

# QuantumGL

Wolfgang Thaller

3rd April 2004

## Abstract

QuantumGL is an OpenGL-based tool for the visualization of volume data. In a text file, you define the data to be visualized and the visualization methods to be used. You can define an arbitrary number of real-valued parameters which can then be modified interactively from the graphical user interface.

## Contents

<b>I</b>	<b>Visualization Task Files</b>	<b>3</b>
<b>1</b>	<b>Lexical Syntax</b>	<b>3</b>
1.1	Whitespaces and Statements . . . . .	3
1.2	Comments . . . . .	3
1.3	Identifiers . . . . .	3
1.4	Floating-Point Constants . . . . .	4
<b>2</b>	<b>Variables</b>	<b>4</b>
2.1	Predefined Variables . . . . .	4
<b>3</b>	<b>Expressions</b>	<b>5</b>
3.1	Expression Attributes . . . . .	5
3.1.1	Type . . . . .	5
3.1.2	Default Resolution . . . . .	5
3.1.3	Range . . . . .	6
3.2	Implicit type conversions . . . . .	6
3.3	Values . . . . .	6
3.3.1	Literal Constants . . . . .	6
3.3.2	Imaginary Unit . . . . .	6
3.3.3	Variable References . . . . .	6
3.3.4	Field References . . . . .	6
3.3.5	Field Minimum and Maximum . . . . .	7
3.4	Operators . . . . .	7
3.4.1	cut_to . . . . .	7

3.4.2	scaled_to . . . . .	7
3.4.3	Vector Composition . . . . .	8
3.4.4	Unary Minus (-) . . . . .	8
3.4.5	Arithmetic Operators . . . . .	8
3.5	Functions . . . . .	8
3.5.1	read . . . . .	8
3.5.2	abs . . . . .	8
3.5.3	Standard Unary Functions . . . . .	8
3.5.4	Self-Made Functions . . . . .	9
<b>4</b>	<b>Setting Options</b>	<b>9</b>
4.1	nobox . . . . .	9
4.2	colorbox . . . . .	9
4.3	grid . . . . .	9
4.4	background . . . . .	9
4.5	fog . . . . .	9
4.6	resolution . . . . .	10
4.7	world_size . . . . .	10
<b>5</b>	<b>Field Declarations</b>	<b>10</b>
<b>6</b>	<b>Animations</b>	<b>10</b>
<b>7</b>	<b>Visualization Objects</b>	<b>11</b>
7.1	Isosurfaces . . . . .	11
7.1.1	Cyclic . . . . .	12
7.1.2	Cutout . . . . .	12
7.2	Slices . . . . .	12
7.3	Vectors . . . . .	12
7.3.1	Point Lists . . . . .	12
7.4	Flow Lines . . . . .	13
7.4.1	Point Lists . . . . .	13
7.4.2	Step Size . . . . .	13
<b>II</b>	<b>Data Files</b>	<b>13</b>
<b>8</b>	<b>ASCII Data Files</b>	<b>14</b>
<b>9</b>	<b>Binary Data Files</b>	<b>14</b>
<b>III</b>	<b>Extending QuantumGL</b>	<b>14</b>

<b>10</b>	<b>Predefined Types</b>	<b>15</b>
10.1	Operations on Scalar Values . . . . .	16
10.2	Operations on Vectors . . . . .	17
10.3	Operations on Fields . . . . .	17
<b>11</b>	<b>How to add a Field Function</b>	<b>17</b>
11.1	The Header File . . . . .	17
11.2	The Source File . . . . .	18
11.3	The Field Function Registry . . . . .	18

## Part I

# Visualization Task Files

A Visualization Task File is a plain text file with the extension `.qgl`<sup>1</sup>. It consists of a list of statements which are processed linearly when the file is opened in the QuantumGL application.

## 1 Lexical Syntax

### 1.1 Whitespace and Statements

Whitespace (spaces, tabs and newlines) is not significant, except where it is required as a separator. Every statement is terminated by a semicolon — multiple statements may be put on one line, and a statement may span multiple lines.

### 1.2 Comments

Line comments start whenever the hash character (`#`) or two slash characters (`//`) are encountered and last until the end of the line.

C-style block comments start with the sequence `/*` and last until the sequence `*/` is encountered. They can not be nested.

You can also use `(*` and `*)` to comment out a block of code. Other than the C-style block comments above, you can nest this style of comments.

