GAViewer Documentation
Version 0.84

Daniel Fontijne
University of Amsterdam

March 19, 2010

Contents

Chapter 1

Introduction

GAViewer is a multi-purpose program for performing geometric algebra computations and visualizing geometric algebra. Some possible uses are:

- Visualizing geometric algebra output from other programs.
- Interactively performing GA computations and visualizing the outcome.
- Presenting lectures, slideshows, doing tutorials, demonstrations, with (interactive) geometric algebra animations.
- Rendering (hi-res) images of GA objects for use in papers.
- Debugging other programs that use geometric algebra.

We do not consider GAViewer appropriate for implementing 'serious' applications. The internal (interpreted) programming language is too slow and limited for such purposes.

GAViewer has outgrown its original purpose. We initially created GAViewer as a small program for visualizing GABLE / Matlab output because we weren't satisfied with the Matlab graphics. Then we wanted to have a typesetting system for labels and support for slideshows. After some time the desire rose to add a console for interactive computations inside the viewer. After a console was added, we wanted to have functions and batches. Then dynamic statements were added. The latest additions include animations based an dynamic statements and scalar controls.

In the following chapters, the various features of GAViewer are described in the following order:

- The user interface.
- Visualizing geometric algebra output from other programs using .geo files.
- The programming language.
- Typesetting labels.
- Using .geo files for slideshows and presentations.

Chapter 2

The user interface

When you start GAViewer, it should look something like figure 2.1. At the top of the window, there is a standard menu bar, desribed in section 2.5. The largest part of window is occupied the *view window* (section 2.1). At the bottom, there is the *console* (section 2.2). The right part of the window is split into the *object controls* (section 2.3) and the *scalar controls* (section 2.4).

Finally, at the very bottom of the window there is the *status bar*, and the *pause* button, which is useful during presentations.

2.1 The view window

In the view window, all currently visible objects are drawn. You can rotate and translate your viewpoint, select objects and translate most types of objects. All this is done using certain combinations of mouse movements, mouse clicks / drags and the **ctrl** button, as explained below.

First, let's create some objects such that the view window isn't empty. On the console, type

```
>> a = no, b = green(e1), c = e2 ^ e3
```

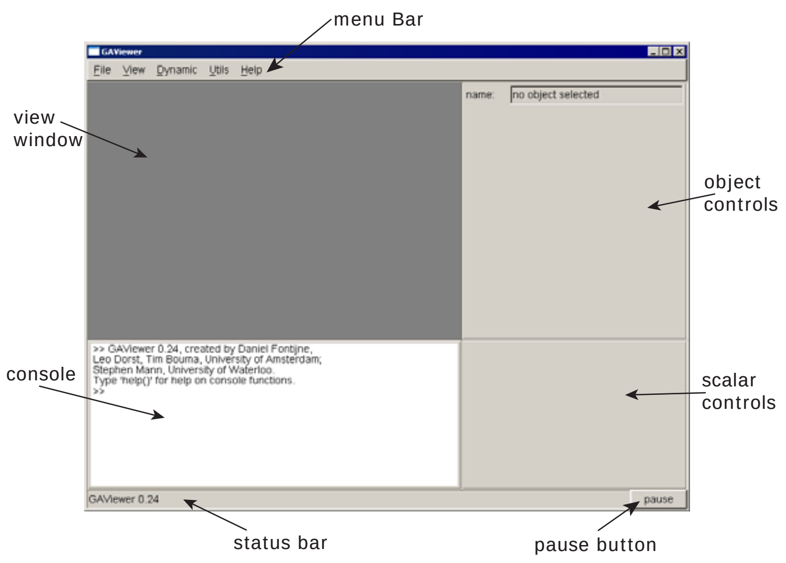
This draws a flat shaded red point **a**, a green vector **b** and a blue bivector (disc) **c**.

2.1.1 Viewpoint rotation

To rotate the your viewpoint, hold the left mouse button down while the mouse is inside the viewpoint, and move the mouse. Let's call this **left mouse drag** for short. The view window has a so-called *spaceball interface*. This means that if you **left mouse drag** in the center of the view window, the viewpoint will rotate about an axis in the screen plane, and if you **left mouse drag** outside the center, the viewpoint will rotate about the axis perpendicular to the screen plane.

2.1.2 Viewpoint translation

Viewpoint translation is a lot simpler than rotation. **Middle mouse drag** will translate your viewpoint parallel to the screen plane, **right mouse drag** will



name: no object selected

```
>> GAViewer 0.24, created by Daniel Fontijne,
Leo Dorst, Tim Bouma, University of Amsterdam;
Stephen Mann, University of Waterloo.
Type 'help()' for help on console functions.
>>
```

GAViewer 0.24

Figure 2.1: GAViewer user interface.

translate perpendicular to the screen plane.

2.1.3 Object selection

Hold down the **ctrl** button and **left mouse click** on one of the objects in the view window. This selects the object and shows information and controls in the object controls window on the right.

If an object is hidden behind another object, you can still select it by *cycling* through the objects. For instance, **ctrl left mouse click** the red point **a** in the center of the disc **c**. This will select **a**. Now don't move the mouse, and **ctrl left mouse click** again at exactly the same location. This will select **c** or **b**. **ctrl left mouse click** again to continue cycling until you have selected the object you want.

2.1.4 Object selection

Sometimes, you may want to select an object and also have it's name on the console, to use it in some calculation. This can be achieved by **ctrl middle mouse clicking** the object.

2.1.5 Object translation / modification

Many objects allow some kind of translation / modification to be performed on them by **ctrl right mouse dragging**. For example, the red point **a** can be translated, as can the tip of the green vector **b**. The blue bivector **c** can not be translated, but you can modify its size by **ctrl right mouse dragging**.

2.1.6 View window control summary

rotate	**left mouse drag**
translate	**middle mouse drag** and **right mouse drag**
select	**ctrl left mouse click**
select, copy name to console	**ctrl middle mouse click**
translate / modify	**ctrl right mouse drag**

2.2 The console

The console is used to do interactive computations. On the console, you can write geometric algebra expressions and a lot more, as described in the chapter 4. Here, we describe only the user interface of the console.

To type something on the console, click somewhere after the current prompt (>>) to place the cursor where you want it. Then type away. If you click before the current prompt, the first character you type will be lost and the cursor will jump to the end of the current input.

You can select text by dragging the mouse, or holding **shift** and using the arrow keys. Copy selected text with **ctrl-c**. You can paste text using **ctrl-v**. Cut using **ctrl-x**.

To jump from one bracket to the first one matching it, use **ctrl-e**. For example, type something like:

```
>> a = ((c . b) x[3 i]) ^ y
```

Locate the cursor at any of the brackets, and press **ctrl-e**. The cursor will jump to the matching bracket.

To execute what you have typed, press enter.

To retrieve previous commands, press the **up arrow**, use the **down arrow** to get back again.

The console is based on the FLTK FL_Text_Editor widget, so if you want to know more details, see the FLTK documentation.

2.3 Object controls window

In this section, we assume you have executed the following line from the previous section:

```
>> a = no, b = green(e1), c = e2 ^ e3
```

You can execute it again if you want the original objects back.

When you **ctrl left click** one of the objects in the view window, it becomes the *current object* and you can control it somewhat using the object controls window on the right of GAViewer (see Figure 2.1).

At the top of the window, you see the name of the current object. The two buttons below can be used to remove or hide that object.

For most objects, you can set the foreground color and alpha (opacity) using the **set foreground color** and **alpha** widgets. Some objects, like labels, also have an outline or background color.

The checkboxes in the middle of the window can be used to make some visual distictions between objects. E.g., you could **stipple** imaginary objects, or turn off the shading to indicate flatness.

Certain objects can be drawn in multiple ways. For instance, select the blue bivector **c**. It has a **draw method** pull down menu from which you can choose various ways of drawing a bivector.

At the very bottom of the object controls window is a text field that shows:

- The interpretation of the object.
- Some numerical properties that are used to draw the object.
- The coordinates of the object (with limited precision).

When the object controls window is hidden (menu **View**$\rightarrow$**Controls**), you can still see a 'condensed' version of this information in the status bar.

