used for look-ahead to determine the type of the expression laying ahead, while `read-it` is used to do the actual syntactic information gathering from

³Starts with `is_`

⁴Starts with `read_`

the expression fragment. The EBNF⁵ syntax for the parsing:

```

plots = term-(-1),[';',expression-(-1)],'\n' (* no support for ';' this \
implementation *) (* -1 is the lowest order expression *)
expression-i = [unary-i],expression-(i+1),[op-(i+1),expression-i]
term-n = var | num | [function],(,term-(-1),) \
(* n is the number of the highest order operator *)

op-0 = '>' | '<'
op-1 = '+' | '-'
op-2 = '*' | '/' | '%'
op-3 = '^'
unary-3 = '+' | '-' | '*'
num = ? all numbers ?
var = 'x'
function = cos | sin | tan | acos | asin | atan | cosh \
| sinh | tanh | exp | log | log10 | sqrt | ceil | abs \
| floor | pi | e (* where pi and e are constant
functions *)

```

In the definition of `expression-i` either `unary-(i+1)` or `op-(i+1)` is chosen. Unary, since on the left, has higher priority.

Worth noting is that the parser has two protected temporary instance variables `_expr` and `_at` which are much more non-local than is needed. This might as well have been solved by passing them by reference down in the parsing to preserve the locality to within the `parse` method. In favour of code readability the later alternative was not considered. This has the consequence that it makes some of the parsing methods technically *non-static*, although the parser as a singleton can therefore only have one instance per program in the same way as if it were static.

static public get_instance()

Gets the singleton instance of the parser, if not yet instantiated it calls the private constructor and instantiates it.

Parameters:

None

Returns:

A pointer to the singleton instance of the parser.

public parse(expr : std::string) Parses the string `expr` to generate a parse tree that represents the math expression in `expr`.

⁵iso-14977.pdf

2.1. PARSER

Parameters:

expr - The string to be parsed.

Returns:

A pointer to the root of the parse tree.

2.1.3 interface iexpression

Acts as abstract base class (ABC) for a node in the parse tree, as the nodes in figure 2.2. The **iexpression** is considered a *functor* since it has overloaded the **operator** and can thus be called as function. The **operator** can be both *recursively* implemented, as in **unary**⁶ and **binary**⁷, or *explicitly*⁸ implemented, as in **constant**⁹ and **variable**¹⁰. The operator function is generally defined by:

$$\begin{aligned}\text{expression} : \mathbb{R} &\rightarrow \mathbb{R} \\ x &\mapsto \text{operator}(x).\end{aligned}\tag{2.1}$$

public operator(x : double) double const Virtual definition of the *evaluation* function for expressions. Since the operator is going to act as a mathematical function one must be certain that it behaves like one, that is it does not modifies the functor when called.¹¹ Therefore the **const** keyword has been added to prevent this from accidentally happening in the realizations.

Parameters:

x - Input value for the expression.

Returns:

The output value of this expression given the parameter **x**.

2.1.4 class constant

A *realization* of **iexpression** 2.1.3 which represents a constant. To keep constancy with the **iexpression** this is implemented as a constant-function:

$$\begin{aligned}\text{constant} : \mathbb{R} &\rightarrow \mathbb{R} \\ x &\mapsto c.\end{aligned}\tag{2.2}$$

public constant(c : double) Constructor that constructs the function in the equation 2.2.

Parameters:

c - The value of the constant in the expression.

⁶As can be seen in equation 2.4.

⁷As can be seen in equation 2.5.

⁸Explicit in the mathematical sense not as the keyword in C++.

⁹As can be seen in equation 2.2.

¹⁰As can be seen in equation 2.3.

¹¹In contrast to a method where that behaviour is allowed.

public operator(x : double) double const Explicit realization of `iexpression.operator` returning a constant.

Parameters:

`x` - Input value of the expression, does not matter only there for compatibility reasons.

