KTH Computer Science
and Communication

SimpleGraphPlotter v1.6

Programkonstruktion för F, DD1342
Laboration 4A

JIM HOLMSTRÖM
JIMHO@KTH.SE

Teacher: Ann Bengtson

Contents

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. This 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. The codebase can be found at github.com/Jim-Holmstroem/SimpleGraphPlotter .

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.

[1]Documentation, binaries and source can be found at: www.gtkmm.org
[2]Documentation, binaries and source can be found at: www.gtk.org
[3]Documentation, binaries and source can be found at: www.cairographics.org
[4]This restriction will be handled in section 1.2

1.2 Scope

The amount of 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. Also no IO support [5]

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 run under the platform 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
```

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

An 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.

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

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

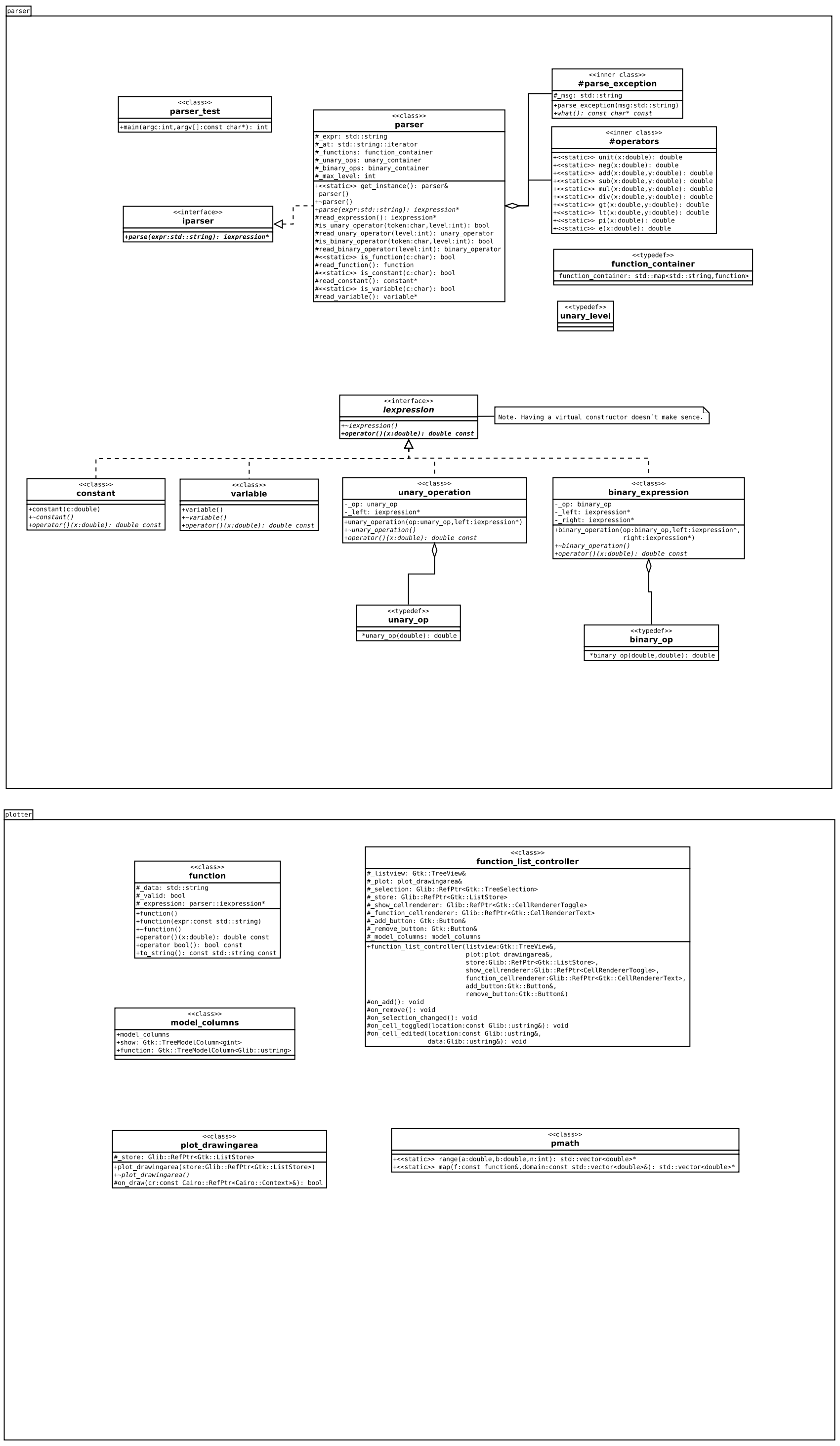
parser
<<class>>
parser_test
+main(argc:int,argv[]:const char*): int
<<class>>
parser
#_expr: std::string
#_at: std::string::iterator
#_functions: function_container
#_unary_ops: unary_container
#_binary_ops: binary_container
#_max_level: int
+<<static>> get_instance(): parser&
-parser()
+~parser()
+parse(expr:std::string): iexpression*
#read_expression(): iexpression*
#is_unary_operator(token:char,level:int): bool
#read_unary_operator(level:int): unary_operator
#is_binary_operator(token:char,level:int): bool
#read_binary_operator(level:int): binary_operator
#<<static>> is_function(c:char): bool
#read_function(): function
#<<static>> is_constant(c:char): bool
#read_constant(): constant*
#<<static>> is_variable(c:char): bool
#read_variable(): variable*
<<interface>>
iparser
+parse(expr:std::string): iexpression*
<<inner class>>
#parse_exception
#_msg: std::string
+parse_exception(msg:std::string)
+what(): const char* const
<<inner class>>
#operators
+<<static>> unit(x:double): double
+<<static>> neg(x:double): double
+<<static>> add(x:double,y:double): double
+<<static>> sub(x:double,y:double): double
+<<static>> mul(x:double,y:double): double
+<<static>> div(x:double,y:double): double
+<<static>> gt(x:double,y:double): double
+<<static>> lt(x:double,y:double): double
+<<static>> pi(x:double): double
+<<static>> e(x:double): double
<<typedef>>
function_container
function_container: std::map<std::string,function>
<<typedef>>
unary_level
<<interface>>
iexpression
+~iexpression()
+operator()(x:double): double const
Note. Having a virtual constructor doesn´t make sence.
<<class>>
constant
+constant(c:double)
+~constant()
+operator()(x:double): double const
<<class>>
variable
+variable()
+~variable()
+operator()(x:double): double const
<<class>>
unary_operation
-_op: unary_op
-_left: iexpression*