2.4 Scalar controls window

You can create a scalar control on the console like this:

```
>> ctrl_range(a = 2.0, 0.5, 10.0)
```

It will appear in the lower right corner of GAViewer.

Scalar controls are explained in 4.9.5. For the scalar controls window to be visible, the console or the object controls window must be visible.

2.5 The menu bar

The menu bar has the following structure:

- **File**
  - **Open**: allows you to open any type of file.
  - **Open**$\rightarrow$
    - **Open geo file**: opens a *.geo file*
    - **Open geo playlist**: opens a *.gpl file*
    - **Open g file**: opens a *.g file*
  - **Load .g directory**: allows you read an entire directory full of *.g* files in one run. Will also read subdirectories.
  - **Save state**: save the current state of the GAViewer into a *.geo* file.
  - **Replay**: replays the current *.geo* file.
  - **Next file in playlist**: switches to the next *.geo* file in the current playlist.
  - **Previous file in playlist**: switches to the previous *.geo* file in the current playlist.
  - **Exit**: terminates GAViewer.
- **View**

- **Select object**: pops up a dialog that allows you to select any object.
- **Hide**: allows interaction with hidden objects.
    - **Unhide all**: shows all hidden objects.
    - **Select hidden object**: pops up a dialog that allows you to select a hidden object.
    - **Show hidden object**: pops up a dialog that allows to toggle the hide / show state of hidden objects.
- **Canvas**: selects the color of the canvas (white, (light) grey or black).
- **Console font size**: selects the font size used on the console.
- **Controls**: toggles whether the object controls window is visible.
- **Scalar Controls**: toggles whether the scalar controls window is visible.
- **Console**: toggles whether the console is visible.
- **Labels always on top**: when on, labels will always be drawn on top of other objects.
- **Fullscreen**: toggles fullscreen / windowed user interface. In fullscreen mode, only the view window will be visible.

- **Dynamic**: contains dynamic statement and animation related items.
    - **View Dynamic statements**: pops up a dialog where you can view / modify the current dynamic statements (see section 4.10).
    - **Start / resume animation**: starts the **atime** variable (see section 4.10.2).
    - **Pause animation**: pauses the **atime** variable.
    - **Stop animation**: stops the **atime** variable.
    - **Playback speed**: controls how fast animations play.

- **Utils**
    - **Search for next bookmark**: goes to the next bookmark in the current *.geo* file, if any.
    - **Output current camera orientation (bivector)**: prints the current camera (viewpoint) orientation (**camori**) to the console, in bivector form.
    - **Output current camera orientation (rotor)**: same as above, but in rotor form.
    - **Output current camera translation**: prints camera the translation (**campos**) to the console.
    - **Screenshot**: pops up a dialog that allows you to renders a screenshot of the view window in arbitrary resolution. The file is stored in the *.png* file format.

- **Help**
    - **About**: displays some info about GAViewer.

Chapter 3

.geo and .gpl files

.geo files are one form of input that GAViewer can handle. The others – which are more interesting to casual users – are the console and *.g* files (see the next chapter). *.geo* files are harder to write by hand than *.g* file, but can be very useful for presentations and displaying output from other programs. The *.geo* file format is also used to store the state of the GAViewer (menu **File**→**Save state**).

The *.geo* file format can do some things that the *.g* 'programming language' cannot and vice versa. This has grown historically. If we were to redesign GAViewer we would choose one unified, richer, programming language able to do everything.

Every line in a *.geo* file has the following format:

```
keyword arguments
```

The keyword can be something like **fgcolor**, **e3ga** or **delete**. The arguments vary per keyword. You can not split keyword and arguments over multiple line. Everything following a '#' sign is considered comment (unless the #' is somewhere inside a quoted string):

```
#this is comment
label label_1 [1.0*e1+1.0*e2] "a quoted string with a # inside"
```

We now describe every keyword and its arguments. At the end of this list there is one entry **play** which is valid only for *.gpl* files. These are 'geo playlists' and can be used to play multiple *.geo* in a sequence. This is useful for presentations.

3.1 title

Sets the title bar of the viewer window. Usage example:

```
title "this is my demo"
```

Sets the title bar of the window to 'GAViewer: this is my demo'

3.2 fgcolor, bgcolor, olcolor, cvcolor

Set the current foreground, background outline and canvas color. All objects read after these keywords will be drawn in those colors. The canvas color affects only the canvas (background) of the viewing window.

As argument you can either supply a color name or a RGB value with optional alpha (transparency). Possible color names are:

- ”r” or ”red”
- ”g” or ”green”
- ”b” or ”blue”
- ”w” or ”white”
- ”gray” or ”grey”
- ”k” or ”black”
- ”c” or ”cyan”
- ”m” or ”magenta”
- ”y” or ”yellow”

Usage examples:

```
fgcolor red             # set foreground color to red
cvcolor 1.0 0.5 0.1     # set canvas color to orange
olcolor 0.0 0.0 0.0 0.5 # set outline color to semi-transparent black
```

The range for the RGBA values is [0.0 1.0].

3.3 e3ga, ca3d, p3ga, ca4d, c3ga, ca5d

Draws an object from the euclidean (**e3ga**, **ca3d**), projective **p3ga**, **ca4d**), conformal model (**c3ga**, **ca5d**) of 3D Euclidean Geometry. The names **caNd** mean ’Clifford Algebra N Dimensional’. **e3ga** means Euclidean 3D Geometric Algebra. **p3ga** means Projective 3D Geometric Algebra. **c3ga** means Conformal 3D Geometric Algebra. These algebras are all considered 3D since they are interpreted as performing 3D geometry. The general syntax is:

```
c3ga "object name" [multivectorCoordinates] flag1 flag2 flagn
```

The multivector coordinates must be in the format that the Gaigen multivector parser can understand.

The ”object name” can be a quoted string or a string not containing any spaces. A maximum of 8 flags is allowed. The flags can alter certain properties of the object. Possible flags are:

- **hide**: immediately hides the object.
- **show**: immediately draws the object (default).

- **stipple**: draws the object stippled.
- **orientation**: draws something related to the orientation of the object, if possible.
- **wireframe**: draws the object in wireframe, if possible.
- **magnitude**: draws the magnitude (called **weight** in the GAViewer UI) of the object, if possible.
- **shade**: shades the object, if possible.
- **versor**: force versor interpretation of the multivector (e.g. to interpret a blade like a versor).
- **blade**: force blade interpretation of the multivector (e.g. to interpret '0' as a blade).
- **grade0** ... **grade8**: force gradeX interpretation of multivector (e.g. to interpret '0' as a vector).
- **dm1** ... **dm7**: use draw method 1 to 7, if supported by the object. Some object can be drawn in multiple ways. The default draw method is '1'.

Usage examples:

```
c3ga "the arbitrary origin" [no]
e3ga stippled_vector [e1+e2+e3] stipple
e3ga z [4.0*e3^e1] magnitude orientation dm2
```

3.4 label

Adds a label to the scene. The syntax is:

```
label "name" "point" "text" flag1 flag2 ... flagn
```

Draws a label (in current colors and font siz, and other typesetting parameters) at position 'point'. 'point' can be any previously specified multivector object which has a point interpretation, or a 3D multivector coordinates like **[1.0*e1 + 2.0*e3]**.

The flags can be some of the following:

- **2d**: label coordinates are in 2D window coordinates.
- **3d**: label coordinates are in 3D world coordinates (default).
- **cx**: x-axis origin is in center of window (only in combination with 2D).
- **cy**: y-axis origin is in center of window (only in combination with 2D).
- **px**: positive x axis is towards the right (only in combination with 2D).
- **nx**: positive x axis is towards the left (only in combination with 2D).
- **py**: positive y axis is towards the bottom (only in combination with 2D).

- **ny**: positive y axis is towards the top (only in combination with 2D).
- **acx**: the label is x-aligned in the center of the window (overrides all other commands related to the 'X'-axis, only in combination with **2D**).
- **dynamic**: the position of the label will follow the multivector object 'point'.
- **image**: the label text is actually a filename of a *.png* image that will be displayed inside the label.
- **fullscreen**: scales the images such that it fill the viewer / screen (only in combination with **image**) (image width / height proportion is not fixed).