Returns:

The output value of the expression.

2.1.5 class variable

A realization of `iexpression` 2.1.3 which represents a variable. A variable can simply be seen as an unit-function:

$$\begin{aligned} \text{variable} : \mathbb{R} &\rightarrow \mathbb{R} \\ x &\mapsto x. \end{aligned} \tag{2.3}$$

public variable() Constructor that constructs the function in the equation 2.3.

parameters:

None

public operator(x : double) double const Explicit realization of `iexpression.operator` returning the value of the input.

Parameters:

`x` - Input value of the expression.

Returns:

The output value of the expression.

2.1.6 class unary_operation

A realization of `iexpression` 2.1.3 which represents an unary operation, a function constructed with `op left`:

$$\begin{aligned} \text{unary} : \mathbb{R} &\rightarrow \mathbb{R} \\ x &\mapsto \text{op}(\text{left}(x)). \end{aligned} \tag{2.4}$$

public unary_operation(op : unary_op, left : iexpression*) Constructor that constructs the function in the equation 2.4.

Parameters:

`op` - The unary operation performed, which is an `unary_op`¹².

`left` - The inner expression on which to perform the operation on.

¹²Typedefined to be a function pointer: `*unary_op(double):double`.

2.2. PLOTTER

public operator(x : double) double const Recursive realization of `iexpression.operator` returning the returned value of `left` through `op`, as in equation 2.4.

Parameters:

`x` - Input value for the `left` expression.

Returns:

The returned value from the equation 2.4.

2.1.7 class binary_operation

A realization of `iexpression` 2.1.3 which represents a binary operation, that is a function constructed with `op` and `left/right`:

$$\begin{aligned} \text{binary} : \mathbb{R} &\rightarrow \mathbb{R} \\ x &\mapsto \text{op}(\text{left}(x), \text{right}(x)). \end{aligned} \quad (2.5)$$

public binary_operation(op : binary_op, left : iexpression*, right : iexpression*)
Constructor that constructs the function in the equation 2.5.

Parameters:

`op` - The unary operation performed, which is a `binary_op`¹³.

`left` - The left expression on which to perform the operation on.

`right` - The right expression on which to perform the operation on.

public operator(x : double) double const Recursive realization of `iexpression.operator` returning the returned values of `left` and `right` through `op`, as in equation 2.5.

Parameters:

`x` - Input value for the expression.

Returns:

The returned value from the equation 2.5.

2.2 Plotter

The GUI code for the plotter is coarsely separated into a *MVC* pattern. The model in this project is the liststore for the `function`'s. This model has two different views: `Gtk::TreeView` which lists the functions, as can be seen in figure 2.4 and `plot_drawingarea`, as can be seen in figure 2.3. The controller is the `function_list_controller` which handles the changes in the model and redraws the views when needed. Almost all layout is separately define in an `.ui`-file¹⁴ which is loaded by the *builder* `Gtk::Builder` at startup.

¹³Typedefined to be a function pointer: `*binary_op(double,double):double`.

¹⁴Follows XML standard.