### 1.3 Identifiers

Identifiers consist of one or more letters, numbers or underscores (`_`). The first character of an identifier must not be a number. Identifiers are case sensitive.

All reserved keywords used by QuantumGL are in lowercase. Future versions of QuantumGL may have more reserved keywords, so in order to make sure that your Visualization Task File can be used unchanged with future versions, use at least one uppercase character in your identifiers.

---

<sup>1</sup>Pronounced “coogle”, like the German word “Kugel” (sphere)

## 1.4 Floating-Point Constants

Examples for floating-point constants:

```
42
3.141592654
0.5
-.5
3e17
-1.5e-6
.3e+2
```

Also, the constants  $\pi$  and  $2\pi$  can be written as follows:

```
pi
2pi
```

## 2 Variables

Variables in QuantumGL have a real value. The QuantumGL application can show a list of all defined variables and allows to change their values. When a variable is changed, all objects that depend on that value are automatically re-calculated.

A variable is declared as follows:

```
variable identifier := constant;
```

Please note that the right-hand value has to be a constant, i.e. the value cannot depend on another variable.

When you declare a variable, you can also specify a minimum and maximum value in square brackets:

```
variable identifier [constant, constant] := constant;
```

### 2.1 Predefined Variables

QuantumGL predifines some variables for global camera and lighting parameters. You can change the initial value of these variables as follows:

```
identifier := constant;
```

The following variables are predefined in the current version of QuantumGL:

**latitude** The latitude angle of the camera, in degrees ( $-90^\circ < \text{latitude} < 90^\circ$ )

**longitude** The longitude angle of the camera.

**distance** The distance of the camera from the center of the cube.

**fovy** The field-of-view angle for the camera.

**lightLatitude** The latitude angle of the light source.

**lightLongitude** The longitude angle of the light source.

**ambient** The ambient lighting level.

**lineWidth** The width of the lines used for the bounding box and for **flow** objects (in pixels).

## 3 Expressions

There are two kinds of expressions in QuantumGL: field expressions and vector expressions. Scalar values are considered to be one-dimensional vectors in QuantumGL.

A VECTOR may have between one and ten components<sup>2</sup>. The components of the vector may be either real or complex.

A FIELD is a function that maps from  $\mathbb{R}^3$  to a vector type.

### 3.1 Expression Attributes

Each expression and subexpression has a set of attributes which are determined only by the expression and its subexpressions, not by its context. The first and most important of these attributes is the type of the expression, which has already been mentioned above.

#### 3.1.1 Type

Let  $K$  be either  $\mathbb{R}$  or  $\mathbb{C}$  and  $n$  a positive integer. A vector expression has a type of  $K^n$  and a field expression has the type  $\mathbb{R}^3 \mapsto K^n$ .

#### 3.1.2 Default Resolution

This attribute applies to field expressions only. The default resolution specifies at how many points the field should be sampled when it is evaluated. You can specify a different resolution in every statement that evaluates a field.

By default, this is initialized to a global default value (see section 4.6 on page 10). For references to fields defined using a **field** statement (see section 5 on page 10), the default resolution is the resolution the field was evaluated at. For fields loaded from a file using the **read** function, it is the resolution stored in the file.

For compound expressions, the default resolution is determined from the resolutions of the subexpressions.

---

<sup>2</sup>This limit can be increased by modifying the **maxDimensions** constant in the source file “CommonEvaluator.h” and recompiling QuantumGL.

### 3.1.3 Range

This attribute applies to field expressions only. This attribute specifies the range of coordinates that the field should be evaluated at. It is either an interval in  $\mathbb{R}^3$ , or undefined. Fields with undefined range cannot be evaluated. You can change this attribute using the `scaled_to` and `cut_to` operators.

For compound expressions, the ranges of all subexpressions whose range is defined have to match. This range is also the range of the compound expression. If all subexpressions have undefined range, the range of the compound expression is undefined, too.

## 3.2 Implicit type conversions

When you use a vector expression where a field expression is required, the vector is automatically converted to a constant field. You cannot use a field expression where a vector expression is required. Likewise, real vectors or real vector fields are automatically promoted to complex vectors or complex vector fields where needed, but there is no implicit conversion in the opposite direction.

## 3.3 Values

### 3.3.1 Literal Constants

See section 1.4 on page 4 for the lexical syntax of floating point constants. Constant expressions have the type  $\mathbb{R}^1$ .