Usage examples:

```
label simple [1.0*e1] "a simple label"
label attached_to_z z "this label follows 'z'" dynamic
label fullscreen_image [0] "c:\images\dog.png" 2d image fullscreen px py
```

3.5 fontsize

Sets the size of the font for the labels in pixels. Usage example:

```
fontsize 30.0
```

3.6 tsmode

Sets the initial parsing mode of the typesetting system for labels. See chapter 5 for more details on typesetting. The mode can be any of: **text**, **equation**, **verbatim**, uppercase or lowercase. In fact, only the first character of the string is used to determine the mode. Verbatim mode bypasses the whole typesetting system and displays labels using the regular ASCII characters directly.

Usage example:

```
tsmode equation
```

3.7 tsfont

Sets the initial font of the typesetting system for labels See chapter 5 for more details on typesetting. The font can be any of: **regular**, **bold**, **italic**, **greek**, uppercase or lowercase. In fact, only the first character of the string is used to determine the font.

Usage example:

```
tsfont italic
```

3.8 tsreset

Resets the typesetting system to its initial mode. See chapter 5 for more details on typesetting.

Usage example:

```
tsreset
```

3.9 tsinterpret

Sends text to the typesetting system. See chapter 5 for more details on typesetting. It is then parsed and interpreted, but not displayed. This is useful for adding custom commands and colors to the typesetting system.

Usage example:

```
tsinterpret "some string"
```

Sends ”some string” to the typesetting system. The typesettings system mode (as set with **tsmode**) is always forced to **text** during a **tsinterpret**!

3.10 factor

Specifies a custom factor for factorization. These are used during the interpretation of some multivectors. (currently only the e3ga bivector and trivector) Syntax:

```
factor model idx [vectorCoordinates]
```

’model’ specifies for which model this factor is intended (**c3ga**, **ca5d**), (**p3ga**, **ca4d**), (**e3ga**, **ca3d**). ’idx’ specifies the index of the factor [1 ... d] (d = dimension of the model).

Usage examples:

```
factor e3ga 1 [1.0*e1]
factor e3ga 2 [1.0*e2]
factor e3ga 3 [1.0*e3]
```

3.11 eyepos, campos

Sets the position of the eye/camera Usage example:

```
campos [10.0*e3]
```

3.12 eyetrl, camtrl

Translates the eye/camera over a specified vector per second during a specified time. Usage example:

```
camtrl 10.0 [1.0*e3]
```

The first argument is the duration, the second the translation vector. If the duration is 0, the translation in instantanious.

3.13 eyeori, camori

Sets the orientation of the eye/camera Usage example:

```
camori [1.0*e1^e2]
```

The coordinates specify a bivector that will be exponentiated to create a rotor.

3.14 eyerot, camrot

Rotates the eye / camera over a specified plane / angle per second during a specified time. Syntax: Usage example:

```
camrot 10.0 [1.0*e1^e2]
```

The first argument is the duration, the second the rotation bivector. If the duration is 0, the rotation in instantanious.

3.15 hide, show

Hides or shows a specified object (can be a label, algebra object, polygon, etc) or user interface element.

Usage examples:

```
show "name of object"
hide "name of object"
```

The user interface elements that can affected by **hide** and **show** are:

- **controls**: object controls window.
- **scalar_controls**: scalar controls window.
- **console**: the console.

3.16 remove

Removes the specified object. Usage example:

```
remove x
```

3.17 fade, fade_and_remove, fade_and_hide, show_and_fade

The keywords allow you to fade in and out objects. Before the fade, the object can be shown (**show_and_fade**). After the fade is over, the object can be hidden (**fade_and_hide**) or removed (**fade_and_remove**). The syntax is:

```
fade "object name" fade_duration fade_target fade_start
```

The first argument is the name of the object. The second argument is the duration of the fade in seconds. The third argument is the target alpha of the fade. The fourth, optional argument is the alpha at the start of the fade. Using fade will not actually modify the alpha of the any of the colors of the object, but rather multiplies those alpha values before they are sent to OpenGL.

Usage examples:

```
fade x 2.0 1.0
fade_and_remove y 1.0 0.0 1.0
```

3.18 sleep

Pauses the reading of the input file for a specified number of seconds. User interface will be fully functional during this sleep.

Usage examples:

```
sleep 10.0
```

Sleeps for 10.0 seconds. The maximum resolution for the sleep time is about 1/30th of a second.

3.19 wait

Pauses the reading of the input file until the **waiting** button is pressed. Usage example:

```
wait
```

3.20 exit

Terminates the GAViewer immediately. Usage example:

```
exit
```

3.21 clearconsole

Clears the console and removes scalar controls. Usage example:

```
clearconsole
```

3.22 console

console allows you to execute a command in a *.geo* as if it was typed on the console. Usage example:

```
console a = e1 ^ e2
```

3.23 resize

Changes the size and optionally the position of the GAViewer window. Syntax:

```
resize w h
resize x y w h
```

The first format (with 2 arguments) changes the width and height of the window to **w** and **h**. The second format (with 4 arguments) also sets the position to **x** and **y**.

3.24 viewwindowsize

Changes the size of the view window. This will resize the main window and keep the height of the console and the width of the controls constant. Usage example:

```
viewwindowsize 1024 768
```

3.25 consoleheight

Changes the height of the console. This will resize the console and the view window to achieve the desired height. Usage example:

```
consoleheight 10 lines
consoleheight 200 pixels
```

You must specify eiter **pixels** or **lines**.

3.26 consolefontsize

Changes the size (in pixels) of the font used on the console. Usage example:

```
consolefontsize 14
```

3.27 fullscreen

Sets the viewer to fullscreen mode or windowed mode. Only the view window is visible in full screen mode. Usage examples:

```
fullscreen
fullscreen on
fullscreen off
```

The first two lines turn fullscreen mode on, the second line turns it off. In fullscreen mode, a small red **W** may be visible in the lower right corner when GAViewer would normally flash the **waiting** button.

3.28 bookmark

Indicates a bookmark in the file. When the user selects menu bar item **utils**→**search for next bookmark**, input will be parsed quickly until such a bookmark is found. This is useful for skipping through a (slow) demo quickly. Usage example:

```
bookmark "optional name that is not used yet"
```

3.29 open, switchto, import

These keywords all open a *.geo* file. GAViewer maintains an internal stack of open files.

- **open** opens a new file at the top level. All current files are removed from the stack and the new file becomes the only open file.
- **switchto** closes the top-level file and replaces it with the new file.
- **import** pushes the new file on top of the file stack and starts reading it

You can give argument to these commands which will be available as $1, $2, etc, in the files. Usage examples:

```
switchto file1.geo
import conformal_paraboload.geo paraboloid
open matlab.geo
```

The second example give the argument 'paraboloid' to *conformal_paraboload.geo*. Any occurence of $1 in *conformal_paraboload.geo* will be replaced with 'paraboloid'.

3.30 clip

Sets the distance of the clipping planes to the origin. Currently not functional. Usage examples:

```
clip 10.0
```

3.31 delete

Specifies whether to delete this *.geo* file or not when a new file is opened or GAViewer is terminated. This used to be useful when GAViewer was only used to visualize Matlab output, where *.geo* files were usually just a temporary communication channel. Syntax:

```
delete [yes|no|ask] "question to ask"
```

If the first argument is 'ask' the user is asked whether to delete or not. The question to ask can be supplied as the optional second argument. If you use 'by the name of the file. Usage examples:

```
delete yes
delete ask "delete %s?"
```

3.32 polygon, simplex

Creates a polygon or simplex object. Syntax:

```
polygon "polygon name" nb "p 1" "p 2" "p n" flag1 flag2 flagn
simplex "simplex name" nb "p 1" "p 2" "p n" flag1 flag2 flagn
```

The first argument is the name of the object. The second the number of vertices, followed by a name of a ’point’ for every vertex. After that, a number of flags can be added. The maximum number of vertices is 3 for a simplex. Polygons must be convex, or the resulting graphics will be unpredicatble. The ”p 1” ... ”p n” are names of objects that have some kind of point interpretation. E.g., they can be vectors in the 3D model, or points in the conformal model.