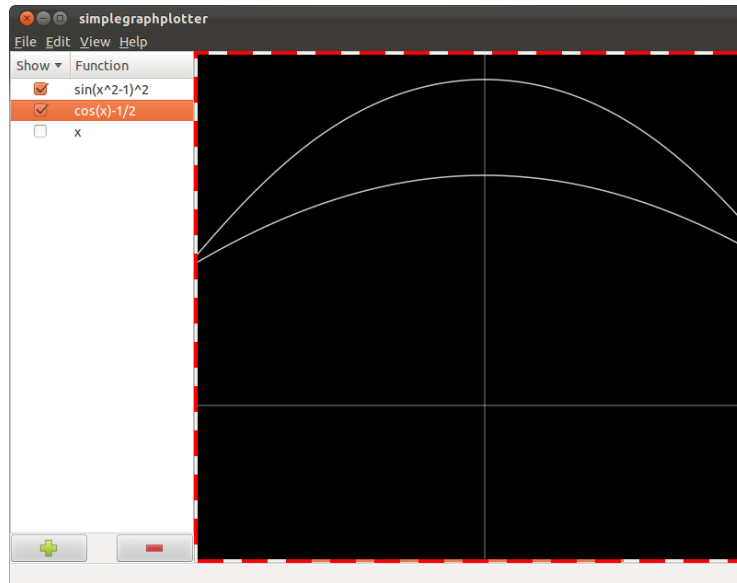


Figure 2.3. A screenshot of the program under normal operation, with the plotarea highlighted.

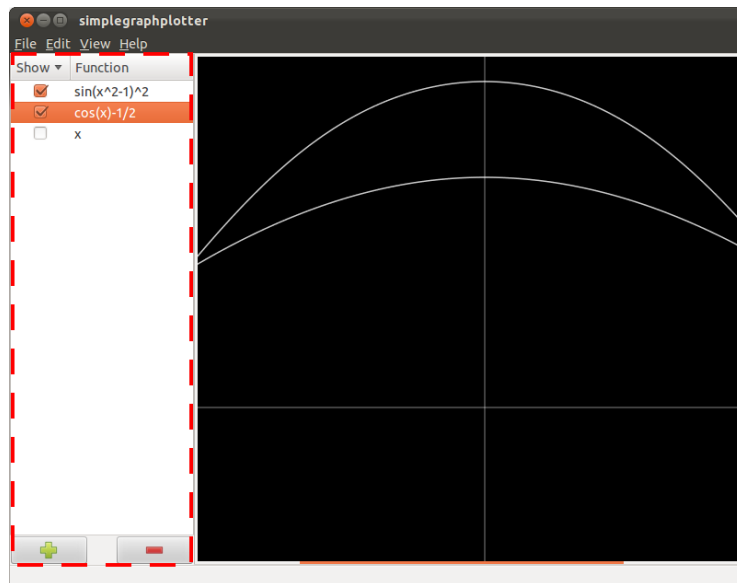


Figure 2.4. A screenshot of the program under normal operation, with the treeview and add/remove buttons highlighted.

2.2. PLOTTER

2.2.1 class function

Acts as a model of a function which is used by `plot_drawingarea`. It has the *signature* for `iexpression` but it has not got the same intended usage and should therefore not be a realization of `iexpression`.

public function(expr : const std::string) Constructor that constructs the function from the string `expr`.

Parameters:

`expr` - The expression to use as a function.

2.2.2 class plot_drawingarea

Acts as a plotting view for the functions in the liststore `store` provided in the constructor. It inherits from the `Gtk::DrawingArea` to get some code and to have the right signature to work in the layout.

public plot_drawingarea(store : Glib::RefPtr<Gtk::ListStore>) Constructor that constructs a `plot_drawingarea`.

Parameters:

`store` - Reference¹⁵ to the liststore that contains the functions to be rendered. Must be, or in the same format as `function_store` defined in the `.ui`-file.

protected on_draw(cr : const Cairo::RefPtr<Cairo::Context>&)
The draw signal handler for the `plot_drawingarea`.

Parameters:

`cr` - The reference to the `Cairo::Context` on which to draw on.

Returns:

Mostly true.

2.2.3 class function_list_controller

A *controller* that handles the functions. The protected methods starting with `on_` is *action listeners* and can be overridden in a new child class to change the functionality on them without the need to reimplement the basecode.¹⁶

¹⁵`Gtk::RefPtr` is an reference-counting shared smart pointer.

¹⁶One should always avoid code duplication.

```

public function_list_controller(
listview : Gtk::TreeView&,
plot : plot_drawingarea&,
store : Glib::RefPtr<Gtk::ListStore>,
show_cellrenderer : Glib::RefPtr<Gtk::CellRendererText>,
add_button : Gtk::Button&,
remove_button : Gtk::Button&
)

```

Constructor for `function_list_controller`.