+unary_operation(op:unary_op,left:iexpression*)
+~unary_operation()
+operator()(x:double): double const
<<class>>
binary_expression
-_op: binary_op
-_left: iexpression*
-_right: iexpression*
+binary_operation(op:binary_op,left:iexpression*,
right:iexpression*)
+~binary_operation()
+operator()(x:double): double const
<<typedef>>
unary_op
*unary_op(double): double
<<typedef>>
binary_op
*binary_op(double,double): double
plotter
<<class>>
function
#_data: std::string
#_valid: bool
#_expression: parser::iexpression*
+function()
+function(expr:const std::string)
+~function()
+operator()(x:double): double const
+operator bool(): bool const
+to_string(): const std::string const
<<class>>
function_list_controller
#_listview: Gtk::TreeView&
#_plot: plot_drawingarea&
#_selection: Glib::RefPtr<Gtk::TreeSelection>
#_store: Glib::RefPtr<Gtk::ListStore>
#_show_cellrenderer: Glib::RefPtr<Gtk::CellRendererToggle>
#_function_cellrenderer: Glib::RefPtr<Gtk::CellRendererText>
#_add_button: Gtk::Button&
#_remove_button: Gtk::Button&
#_model_columns: model_columns
+function_list_controller(listview:Gtk::TreeView&,
plot:plot_drawingarea&,
store:Glib::RefPtr<Gtk::ListStore>,
show_cellrenderer:Glib::RefPtr<CellRendererToogle>,
function_cellrenderer:Glib::RefPtr<Gtk::CellRendererText>,
add_button:Gtk::Button&,
remove_button:Gtk::Button&)
#on_add(): void
#on_remove(): void
#on_selection_changed(): void
#on_cell_toggled(location:const Glib::ustring&): void
#on_cell_edited(location:const Glib::ustring&,
data:Glib::ustring&): void
<<class>>
model_columns
+model_columns
+show: Gtk::TreeModelColumn<gint>
+function: Gtk::TreeModelColumn<Glib::ustring>
<<class>>
plot_drawingarea
#_store: Glib::RefPtr<Gtk::ListStore>
+plot_drawingarea(store:Glib::RefPtr<Gtk::ListStore>)
+~plot_drawingarea()
#on_draw(cr:const Cairo::RefPtr<Cairo::Context>&): bool
<<class>>
pmath
+<<static>> range(a:double,b:double,n:int): std::vector<double>*
+<<static>> map(f:const function&,domain:const std::vector<double>&): std::vector<double>*

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

2.1 Parser

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

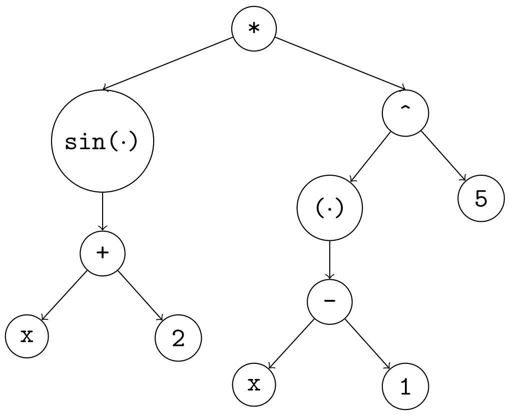

Figure 2.2. An example of the parse tree data structure generated by the parser from the expression `sin(x+2)*(x-1)^5`. Trivial nodes made by the actual implementation where here left out for added clarity.

2.1.1 interface iparser

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

public parse(expr : std::string) Virtual method that should be overloaded so that it will parse 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 an *singelton* implementation of a *recursive descent parser*. 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 which type of expression that lays ahead, while `read-it` is used to do the actual syntactic information gathering from the expression fragment.

[3]Starts with `is_`
[4]Starts with `read_`

The EBNF syntax for the parsing made by this algorithm is as follows:

```
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 choosen. Unary, since on the left, has higher priority.

Note that the parser has the protected temporary instance variables `_expr` and `_at`. This might as well have been solved by passing them by reference down in the parsing to preserve the locality inside the `parse` method. Avoided this in favour for code readability. This technically makes some of the parsing methods non-*static* but since being singelton they can only be one instance per program in the same way as static.

static public get_instance()
Gets the singelton instance of the parser, if not yet instantiated it calls the private constructor and instantiates it.

Parameters:
`None`

Returns:
A pointer to the singelton 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`.

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 in the same way as any other function. The `operator` can be both *recursively* implemented, as in `unary`[5] and `binary`[6] , or *explicitly* implemented, as in `constant`[7] and `variable`[8] . The operator function is generally defined by:

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

public operator(x : double) double const Virtual definition of the *evaluation* function for expressions. Since the operator is going to act as an mathematical function one must be certain that it behaves like one, that is it does not modifies the functor when called.[9] 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

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

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

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

Parameters:
`c` - The value of the constant in the expression.

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.

[5]As can be seen in equation 2.4.
[6]As can be seen in equation 2.5.
[7]As can be seen in equation 2.2.
[8]As can be seen in equation 2.3.
[9]In contrast to a method where that behaviour is allowed.

Returns:
The output value of the expression.

2.1.5 class variable

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

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

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

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

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

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`[10] .
`left` - The inner expression on which to perform the operation on.

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.

[10]Typedefined to be a function pointer: `*unary_op(double):double`.

2.1.7 class binary_operation

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

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

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 an `binary_op`[11] .
`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 in this pattern is the `function_list_controller` .....

[11]Typedefined to be a function pointer: `*binary_op(double,double):double`.

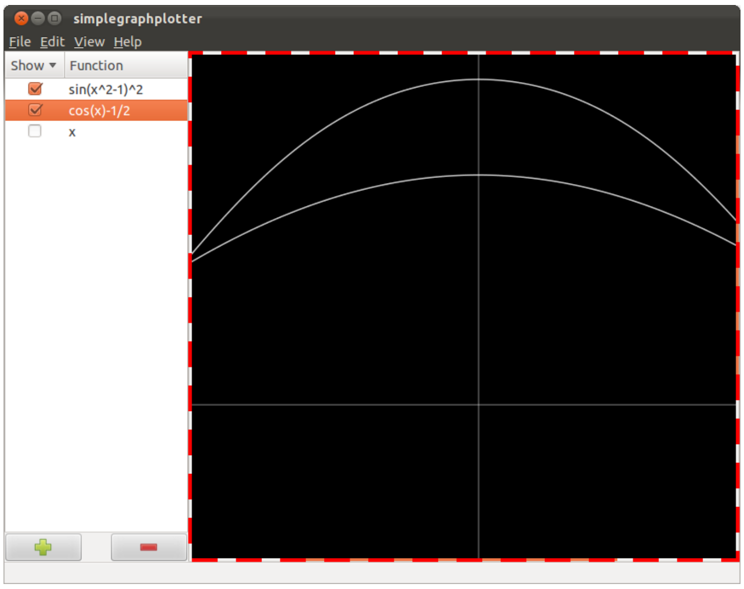

Figure 2.3. A screenshot of the program under normal operation, with the plotarea highlighted.