Possible flags:

- **dynamic**: the vertices of the polygon will lookup their position from the original point objects everytime the polygon gets redrawn.
- **outline**: draws an outline around the polygon.
- **dm1** ... **dm7**: use draw method 1 to 7 (**dm1**: filled, **dm2**: line strip, **dm3**: line loop, **dm4**: 1D simplex is drawn as true vector).

Usage example:

```
polygon "P1 -> Q2" 2 P1 Q2 dm2
```

3.33 mesh

This keyword allows for the creation of a mesh object. A mesh consists of a number of vertices (with optional surface normals) and polygons. The vertices, surface normals and polygons are specified after the mesh. The syntax of **mesh** is:

```
mesh "mesh name" normal_flag
```

’normal_flag’ can be

- **compute_normals_flat**: compute the surface normals such that the object will have a flat shaded appearance.
- **compute_normals_gouraud**: compute the surface normals such that the object will have a smooth (Gouraud) shaded appearance.
- **specify_normals**: specify normals in the file.

The **mesh** must be followed by its vertices, normals and faces, described below.

Usage example (a full example is given after the **meshvertex**, **meshnormal** and **meshface** have been described:

```
mesh teapot compute_normals_gouraud
```

3.33.1 meshvertex

The syntax of **meshvertex** is:

```
meshvertex "mesh name" index point
```

The mesh name refers to the mesh name given in an earlier **mesh** keyword. **index** is the positive index of the vertex in the list of vertices. **point** can be the name of an existing object with a point interpretation, or the 3D coordinates of the point between square brackets. Usage example:

```
meshvertex teapot 0 [1.0e1 + 1.0*e3]
```

3.33.2 meshnormal

meshnormal allows you to specify the surface normal at a vertex. The syntax of **meshnormal** is:

```
meshnormal "mesh name" index vector
```

The mesh name refers to the mesh name given earlier in a **mesh** keyword. **index** is the positive index of the vertex in the list of vertices. **point** can be the name of an existing object with a vector interpretation, or the 3D coordinates of the vector (not a bivector.... :) between square brackets. Usage example:

```
meshnormal teapot 0 [1.0e1 + 1.0*e3]
```

3.33.3 meshface

meshface specifies a face of a mesh. It can have an arbitrary (current max 16) number of vertices. Vertices should be listed in counter clockwise order, when viewed from the front side. The syntax is:

```
meshface "mesh name" vertex_idx1 vertex_idx2 ... vertex_idxN
```

Usage example:

```
meshface teapot 295 327 328
```

3.33.4 Full usage example for mesh

This example should draw a cube (you may have to zoom out to see it, if these are the only commands in a *.geo* file. Usage example:

```
mesh cube compute_normals_flat
meshvertex cube 0 [1.0*e1+-1.0*e2+1.0*e3]
meshvertex cube 1 [1.0*e1+-1.0*e2+-1.0*e3]
meshvertex cube 2 [-1.0*e1+-1.0*e2+-1.0*e3]
meshvertex cube 3 [-1.0*e1+-1.0*e2+1.0*e3]
meshvertex cube 4 [1.0*e1+1.0*e2+1.0*e3]
meshvertex cube 5 [1.0*e1+1.0*e2+-1.0*e3]
meshvertex cube 6 [-1.0*e1+1.0*e2+-1.0*e3]
meshvertex cube 7 [-1.0*e1+1.0*e2+1.0*e3]
meshface cube 3 2 1 0
meshface cube 0 1 5 4
meshface cube 0 4 7 3
meshface cube 5 1 2 6
meshface cube 6 2 3 7
meshface cube 4 5 6 7
```

3.34 play (.gpl files only)

.gpl files are very special *.geo* files that can contain only one type of keyword: **play**. The syntax of **play** is:

```
play filename.geo arg1 arg2 ... argN
```

The arguments are optional and will replace $1, $2 ... $N. A playlist for a presentation could look like this:

```
play ppt/ppt.geo ppt_01_title.png

play ppt/ppt.geo ppt_02_overview_1.png
play ppt/ppt.geo ppt_02_overview_2.png
play ppt/ppt.geo ppt_02_overview_3.png
play ppt/ppt.geo ppt_02_overview_4.png

#block 1
play ppt/ppt.geo ppt_03_block1.png

play demos/crossproduct.geo
play demos/outerproduct.geo
play demos/trivector.geo
play demos/basiselements.geo
...
```

Chapter 4

The Programming Language and the Console

GAViewer contains a small internal programming language. It is basically a C-like language, with features like functions and conditional control structures, global and local scopes. It has a very limited set of types (3D Euclidean multivectors, 4D homogeneous multivectors, 5D conformal multivectors). These types can be automatically coerced. Functions can be overloaded. There is limited support for arrays. Dynamic statements allow for quite amazing flexibility. Dynamic statements *depend* on the variables used to evaluate them. Every time such a variable changes, the dynamic statement is reevaluated.

The programming language can be used on the console, and in .g files. These files typically contain functions and batches.

A large number of built in functions are provided to handle all kinds of typical GA operations.

4.1 Comma, semicolon and space

As in Matlab, the symbol used to terminate a statement determines if the result is shown on the console / view window. Typing

```
>> a = e1,
```

will pop up a vector in the view window and show you the coordinates of **a** on the console. On the other hand, typing

```
>> a = e1;
```

hides the vector and does not print its coordinates. The multivector **a** still exists, but it is simply not shown. You can use the menu **view**→**hide**→**show hidden object** to make it appear again. Using no symbol to end a statement is equivalent to a comma:

```
>> a = e1
```

This will again pop up a vector in the view window and show you the coordinates of a on the console. In *.g* files, you must always terminate all statements with either a semicolon or a comma.

4.2 ans

When you type **e1** on the console:

```
>> e1
ans = 1.00*e1
```

you'll see that the value of the statement **e1** gets assigned to **ans**.

When you enter a statement on the console that's (implicitly) terminated with a comma, every variable that was assigned a value is displayed on the console and in the view window. If no assignments were made, the result of the statement is assigned to the **ans** variable.

When you enter a statement that is terminated by a semicolon, **ans** is deleted.

4.3 Operators, assignment, precendence

A number of operators is available for commonly used functions. Unary prefix operators:

symbol	function
$\sim$	reverse
$-$	negate
!	inverse

The unary operators have the highest precedence, so they are executed before any other operations. Because each of these operators will cancel itself, GAViewer will determine if it is necessary to execute it. So if you can write **minus minus not not tilde tilde x** not a single operator function will be to **x** because all operators cancel each other.

The following binary operators are available: (in order of precedence):

symbol	function	precedence level
$\wedge$	outer product	9
$\vert$	join	8
&	meet	8
.	inner product	7
'space'	geometric product	6
$*$	geometric product	6
/	inverse geometric product	6
$+$	addition	5
$-$	subtraction	5
$<$	less	4
$>$	greater	4
$<=$	less or equal	4
$>=$	greater or equal	4
$==$	equal	3
$!=$	not equal	3
&&	boolean and	2
$\vert\vert$	boolean or	1
$=$	assignment	0

All operators are left associative, except assignment which is right associative (but it isn't really an operator...) The geometric product can be written as either a space or a $*$, so the following two lines are equivalent:

```
>> x = a b;
>> x = a * b;
```

All these operators are internally translated to function calls by GAViewer (sections 4.9.1 and 4.9.3). The . operator is translated to **hip** (Hestenes inner product) by default, but it can be set by the **inner_product()** function. The default is **hip**, but **mhip**, **rcont** and **lcont** are also possibilities [1]. This example shows the effect of changing the inner product from Hestenes inner product to left contraction:

```
>> e1 ^ e2 . e1   // here the default Hestenes inner product
ans = -1.00*e2
>> inner_product(lcont)
>> e1 ^ e2 . e1   // now the left contraction is used
ans = 0
>>
```

4.4 Variables, types and casting

Variables like **x**, **a** and **b** in the example above always have a type. This type can be **e3ga** (Euclidean), **p3ga** (projective, homogeneous) or **c3ga** (conformal). Variables usually 'inherit' their type from the variables used to compute them. So if in th example above **a** and **b** are both of type **p3ga**, then the result **x** will also be of type **p3ga**.