Parameters:

`listview` - Reference to the view of the list.
`plot` - Reference to the plot view for the functions.
`store` - Reference to the liststore that handles the functions.
`show_cellrenderer` - Reference to the `Gtk::CellRendererText` which defines the textfield's appearance in the `store`.
`add_button` - A reference to the button that controls adding new functions into the `store`.
`remove_button` - A reference to the button that controls removing functions from the `store`.

protected on__add()

Signal handler for adding a function.

Parameters:

None

protected on__remove()

Signal handler for removing a function.

Parameters:

None

protected on__selection_changed()

Signal handler for selection change.

Parameters:

None

protected on__cell_toggled(location : const Glib::ustring&)

Signal handler for toggling the visibility of a function.

Parameters:

`location` - The location in the liststore of the function being toggled.

2.2. PLOTTER

protected on__cell__edited(location : const Glib::ustring&, data : const Glib::ustring&)

Signal handler for editing a function.

Parameters:

location - The location in the liststore of the function begin edited.

data - What the function is being edited to.

Chapter 3

Results and Discussion

3.1 Results

A screenshot of the final result of this project can be seen in figure 3.1.

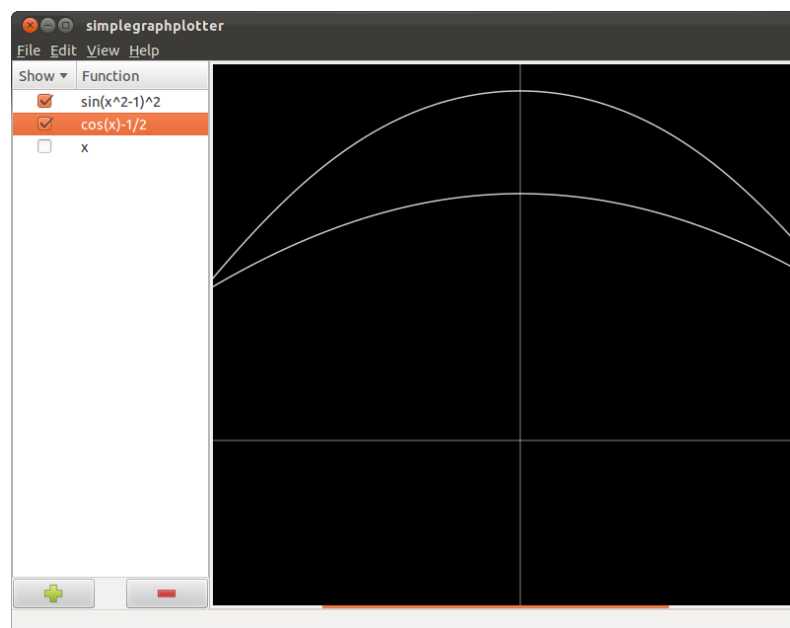


Figure 3.1. A screenshot of the program under normal operation.

The program did not have any memory leaks under normal operations when tested in the program `valgrind`.

All the points in the list from section 1.1 was fulfilled and no failing test cases for the parser was found.

3.2 Discussion

3.2.1 Problems

The unofficial C++ wrapper `gtkmm`¹ where quite immature and to implement much of the common functionality for this type of applications easily became hacky or needed to workaround bugs. Also the sparse documentation was sometimes a problem.

In a parser it is easy to miss out test case combinations, the latest miss which almost went through to release was a priority miss between unary and binary operators which resulted in:

$$-x^2 = (-x)^2$$

which is not the case for math expressions.

3.2.2 Reflections

One needs, even in this rather small project, quite a few iterations of the overall design as well as on the lower level to get an overall good and flexible structure.

¹Which only was used to get inheritance, polymorphism and having it compatible with the standard C++Library.