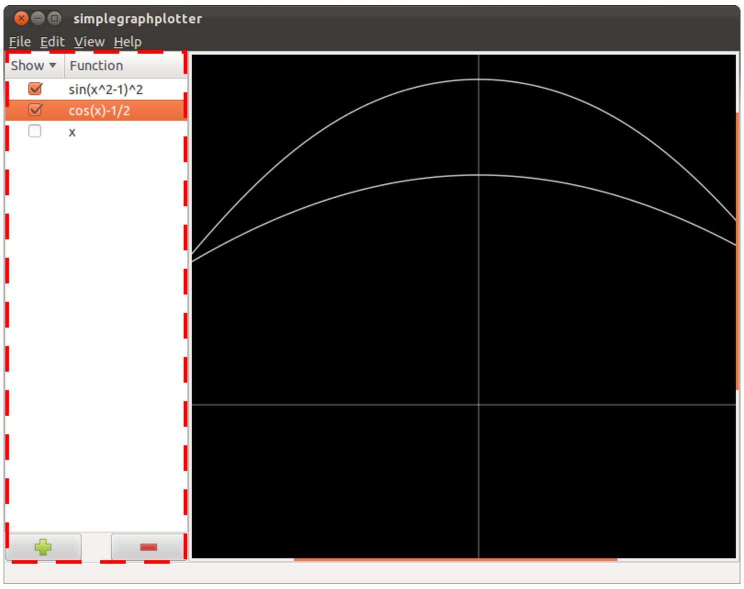

Figure 2.4. A screenshot of the program under normal operation, with the view of the function_list_controller highlighted.

>especially point out the inheritance in the custom widgets. >almost all layout

is separetally defined in a .ui file[12] and loaded by the gtk *builder* at startup. >all signal connections are defined in main at generation. (main.cc)

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.

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

Parameters:
`store` - Reference 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 ...

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 overriden in a new child class to change the functionality on them without the need to reimplement basecode.[13]

[12]Follows XML standard.
[13]One should always avoid duplicating code.

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 in the list.
`plot` - Reference to the view of the actual plot of the functions.
`store` - Reference to the liststore that handles the functions.
`show_cellrenderer` -
`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.

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

Runned trough valgrind, results?.

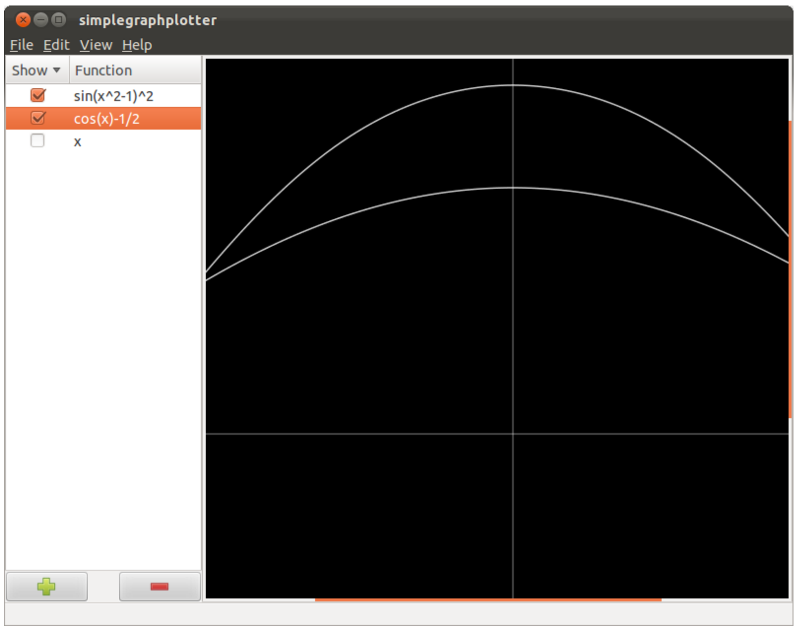

Figure 3.1. A screenshot of the program under normal operation.

3.2 Discussion

= Problems with the unofficial `C++`wrapper `gtkmm`, only used it to avoid missing out inheritance, polymorphism and to get it compatible with the standard `C++`Library.

Many normal things easily became hacky. = Easy to miss combinations in the parser and have bugs.

One needs a few iterations of the overall design to get a good flexible structure.