The type of a variable determines how a variable gets interpreted and drawn. GAViewer can analyze blades and versors from the three models, and draws them as such. Multivectors that can not be analyze are cam not be drawn.

The type of a variable can also be explicit set or *cast*:

```
>> a = (c3ga)e3
a = e3
>> b = (e3ga)ni
b = 0
```

As you can see in the example **b = (c3ga)ni**, no interpretation is done during casting. Since there is no **ni** basis vector in the **e3ga** model, it is simply discarded. As an other example, a 'free vector' in the conformal model will not turn into a regular vector in the Euclidean model by simply casting it.

If a variable name does not exist yet, it is assumed to be 0:

```
>> a = longVariableNameThatDoesNotExistYet
a = 0
```

This can be quite confusing if you mistype the name of a function:

```
>> a = duel(x)
a = 0
```

Here, a typo was made: **duel** instead of **dual**. GAViewer will assume **duel** is a variable instead of the function **dual**. It computes the geometric product of **duel** (which is assumed to be **0**) and **(x)**, so the result will be **0** as well.

[1] Actually you can use any 2-argument function

If some identifier, like **alpha** is currently a function, you can force it to become a variable by using the **variable** statement. The other way around can be done by declaring the function again. Consider:

```
>> a = alpha(e1, 0.5)
a = 1.00*e1
>> variable alpha; // declare alpha -> variable
>> alpha = 1
alpha = 1.00
>>
>> function alpha(x, y); // declare alpha -> function
>> alpha = 1  // this is no longer allowed
line 1:7: expecting (, found '= '
ans = 1.00
>> a = alpha(e1, 0.5)
a = 1.00*e1
```

4.5 Builtin Constants

The following builtin constants are available:

- All scalar numbers are constants. Scalar numbers can have the following forms: **1**, **1.2**, **1.2e3**, **1.2e-3** They are of type **e3ga**.
- **e1**, **e2**, **e3**: These are the three Euclidean basis vectors, type **e3ga**.
- **e0**: the origin in the projective model, type **p3ga**.
- **ni** and **no**: conformal infinity and origin, type **c3ga**.
- **einf**: synonym for conformal infinity, type **c3ga**.
- **pi**: 3.14159265358979323846264338327950, type **e3ga**.
- **e_**: 2.71828182845904523536028747135270, type **e3ga** [2].

By default, constants always have the type of the smallest model that contains them. So scalars and the Euclidean basis vectors are all of type **e3ga** by default. But this behaviour can be changed by calling the **default_model()** function. For example:

```
>> default_model(c3ga);
```

After this call to **default_model()**, all constants (except **e0**) will be of type **c3ga**. Valid arguments to **default_model()** are **e3ga**, **p3ga**, **c3ga**, **c5ga**, **i2ga**.

If you want to go back to the normal behaviour, just call **default_model()** without any arguments:

```
>> default_model();
```

Why does it matter what the default type/model of constants is? It makes a difference for certain functions, such as dualization with respect to the whole space. It also makes a difference in interpretation. For example, **e3** is drawn as a vector in the **e3ga** model, but drawn as a (dual) plane in the **c3ga** model.

[2]'**e_**' is called '**e_**' and not **e** because otherwise you could not use **e** as a variable name, which is quite common.

4.5.1 Renaming builtin constants

You can rename the builtin constants using the function **rename_builtin_const()**. This also effects the textual output of multivector coordinates, as shown in this small example, which renames the conformal origin (**no** by default) and infinity (**ni** by default) to **e0** and **einf**, respectively.

```
>> rename_builtin_const(e0, e4); // proj. origin e0 -> e4
>> rename_builtin_const(no, e0); // conf. origin no -> e0
>> rename_builtin_const(ni, einf); // conf. infinity ni -> einf
>> a = e0 ^ einf
a = 1.00*e0^einf
```

You have to be careful to avoid nameclashes. In the example, the projective origin **e0** first had to be renamed to **e4**, so that **no** could be renamed to **e0**.

4.6 Adding your own constants

You can add your own *user constants* using the **add_const()** and remove them using **remove_const()**. This is useful, because ordinary variables are removed when you do a **clf()**, constants are not. Also, user constants are subject to the casting/default_model rules described above. Here is an example of using **add_const()** and **remove_const()**:

```
>> add_const(I3 = e1 ^ e2 ^ e3);
>> I3
ans = 1.00*e1^e2^e3
>> remove_const(I3);
```

4.7 'Arrays'

Array indices can be used to generate new variable identifiers. GAViewer does not have any real support for arrays. For example, you can not pass arrays to functions or return them from functions. The C-like syntax for accessing an array element in GAViewer is **A[idx1][idx2] ... [idxN]**.

4.8 Calling functions

A function is called as follows:

```
>> returnValue = func_name(arg1_name, ... , argn_name);
```

For example, if you want to project blade 'a' onto blade 'b' and store the result in 'x', you can call **project** like this:

```
>> x = project(a, b);
```

GAViewer will always search for the *best matching* function to do the job. *Best matching* means:

- The function must have the right name and right number arguments.

- Preferably, all arguments have the right type (**e3ga**, **p3ga**, **c3ga**). This would be a perfect match.
- If no perfect match can be found, the next best match is searched: all functions with the right name and right number arguments are collected. The best matching function is the one for which the 'coercing distance' is smallest. This distance is defined as follows: coercing to a higher dimensional model is preferred over coercing to a lower dimensional model, since no information is lost.

4.9 Built-in functions

A description of these functions is also accessible on the GAViewer console through

```
>> help();
```

and

```
>> help(topic);
```

4.9.1 Products

While many products are accessible through operators, they can also be explicitly called through the following functions:

function(arguments)	return value
gp(a, b)	geometric product of **a** and **b**
igp(a, b)	inverse geometric product of **a** and **b**
op(a, b)	outer product of **a** and **b** (**a** ^ **b**)
hip(a, b)	Hestenes inner product of **a** and **b**
mhip(a, b)	modified Hestenes inner product of **a** and **b**
lcont(a, b)	left contraction of **a** and **b**
rcont(a, b)	right contraction of **a** and **b**
scp(a, b)	scalar product of **a** and **b**
meet(a, b)	meet of **a** and **b**
join(a, b)	join of **a** and **b**

Since version 0.41, two products are available in a Euclidean Metric flavour also, which can be useful for low level work (for example, the meet and join use these products internally):

function(arguments)	return value
gpem(a, b)	geometric product (Euclidean Metric) of **a** and **b**
lcem(a, b)	left contraction (Euclidean Metric) of **a** and **b**

4.9.2 Basic GA functions

The following table list some basic GA functions:

function(arguments)	return value
add(a, b)	the sum of **a** and **b**
sub(a, b)	the sum of **a** and **-b**
scalar(a)	the scalar part of **a**
dual(a)	the dual of **a** with respect to the full space
dual(a, b)	the dual of **a** with respect to **b**
reverse(a)	the reverse of **a**
clifford_conjugate(a)	the clifford conjugate of **a**
grade_involution(a)	the grade involution of **a**
inverse(a)	the (versor) inverse of **a**
general_inverse(a)	the inverse of **a** even if
	a is not a versor (returns 0 if inverse does not exist)
negate(a)	returns the negation of **a**

To extract a grade part, or to determine the grade of a blade:

- **grade(a, b)**: returns the grade **b** part of **a**, e.g., **grade(a, 2)**.
- **grade(a)**: if **a** is a blade, returns the grade of **arg1**, returns **-1** otherwise.

To determine the parity of a versor :

- **versor_parity(a)**: if **a** is an even versor returns: 0, odd versor: 1; not a versor : -1.