A negative sign is not considered part of the constant. Rather, it is an application of the unary minus operator (see 3.4.4).

### 3.3.2 Imaginary Unit

The constant `i` is defined to be the imaginary unit constant. Its type is  $\mathbb{C}^1$ .

### 3.3.3 Variable References

You can refer to variables (section 2 on page 4) by their name. As variables have real values, the type of a variable reference expression is  $\mathbb{R}^1$ . The statement containing the expression will automatically be recalculated when the value of the variable is changed interactively.

### 3.3.4 Field References

You can refer to fields (section 5 on page 10) by their name. The resulting type is, of course, that of the defined field, and its default resolution and range are the resolution and range used to evaluate the field in the `field` statement. If a value between the evaluated data points of the defined field is required, it is interpolated using trilinear interpolation. Outside of the range of the field, its value is considered to be 0.

### 3.3.5 Field Minimum and Maximum

```
field_name.min  
field_name.max
```

You can access the minimum and maximum values of a real scalar field by adding `.min` or `.max` to the field name, respectively. You can only do that for fields that you have declared with a `field` statement, not to more complex field expressions.

## 3.4 Operators

### 3.4.1 cut\_to

```
field_expression cut_to <[x_min,x_max],[y_min,y_max],  
                        [z_min,z_max]>  
field_expression cut_to <x_diameter,y_diameter,z_diameter>  
field_expression cut_to [min,max]  
field_expression cut_to diameter
```

The result is the same as *field\_expression* with its range reduced to the specified range. If the *field\_expression* already has a range, the specified range must fit inside it.

The range may be specified in several different ways. You may specify the minimum and maximum values separately for each dimension. You may also just specify one [*min*,*max*] pair for all three dimensions. Instead of a pair, you may also specify just a single value as the diameter of the interval. This is taken to mean [ $-\text{diameter}/2, \text{diameter}/2$ ].

Note that all the values on the right-hand side of the `cut_to` operator must be constants, not expressions.

### 3.4.2 scaled\_to

```
field_expression scaled_to <[x_min,x_max],[y_min,y_max],  
                        [z_min,z_max]>  
field_expression scaled_to <x_diameter,y_diameter,z_diameter>  
field_expression scaled_to [min,max]  
field_expression scaled_to diameter
```

The *field\_expression*, which must already have a defined range, is translated and scaled. The old range of coordinates is mapped to the specified range. The range of the result is equal to the specified range.

The range is specified in the same way as with `cut_to` (section 3.4.1). As with `cut_to`, all the values on the right-hand side of the `scaled_to` operator must be constants.

### 3.4.3 Vector Composition

You can compose a vector from scalar values by enclosing the components in angle brackets, separated by commas:

`<component1, component2, ..., componentN>`

All of the components must be scalar expressions. The resulting vector is complex if at least one of the components is complex, and real otherwise.

### 3.4.4 Unary Minus (-)

You can prefix any expression (both vectors and fields) with a minus sign to negate it.

### 3.4.5 Arithmetic Operators

Arithmetic operators in QuantumGL may be applied to both fields and vectors. The result is a vector if both arguments are vectors. Otherwise, the result is a field. If both arguments are real, the result is real, otherwise, it is complex.

The following table shows the available arithmetic operators and the allowed combinations of vector and scalar types (let  $K$  be either  $\mathbb{R}$  or  $\mathbb{C}$ , and let  $n$  be a positive integer  $\geq 2$ ):

operation	operator	$K \circ K$	$K \circ K^n$	$K^n \circ K$	$K^n \circ K^n$
Addition	+	$K$	no	no	$K^n$
Subtraction	-	$K$	no	no	$K^n$
Multiplication	*	$K$	$K^n$	$K^n$	no
Division	/	$K$	no	$K^n$	no
Scalar Product	.	$K$	no	no	$K$

## 3.5 Functions

### 3.5.1 read

`read("filename")`

The `read` function will read a field from the specified file; the file format is specified in Part II, “Data Files”. The data points from the file are spaced in the unit cube (the range of the resulting field is  $<[-1,1], [-1,1], [-1,1]>$ ), and the values are interpolated if necessary. The fields default resolution and type correspond to the resolution and type specified in the data file.