To compute the norm of a multivector, orto normalize a multivector, you can use of the following functions:

function(arguments)	return value
norm_2(a)	the sum of the square of all coordinates of **a**
norm_r(a)	the grade 1 part of **a**$\tilde{a}$
norm(a)	the square root **abs(norm_r(a))**, multiplied by
	the sign of **norm_r(a)**
normalize(a)	**a / abs(norm_r(a))**

To wrap it up, there are a few handy functions for doing versor products, projection, rejection and factorization of a blade:

function(arguments)	return value
versor_product(a, b)	returns **a b inverse(a)**
vp(a, b)	synonym of **versor_product(a, b)**
inverse_versor_product(a, b)	returns **inverse(a) b a**
ivp(a, b)	synonym of **inverse_versor_product(a, b)**
project(a, b)	returns the projection of **a** onto **b**
reject(a, b)	returns the rejection of **a** from **b**
factor(a, b)	returns factor '**b**' of blade **a**
	(**b** must be an integer in range [1 **gradea**])

4.9.3 Boolean

A number of functions for doing boolean arithmetic are available. 0.0 is 'false'. Any value that is not 'false' is considered to be 'true'. The function **scalar()**

returns the grade 0 part of a multivector.

function(arguments)	return value
equal(a, b)	true if (**a** - **b**) equals 0
ne(a, b)	true if (**a** - **b**) does not equal 0
less(a, b)	true if **scalar(a)** $<$ **scalar(b)**
greater(a, b)	true if **scalar(a)** $>$ **scalar(b)**
le(a, b)	(less or equal) true if **scalar(a)** $\leq$ **scalar(b)**
ge(a, b)	(greater or equal) true if **scalar(a)** $\geq$ **scalar(b)**
and(a, b)	true if **a** is true and **b** is true
or(a, b)	true if **a** is true or **b** is true
not(a)	true if **a** is false

Bitwise boolean arithmetic can be done with the functions in the following table. Arguments are converted to 32 bit unsigned integers before performing the bitwise operation.

function(arguments)	return value
bit_not(a)	returns the bitwise *not* of **scalar(a)**
bit_and(a, b)	returns the bitwise *and* of **scalar(a)** and **scalar(b)**
bit_or(a, b)	returns the bitwise *or* of **scalar(a)** and **scalar(b)**
bit_xor(a, b)	returns the bitwise *xor* of **scalar(a)** and **scalar(b)**
bit_shift(a, b)	returns bitwise *shift left* of **scalar(a)** by **scalar(b)**

The second argument of **bit_shift()** can be negative for right shift. The only reason to include these bitwise boolean functions was to allow user to write there own **autocolor()** 4.13 function, where some tests on bitfields need to be done. The bitwise boolean functions are not of much use for geometric algebra.

4.9.4 Drawing

Various drawing properties of multivectors can be set with the functions described below. The functions are always used as in the example:

```
>> a = cyan(e1)
a = 1.00*e1
>>
```

This draws the vector **a** in cyan.

Compare this to the following example, which will not work as expected:

```
>> green(a)
ans = 1.00*e1
```

One might expect **green(a)** to turn the multivector **a** green. Bit what actually happens is that the value of 'a' gets assigned to **ans**, and then **ans** gets drawn. Since **ans** is equal to **a**, it may or may not be drawn on top of **a**. On the next statement you enter, the value of **ans** will be overwritten or removed, and you'll see that the actual color of **a** has not changed.

So the functions that modify drawing properties only set certain flags and values on the intermediate variables and have no effect unless such intermediate variables are assigned to something.

Drawing functions can be nested like this:

```
>> a = cyan(stipple(e1))
a = 1.00*e1
>>
```

This draws a stippled vector **a** in cyan.

Color and alpha

The color and opacity (often called 'alpha' in computer graphics) of variables can be set using these functions:

function(arguments)	effect:
red(a)	**a** turns red
green(a)	**a** turns green
blue(a)	**a** turns blue
white(a)	**a** turns white
magenta(a)	**a** turns magenta
yellow(a)	**a** turns yellow
cyan(a)	**a** turns cyan
black(a)	**a** turns black
grey(a)	**a** turns grey
gray(a)	**a** turns gray
color(a, R, G, B)	the color of **a** becomes the RGB value [**R**, **G**, **B**]
color(a, R, G, B, A)	the color / opacity of **a** becomes the RGBA value [**R**, **G**, **B**, **A**]
alpha(a, value)	the alpha (opacity) of **a** becomes **alpha**

Red, green, blue and alpha values should be in the range [0.0 1.0]. A **alpha** of 0.0 is entirely transparent (the object will be invisible), while a value of 1.0 will be entirely opaque. Values outisde the [0.0 1.0] range will be clamped by OpenGL.

Multivectors can be drawn stippled, wireframed, with or without weight or orientation, and some multivector interpretations allow for an outline to be drawn. All of this can be set with these functions:

function(arguments)	effect:
stipple(a)	draws **a** stippled
no_stipple(a)	draws **a** not stippled
wireframe(a)	draws **a** in wireframe
no_wireframe(a)	draws **a** without wireframe
outline(a)	outlines **a**
no_outline(a)	does not outline **a**
weight(a)	draws the weight of **a**
no_weight(a)	does not draw the weight of **a**
ori(a)	draws the orientation of **a**
no_ori(a)	does not draw the orientation of **a**

To force hiding / showing a multivector, use the **show()** and **hide()** functions, but rememeber to assign!

```
>> a = e1       // Draws 'a'
a = 1.00*e1
>> a = hide(a), // Hides 'a', despite the comma.
a = 1.00*e1
>> a = show(a); // Shows 'a', despite the semicolon.
a = 1.00*e1
```

```
>> hide(a) // Does not hide 'a'.
           // Instead, assigns the value of 'a' to 'ans',
           // and hides 'ans'.
```

Some multivector interpretations can be drawn in multiple ways. For instance, if you type:

```
>> line = ori(ni ^ no ^ e1)
```

you'll see a popup menu labeled 'draw method' in the controls on the right hand side of tha GAViewer UI. Use it to select various ways to draw the orientation of the line. From the console, you can also set the draw method, using the **dm** functions:

```
>> line = dm2(ori(ni ^ no ^ e1))
```

The index ('2' in the example) can range from '1' to '7'. If it is out of range for the specific multivector interpretation, the default is used.

You can retrieve the drawing properties of variables using the following functions:

function(arguments)	return value:
get_color(a)	a vector with rgb color of **a**
get_alpha(a)	a scalar with alpha of **a**
get_stipple(a)	a boolean scalar with flag stipple of **a**
get_wireframe(a)	a boolean scalar with flag wireframe of **a**
get_outline(a)	a boolean scalar with flag outline of **a**
get_weight(a)	a boolean scalar with flag weight of **a**
get_ori(a)	a boolean scalar with flag ori of **a**
get_hide(a)	a boolean scalar with flag hide of **a**

A label can be drawn at the 'position' of an object using the **label()** function. Not every object has a positional aspect to it, in which case the label will be drawn in the origin. The first argument to **label()** is the variable that you want to label. The optional second argument is that text of the label (by default, the name of the variable is used as the label text). Some examples:

```
>> a = e1
a = 1.00*e1
>> label(a);
>> label(b = 2 a) // this is short for b = 2a, label(b);
b = 2.00*e1
>> label(c = e2, "this is c")
c = 1.00*e2
```

The following two functions don't really affect how a variable is drawn, but more how a variable is interpreted.

- **versor(a)**: forces a versor interpretation of **a**.
- **blade(a)**: forces blade interpretation of **a**.

versor() can be useful when a versor coincidentally becomes single-grade. Consider:

```
>> a = versor(e1 ^ e2)
a = 1.00*e1^e2
```

This draws a rotor, whereas simply

```
>> a = e1 ^ e2
a = 1.00*e1^e2
```

draws a bivector.

blade() is useful when floating point noise on some grade parts becomes so large that GAViewer mistakes a blade for a versor. Suppose you have a bivector **a = e1** $\wedge$ **e2** but due to some manipulations, floating point nouse causes the scalar part to be **0.01** instead of **0**:

```
>> a = e1 ^ e2 + 0.01
a = 0.01 + 1.00*e1^e2
```

a gets interpreted and drawn as a rotor in this case. Now, to force **a** to be interpreted as a blade, use:

```
>> a = blade(e1 ^ e2 + 0.01)
a = 0.01 + 1.00*e1^e2
```

4.9.5 Controls

Scalar controls can be created by using the functions described in the list below. To get an idea of what scalar controls are useful for, enter the following code on the console:

```
>> ctrl_range(a = 2.0, 0.5, 10.0);
>> dynamic{v = a e1,}
v = 2.00*e1
```

Now move the slider that appeared in the lower right window (section 4.10 for a discussion of **dynanic**).

- **ctrl_bool(name = value)**: creates a boolean control with name **name**, set to **value**.
- **ctrl_range(name = value, min_value, max_value)**: creates a slider control with name **name**, set to **value**, limited to **min_value** and **max_value**.
- **ctrl_range(name = value, min_value, max_value, step)**: creates a slider control with name **name**, set to **value**, limited to **min_value** and **max_value** values, that can only be changed **step** at a time.
- **ctrl_select(name = value, option1 = value1, ..., optionN = valueN)**: creates a selection menu with name **name**, set to **value**. A maximum op 7 options can be specified, **value** must be one of the options
- **ctrl_remove(name)**: removes any control with name **name**

All these functions also have a variant where you can specify a callback batch function to be called when the user changes the widget. These functions have names ending in **_with_callback**. An example:

```
batch myCallback() {
    switch(choice) {
    case 1:
        x = c3ga_point(e1),
        break;
    case 2:
        x = c3ga_point(e2),
        break;
    }
}

ctrl_select_with_callback(choice=1, choice1 = 1, choice2 = 2, "myCallback");
```

4.9.6 Goniometric functions, sqrt, abs, log, exp, pow.

All values for goniometric functions in radians. For all functions except **exp** and **pow**, only the scalar part of the argument is used.

function(arguments)	return value:
tan(a)	obvious
sin(a)	obvious
cos(a)	obvious
atan(a)	obvious
asin(a)	obvious
acos(a)	obvious
atan2(a)	obvious
sinh(a)	hyperbolic sine of **a**
cosh(a)	hyperbolic cosine of **a**
sinc(a)	$sin(\mathbf{a})/\mathbf{a}$
sqrt(a)	square root of **a**
abs(a)	absolute value of **a**
log(a)	natural logarithm of **scalar(a)**
exp(a)	exponentiation of **a**,
	inaccurate and slow series expansion
pow(a, b)	**a** multiplied **b** times with itself
	(**b** must be integer ≥ 0)
scalar_pow(a, b)	**a** raised to the power of **b**

4.9.7 Projective model functions

Two functions are available to construct point in the **p3ga** model.

- **p3ga_point(c1, c2, c3)**: returns the **p3ga** point constructed from the **e3ga** vector [**c1 e1 + c2 e2 + c3 e3**].
- **p3ga_point(e3ga a)**: returns the projective point constructed from the **e3ga** vector **a**.

4.9.8 Conformal model functions

The following functions are of use in the **c3ga** model:

- **c3ga_point(c1, c2, c3)**: returns the conformal point constructed from the **e3ga** vector [**c1 e1 + c2 e2 + c3 e3**].
- **c3ga_point(e3ga a)**: returns the conformal point constructed from the **e3ga** vector **a**.
- **c3ga_point(p3ga b)**: returns the conformal point constructed from the **p3ga** point **a**.
- **translation_versor(a)**: returns a conformal versor that translates over the **e3ga** vector **a**.
- **tv(a)**: synonym of **translation_versor(a)**.
- **translation_versor(c1, c2, c3)**: returns a conformal versor that translates over vector [**c1 e1 + c2 e2 + c3 e3**]
- **tv(c1, c2, c3)**: synonym of **translation_versor(c1, c2, c3)**.

4.9.9 System functions

A collection of functions that give access some low level aspects of the viewer.

- **assign(a, b)**: assigns the value of **b** to **a**, returns **b**. This is what the **=** operator evaluates to.
- **cprint(”a string”)**: prints **”a string”** to the console.
- **print(a)**: prints coordinates of **a** to the console, returns 0.
- **print(a, prec)**: prints coordinates of **a** to the console with precision **prec** (e.g., **prec** = ”e”), returns 0.
- **cmd(”a command”)** executes **”a command”** as if it had been read from a .geo file.
- **prompt()** sets the default console prompt.
- **prompt(”prompt text”)** turns the console prompt into **”prompt text”**.
- **select(a)** selects **a** as the current variable / object, as if it had been selected using **ctrl-left-click**.
- **remove(a)** removes the variable / object **a**, as if the **remove this object** had been clicked.
- **clc()** clears the console and removes all control variables.
- **clf()** removes all variables / objects.
- **reset()** resets the entire viewer (console, dynamics, variables, user constants, etc).

4.9.10 Networking

Since version 0.4, other programs can communicate with GAViewer over a TCP connection. To enable this, use the command:

```
>> add_net_port(6860);
Server socket setup correctly at port 6860
```

After this command, GAViewer will listen on TCP port 6860 to clients trying to connect.

You can try the connection out by using telnet. In a Unix terminal, or on a Windows command prompt use something like

```
telnet localhost 6860
```

to connect. GAViewer will immediately send the values of all known variables:

```
"autocolor" = 1.000000000000000e+000;$"camori" = 1.000000000000000e+000;$"cam
" = 1.100000000000000e+001*e3;$
```

In the telnet application, you can type commands that you would otherwise type on the GAViewer console, but they must be terminated with a dollar sign. For example:

```
a = e1 + e2,$
```

GAViewer will reply:

```
"a" = 1.000000000000000e+000*e1 + 1.000000000000000e+000*e2,$
```

because the value of variable **a** changed. Every time a variable changes value, GAViewer will send such a message to all connected clients.

Here are the commands related to networking

- **add_net_port(port)**: start listening on TCP port.
- **remove_net_port(port)**: stop listening on TCP port; disconnect all current clients.
- **net_status()**: prints out a summary of the network status.
- **net_close()**: shuts down all network connections and ports.

You can start up GAViewer as follows:

```
gaviewer -net
```

This enables TCP port 6860 immediately. Optionally you can specify the port, for example

```
gaviewer -net 5000
```

This enables TCP port 5000 immediately.

Networking is disabled by default, because anyone on the internet can connect to your GAViewer this way (unless you block this using a firewall) and I cannot guarantee that GAViewer cannot be exploited (through buffer overflows, for example) to give someone access to your computer. However, it is unlikely that someone will go through the effort of finding such an exploit as long as GAViewer is a niche application.

4.10 Dynamic statements

Dynamic statements are created using the **dynamic** language construct as shown in the following example:

```
>> a = e1,
a = 1.00*e1
>> dynamic{b = a ^ e2,}
b = 1.00*e1^e2
```

Dynamic statements *depend* on the variables used to evaluate them. In the example, the statement **b = a** $\wedge$ **e2,** depends on the variable **a**, and **b**. Every time one of these variables changes, the dynamic statement is re-evaluated. Because constants like **e2** never change, they don't effect when a dynamic statement is re-evaluated.

If, after entering the example above on the console, you would alter the value of **a**, e.g.:

```
>> a = e1,
a = 2.00*e1
```

you would see that **b** gets updated automatically. You can also **ctrl-right-drag a** and **b** will change as well.

If you would set the value of **b** by hand, e.g.:

```
>> b = 0,
b = 0
```

you will notice that **b** does not change. GAViewer detects that **b** has been altered and reevaluates all dynamic statement involving **b**.

You can view, edit and remove active dynamic statements by selecting the menu item **Dynamic**$\rightarrow$**View dynamic statements**.

If you enter multiple dynamic statements that assign values to the same variables, things can get tricky and confusing. Consider:

```
>> a = e1,
a = 1.00 * e1
>> dynamic{b = a ^ e2,}
b = 1.00*e1^e2
>> dynamic{b = c ^ e2,}
b = 0
```

Directly after the second dynamic statement has been entered, **b** will be set to **0**. But immediately after that, GAViewer will notice that the first dynamic statement (**b = a** $\wedge$ **e2,**) should be reevaluated, because the value of **b** has been altered. Thus GAViewer will execute **b = a** $\wedge$ **e2,**. If no infinite loop protection was in place, GAViewer would now re-evaluate the second dynamic statement again, followed by the first, etc.

However, GAViewer protects against infinite loops in dynamic statement evaluation. A dynamic statement will never get reevaluated twice due to some kind of loop-dependency in the dynamic statements. In general it is better to avoid such loops because they can be quite confusing.