### 3.5.2 abs

### 3.5.3 Standard Unary Functions

`exp`, `sin`, `cos`, `tan`, `log`, `log10`, `sqrt`, `asin`, `acos`, `atan`, `sinh`, `cosh`, `tanh`

### 3.5.4 Self-Made Functions

## 4 Setting Options

### 4.1 nobox

By default, QuantumGL draws a wireframe bounding box around the visualized data. If a statement of the form

```
nobox;
```

is present in the `.qgl` file, the bounding box is not drawn.

### 4.2 colorbox

```
colorbox;  
colorbox <r,g,b>;  
colorbox <rx,gx,bx>, <ry,gy,by>, <rz,gz,bz>;
```

If you use a `colorbox` statement, the bounding box will be drawn in color. If you specify three RGB colors, the first parameter will be used for the edges along the  $x$ -axis, the second for the  $y$ -axis and the third for the  $z$ -axis. If you specify just one color, that will be used for all edges.

Using `colorbox` without parameters is equivalent to:

```
colorbox <1,0,0>, <0,1,0>, <0,0,1>;
```

This means that the edges along the  $x$ -axis will be drawn in red, those along the  $y$ -axis will be drawn in green, and those parallel to the  $z$ -axis in blue.

### 4.3 grid

```
grid n;
```

Draws a grid with  $n$  subdivisions on the faces of the bounding box.

### 4.4 background

You can set the background color using a `background` statement

```
background <red,green,blue>;
```

... where *red*, *green* and *blue* are constant values in the range 0–1.

### 4.5 fog

```
fog [near,far];
```

Objects will start to fade if they are farther than *near* units from the camera. At *far* units, they will be invisible (the same color as the background).

## 4.6 resolution

```
resolution res;
```

Fields will be evaluated at  $res \times res \times res$  data points by default.

## 4.7 world\_size

```
world_size length;  
world_size [minimum, maximum];  
world_size <xdiameter, ydiameter, zdiameter>;  
world_size <[xmin, xmax], [ymin, ymax], [zmin, zmax]>;
```

Defines the default size of the bounding box. If it is not specified, `world_size` `[-1,1]` is assumed.

The `world_size` is automatically expanded to fit all objects you define. Therefore, you need only specify a `world_size` if you wish the bounding box to be larger than the largest of your objects, or if you want to use bounding boxes smaller than the default size of `[-1,1]`.

The following `world_size` statements are equivalent:

```
world_size 2;  
world_size [-1,1];  
world_size <2,2,2>;  
world_size <[-1,1], [-1,1], [-1,1]>;
```

## 5 Field Declarations

```
field identifier := expression;  
field identifier resolution res := expression;
```

The *expression* must be a field expression. The *identifier* must not have been declared. When this statement is encountered in the input, the field expression is evaluated at  $res \times res \times res$  datapoints, or at its default resolution (section 3.1.2 on page 5), if no resolution is specified in the `field` statement.

You can later use the *identifier* to refer to the evaluated field (section 3.3.4 on page 6).

## 6 Animations

```
animate variable_identifier from constant  
                                     to constant [inclusive]  
  [and variable_identifier from constant  
                                     to constant [inclusive]]+  
  in n frames  
  [[and variable_identifier from constant
```

```

to constant [inclusive]]+
in n frames]+;
```

In order to create an animation with QuantumGL, you first add an **animate** statement to your **.qgl** file. Once you have done that, you can view the animation in the QuantumGL application or export it to a series of image files using the “Export Animation...” command. If there is more than one **animate** statement, you will be able to choose which animation to export.

The simplest form of the **animate** statement refers to just one variable (that must already have been declared using the **variable** statement):

```
animate foo from 0 to 1 in 10 frames;
```

This will set the variable **foo** to 0 for the first frame of the animation, to 0.1 for the second and so on, up to 0.9 for the tenth frame. The final value of 1 is never reached; use this form of the **animate** statement to generate an animation that you will play back as a loop.

If you want the last frame to have exactly the final parameter value that you specified, add **inclusive** to the statement:

```
animate foo from 0 to 1 inclusive in 11 frames;
```

The variable **foo** will be 0 for the first frame, 0.1 for the second, 0.9 for the tenth and finally 1.0 for the eleventh.

You can also create an animation that consists of two parts:

```
animate foo from 0 to 1 in 10 frames
and foo from 1 to 2 inclusive in 20 frames;
```

In the above example, **foo** starts at 0, reaches 1 in the eleventh frame, and then “slows down” so that it reaches 2 in the 30th frame.

You can also use two variables at the same time:

```
animate foo from 0 to 1 and bar from 1 to 0 in 10 frames;
```

## 7 Visualization Objects

### 7.1 Isosurfaces

```
isosurface field_expression
at value_expression
[cyclic constant]
[color field_expression]
[cutout <[min_x,max_x],[min_y,max_y],[min_z,max_z]>]
[transparency value_expression]
[shininess value_expression];
```

Renders a surface connecting all points where the field has the value specified after the **at** keyword.

### 7.1.1 Cyclic

When you want to create an isosurface from a field whose values are periodic in nature (angles, for example), then you have to specify the `cyclic` keyword and the period (for example, `2*pi`). Otherwise QuantumGL would think that the field has crossed the threshold value, when in fact it has just “wrapped around” at  $2\pi$  and gone back to zero *without* going through the threshold value.

### 7.1.2 Cutout

You can specify a range of coordinates to be “cut out” from the isosurface (so you can see the interior). The values `min_x`, `max_x`, `min_y`, etc. have to be constants.

## 7.2 Slices

```
slice
    coordinate = value_expression
    [framed]
    [color field_expression]
    [cutout <[min_x,max_x],[min_y,max_y],[min_z,max_z]>]
    [transparency value_expression]
    [shininess value_expression];
```

Renders a cross-section through the data. You can specify either `x`, `y` or `z` as the *coordinate*. If the keyword `framed` is present, a white frame is drawn around the borders of the cross-section.

## 7.3 Vectors

```
vectors field_expression
    [color field_expression]
    [secondary_color field_expression]
    [sphere_color field_expression]
    [sphere_radius field_expression]
    [at point_list]
    [shininess value_expression];
```

Renders one double-cone at each point specified in the `point_list`. One end of the double cone is attached at the point, the length and direction of the double cone is given by the value of the field at that point.

### 7.3.1 Point Lists

#### Explicit List

```
value_expression [, value_expression]_+
```

Each of the expressions is expected to be of type  $\mathbb{R}^3$ , a three-dimensional point.

## Grid

```
grid <x,y,z>, < $\Delta x, \Delta y, \Delta z$ >, <nx,ny,nz>
```

Rather than specifying all coordinates explicitly, you can also use the `grid` keyword. The `grid` keyword will place  $n_x \times n_y \times n_z$  points in a regular, three-dimensional grid. One corner of the grid is specified by  $x$ ,  $y$  and  $z$ ; the points are placed at  $\Delta x$ ,  $\Delta y$  and  $\Delta z$  units from each other in the  $x$ ,  $y$  and  $z$  directions, respectively.

## 7.4 Flow Lines

```
flow field_expression  
    [at point_list]  
    [color field_expression]  
    [length value_expression]  
    [stepsize expression]  
    [shininess value_expression];
```

Renders one “flow line” for each point specified in the `at` clause. The flow lines start in the specified points and follow the direction of the field specified as the first parameter until they either reach a zero of the field or they have reached the maximum length. The maximum length can be specified using the `length` keyword; if omitted, it defaults to 10.

### 7.4.1 Point Lists

You specify point lists for flow lines in the same way as for vectors. See Section [7.3.1](#) for details.

### 7.4.2 Step Size

QuantumGL uses the Runge-Kutta integration method. The direction vectors from the field are normalized before they are used, therefore the magnitude of the field is irrelevant. The default step size is 0.05; you can override it using the `stepsize` keyword.

You can also specify a field as the argument of the `stepsize` clause in order to achieve a variable stepsize. This might be useful if you want highly curved flowlines in one part of the image while saving computing power in others.

## Part II

# Data Files

Using the predefined function `read(“filename”)`, you can read in data generated by other programs. QuantumGL currently supports two simple data formats,

one plain-text data format, and one binary data format. The plain-text format is probably easier to generate in most situations, and it is more portable<sup>3</sup>. The binary format is substantially faster and more compact.

## 8 ASCII Data Files

A QuantumGL ASCII data file consists of numbers separated by whitespace (whitespace means spaces, tabs or linefeeds). It doesn't matter if you put several numbers in one line, or if you put all numbers in lines of their own, all that counts is that there is at least one space, tab or linefeed in between.

- The first number has to be either 0 or 1. A value of 0 means that it's a real-valued field, a value of 1 means it's complex.
- The second number indicates the number of components of each vector. Use 1 for scalar fields.
- Next, there are three numbers indicating the number of data points in  $x$ ,  $y$  and  $z$  directions.
- Finally, for every data point, a vector is stored.

A complex number is stored as two consecutive real numbers. A real vector is stored as  $n$  consecutive real numbers (real part and imaginary part). A complex vector is stored as  $n$  consecutive complex numbers (pairs of real numbers).

## 9 Binary Data Files

A QuantumGL binary data file starts with a twelve-byte header structure shown in Table 1.

After that, there's one vector per data point. A vector consists of  $n$  consecutive real or complex values. A complex value consists of two consecutive real values. A real value is stored as a big-endian 32-bit IEEE-754 floating point value.

## Part III

# Extending QuantumGL

With basic knowledge of C and just a few bits of C++, you can extend QuantumGL by adding your own field-valued functions. This is both faster and more convenient than using a separate program to generate big data files.

---

<sup>3</sup>Binary data files can be exchanged between Intel- and PowerPC-based computers, and with other computers that use the IEEE-754 floating point format. "Exotic" platforms might cause trouble, though.

Size	Description	Format
4 bytes	magic number	ASCII codes for 'B', 'i', 'n', 'F' (for <b>Binary Format</b> )
1 byte	real/complex indicator	ASCII 'C' (0x43) for complex, 'R' (0x52) for real
1 byte	number of components per vector	unsigned one-byte integer
2 bytes	number of data points in x direction	unsigned two-byte integer, big-endian
2 bytes	number of data points in y direction	unsigned two-byte integer, big-endian
2 bytes	number of data points in z direction	unsigned two-byte integer, big-endian
12 bytes	total size	

Table 1: Header for Binary Data Files

By writing a small piece of C++ code, you can define functions that can be used in the Visualization Task File. These functions can take both fields and real numbers as arguments. The implementation currently requires that the fields come first, i.e. the function can take zero or more field arguments followed by zero or more real arguments. The total number of arguments is currently limited to seven, but this limit can be increased easily.

As an example, we will describe how to write a new color function that uses a blue-red-gradient to visualize scalar values. We will write a function that takes a real scalar field, a (real) minimum value and a (real) maximum value, and returns a field of  $\mathbb{R}^3$  vectors (which will be interpreted as RGB colors). The function will map the minimum value (and everything below it) to blue and the maximum value (and everything above it) to red.

The function therefore has the following type:

$$((\mathbb{R}^3 \mapsto \mathbb{R}) \times \mathbb{R} \times \mathbb{R}) \mapsto (\mathbb{R}^3 \mapsto \mathbb{R}^3) \quad (1)$$

In C++, however, it is difficult to create a function of this type, because it is difficult to *dynamically* create a new function (we want to return a field, which is a function  $\mathbb{R}^3 \mapsto \mathbb{R}^3$ ). Instead, we will create an equivalent function that evaluates the resulting field at one point and returns just a plain vector. The type of this new function is

$$(\mathbb{R}^3 \times (\mathbb{R}^3 \mapsto \mathbb{R}) \times \mathbb{R} \times \mathbb{R}) \mapsto \mathbb{R}^3 \quad (2)$$

If we later want to evaluate the resulting field, we just have to call the function repeatedly, once for each point. QuantumGL does this automatically.

## 10 Predefined Types

QuantumGL defines a set of C++ types (available in the header file “FieldFunction.h”)

`Real` a real (floating-point) number. Use this instead of `float` or `double`<sup>4</sup> ( $\mathbb{R}$ ).

`Complex` a complex number ( $\mathbb{C}$ ).

`vec<Real,n>` a vector with  $n$  real components<sup>5</sup> ( $\mathbb{R}^n$ ).

`vec<Complex,n>` a vector with  $n$  complex components ( $\mathbb{C}^n$ ).

`r1field` a real-valued scalar field ( $\mathbb{R}^3 \mapsto \mathbb{R}$ ).

`c1field` a complex-valued scalar field ( $\mathbb{R}^3 \mapsto \mathbb{C}$ ).

`rvecfield<n>` a real vector field with  $n$  components ( $\mathbb{R}^3 \mapsto \mathbb{R}^n$ ).

`cvecfield<n>` a complex vector field with  $n$  components ( $\mathbb{R}^3 \mapsto \mathbb{C}^n$ ).

So if we write the type (2) in C++, we get the following prototype for our function in C++:

```
vec<Real,3> bluered(vec<Real,3> pos,
                  r1field f,
                  Real minValue,
                  Real maxValue);
```

## 10.1 Operations on Scalar Values

`Real` and `Complex` in QuantumGL are just synonyms<sup>6</sup> for `float` and `complex<float>`, so you can use all the basic arithmetic operators and the functions defined in the C++ standard library. Consult your favourite C++ textbook.

Do not use `float` or `double` instead of `Real`, because in future versions, `Real` might be defined to `double` instead. Also, always use `Complex` rather than `complex<float>` or `complex<double>`.

Instead of using a normal floating point literal like 0.5, you might want to write `Real(0.5)` to avoid problems with C++'s type system<sup>7</sup>.

If you want a complex constant or if you want to build a value of type `Complex` from two `Real` values, write `Complex(x,y)` where  $x$  is the real part and  $y$  the complex part.

---

<sup>4</sup>currently, it is defined to be `float`. This might be changed in future versions of QuantumGL.

<sup>5</sup> $n$  has to be an integer constant — the dimension of a vector has to be known at compile time

<sup>6</sup>defined using `typedef`

<sup>7</sup>A plain floating point literal like 0.5 is considered to be of type `double`. Thus, when working with variables of type `Real` or `float` and constants of type `double`, C++'s implicit type conversions are used. This can lead to error messages that are hard to understand for C++ beginners.

## 10.2 Operations on Vectors

You can create a real vector (type `vec<Real,n>`) by calling the function `makeVecR()` with  $n$  parameters (of type `Real`). Likewise, you can use `makeVecC()` to create complex vectors (type `vec<Complex,n>`).

You can access the  $n$ th component of a vector using the `[]` operator, that is `v[1]` is the first component of the vector `v`.

The basic arithmetic operators are defined (“overloaded”) for vectors:

+ Addition

- Subtraction

\* Multiplication by a scalar value. Both *scalar\*vector* and *vector\*scalar* are allowed.

/ Division by a scalar value.

The function `vdot(a,b)` calculates the inner product of  $a$  and  $b$ , and the function `abs(x)` returns the absolute value of a vector.

## 10.3 Operations on Fields

You can access the value of a field at any point in space using parentheses: `myField(pos)` is the value of the field `myField` at the position `pos`. The position must be a value of type `vec<Real,3>`.

You cannot explicitly create a field yourself. You can only get them as parameters for your function. That’s why we write our function in such a way that we do not explicitly return a field.

## 11 How to add a Field Function

When you add a new Field Function, you have to modify three different files: One source file, one header file, and the file `FieldFunctionRegistry.cp`.

### 11.1 The Header File

The header file contains a so-called *prototype* for our function; it just tells the C++ compiler that a function with that name and that type is defined elsewhere. Note that you can put prototypes for more than one function in the same header file. A header file should have a file name that ends in “.h”. For our “bluered” example, we can create a text file named “BlueRed.h”, with the following contents:

```
#include "FieldFunction.h"

namespace field
```

```

{
    vec<Real,3> bluered(vec<Real,3> pos,
                      r1field f,
                      Real minValue,
                      Real maxValue);
}

```

## 11.2 The Source File

Next, we need the source file, where we will put in the actual code. The source file should have the same name as the header file, but end in “.cp”<sup>8</sup> instead of “.h”. The source file has to include the header file at the top. For our example, it could look like this:

```

#include "BlueRed.h"

vec<Real,3> field::bluered(vec<Real,3> pos,
                        r1field f,
                        Real minValue,
                        Real maxValue)
{
    Real t = (f(pos) - minValue) / (maxValue - minValue);
    if(t < 0)
        t = 0;
    if(t > 1)
        t = 1;
    return makeVecR(t,1-t,0);
}

```

## 11.3 The Field Function Registry

To make our example Field Function “bluered” known to QuantumGL, we have to add one line to the function `RegisterFieldFunctions` in the file `FieldFunctionRegistry.cp`:

```
RegisterFieldFunction("bluered",&field::bluered);
```

At the top of the same file, we have to add an `#include` statement so that the compiler can see the prototype from our header file:

```
#include "BlueRed.h"
```

---

<sup>8</sup>The extension “.cp” for C++ source files is not very widely used — it’s an old Macintosh tradition. You can also use “.cc”, “.cpp”, “.cxx”, or “.C”. Do *not* use “.c”, as this would mean “plain” C, not C++.