Some short remarks:

- **dynamic** can only be called in the global scope, not inside functions or batches called by functions.
- Setting the inner product using **inner_product()** (section 4.3) will not cause dynamic statements using the **.** operator to be re-evaluated.
- The function **cld()** removes all dynamic statements
- Terminating statements with colon or semincolon only effects the visibility of variables on the first time the dynamic statement is evaluated. **hide()** and **show()** *do* affect visibility on every re-evaluations.

4.10.1 Named Dynamic Statements

The problem with dynamics is that you have little control over them once they have been created. You can remove them all using **cld()**, you can edit them in **Dynamic→View dynamic statements**, but that's it. *Named dynamics* offer more flexibility. By adding a name tag to every dynamic, you can overwrite it later.

An example of the syntax for creating and overwriting a named dynamic is:

```
>> a = e1,
a = 1.00 * e1
>> dynamic{my_dynamic: b = a ^ e2,}
b = 1.00*e1^e2
>> dynamic{my_dynamic: b = a ^ e3,}
b = 1.00*e1^e3
```

4.10.2 Animations

One way to achieve animation in GAViewer is by writing dynamic statements that depend on the **atime** variable.

For example, enter the following dynamic statement on the console:

```
>> dynamic{print(atime);}
atime = 0
```

Now select **Dynamic→Start/Resume Animation**:

```
>> atime = 0
>> atime = 0.01
>> atime = 0.07
>> atime = 0.13
>> atime = 0.19
```

As you can see, **atime** will be set to the time elapsed since the 'animation' was started. **atime** will be set at most 30 times per second, but it may be slower, depending on the time required to redraw the view window and to re-evaluate dynamics.

To stop the animation, select **Dynamic→Stop Animation**. To pause, select **Dynamic→Stop Animation**.

By writing more complicated dynamic statements involving **atime**, more interesting animations can be produced.

You can also start and stop animations from the console.

- **start_animation()** starts the animation of dynamics depending on **atime**. It is guaranteed that when an animation starts, **atime** will be **0**. You can check **atime == 0** and do some kind of initialization if you need to.
- **stop_animation()**: stops animation.
- **pause_animation()**: pauses animation.
- **resume_animation()**: resumes animation (synonym of **start_animation()**).

For example:

```
>> cld(); // remove all current dynamics
>> dynamic{a = sin(atime) e1, if (atime > 10) stop_animation();}
a = -0.86*e1
>> start_animation();
```

This animation will run for about 10 seconds because it stops itself when **atime** is larger than 10.

4.11 Control constructs

Several control constucts are available in the language. They are all modelled after the C programming language.

4.11.1 if else

An **if else** looks like this

```
if (condition) statement
[else statement]
```

As indicated by the square brackets, the else part is optional.

For example:

```
>> if (a == 1) {b = 1;} else {b = 2;}
```

For simple **if else** statements you can leave out the curly brackets. So the example from above can be rewritten without change in semantics to the following:

```
>> if (a == 1) b = 1; else b = 2;
```

4.11.2 for

A **for** statement allows one to write a loop. It looks like:

```
for ([init] ; [cond] ; [update]) statement
```

It works as follows: to begin with, the **init** statement is executed. Then before every execution of the **statement**, it is checked whether **cond** is **true**. If so, **statement** is executed, otherwise the loop terminates. After the **statement** has been executed, **update** is executed.

The square brackets around **init**, **cond** and **update** indicate that they are all optional. If **cond** is not specified, it is assumed to be **true**.

An example of a **for** loop is:

```
>> for (i = 0; i < 5; i = i + 1) {print(i);}
i = 0
i = 1.00
i = 2.00
i = 3.00
i = 4.00
```

A **for** loop can be terminated prematurely by issuing a **break** statement. A **for** loop can be forced to terminate the execution of the current loop body by using a **continue** statement.

The following **for** loop would loop forever, if it wasn't for the **break** statement. The **print()** of **i** for **i == 1** is skipped, as you can see in the output.

```
>> for (;;) {
    if (i == 1) continue;
    print(i);
    if (i == 3) break;
}
i = 0
i = 2.00
i = 3.00
```

4.11.3 while

The **while** language construct also allows you to perform loops, but in a slightly different format. A **while** loop looks like

```
while (cond) statement
```

It loops as long as **cond** is true, executing the **statement** on every loop.

An example of a **while** loop is:

```
>> while (i < 5) {print(i); i = i + 1}
```

It is very easy to write infinite **for** or **while** loops, for example, by forgetting to increment **i**. In such a case, GAViewer will hang forever. Try for example:

```
>> while (1) {}
```

As with the **for** loop, **continue** and **break** can be used to control the loop.

4.11.4 switch

A **switch** statement allows you to compare the value of an expression against the value of a bunch of other expressions, and to undertake some action depending on the outcome of that. It looks like this:

```
switch(expression) {
case expr1:
    [statements]
case exprn:
    [statements]
default:
    [statements]
}
```

All the statements are optional. They may include **break** statements to leave the **switch**.

A more or less typical **switch** would look like:

```
switch(i) {
case 1:
    cprint("i is 1");
    break;
case 2:
case 3:
    cprint("i is 2 or i is 3");
    break;
case j + 2:
    cprint("i is (j + 2)");
    break;
default:
    cprint("the default action was called");
}
```

Note that, as opposed to C, arbitrary expressions (such as **j + 2**) are allowed for **case**s.

4.12 Writing functions and batches

You can write your own functions and *batches* and load them into GAViewer. A batch is a function, except it executes in the same scope as the caller. In what follows we will refer to both functions and batches as just 'functions', and explain the special stuff for batches later.

You can store functions in *.g* files, or type them at the console. For example, a small function that returns the largest of two values can be entered directly on the console:

```
>> function max(a, b) {if (norm(a) > norm(b)) {return a;}
       else {return b;}};
Added function max( a,  b)
```

You can then use **max()** like this:

```
>> max(1, 3)
ans = 3.00
```

A function definition looks like this:

```
function func_name([e3ga|p3ga|c3ga] arg1_name, ...
                   [e3ga|p3ga|c3ga] argn_name) {
    // function statements
}
```

The first word of a function definition is always **function** or **batch**. This is followed by the name of the function (**func_name** in the example above. Between a round open and a round close parentheses, the arguments to the function are specified. A function can have 0 to 8 arguments. The value 8 is hard coded.

More than 8 arguments currently crashes the GAViewer. An arbitrary number of arguments should be allowed for in the future.

The type specifier (**e3ga**, **p3ga**, **c3ga**) is optional. If no type is specified, actual arguments always match perfectly. A return type specifier can not be given. A multivector is always returned, without any restrictions on the model/type.

Inside a function, you can type arbitrary statements. A **return expr** statement cause the function to terminate and return the value of **expr**.

It is also allowed to define new functions inside functions. For example:

```
function f(a) {
    function neg(b) {
        return -b;
    }
    return 2 * neg(a);
}
```

You can declare the existence of a certain function without defining it like this:

```
function f(a);
```

So a function declaration is like a function definition, except you leave out the function body and replace it with a semicolon.

All variables inside a function are local by default. This means that you can not see variables from the global scope, nor can you set variables in the global scope. The **::** operator can be used reach variables in the global scope from inside a function:

```
function setGlobalVar(a) {
    ::globalVar = a;
}
```

4.12.1 Batches

Batches are functions that execute in the same scope as the caller. This means that you'll have to be really careful when calling batches and make sure that variable names used in the batch are not used for some other purpose in the calling scope. For instance, if formal argument names of a batch are already present in the scope of the caller, they will be overwritten.

Batches are most useful for writing interactive demos and tutorials. Often you want to execute a bunch of statements that are too tedious for the user of your tutorial to type by hand. Then you can collect these statements into a batch, store them in a .g file and have the user load that. There is a special **suspend** statement to allow for interaction inside batches. Consider:

```
batch demo1() {
a = show(e1);
cprint("'a' is now equal to e1.");
cprint("Drag the left mouse button to rotate your view.");
cprint("Type arbitrary commands on the console.");
cprint("Press enter to continue.");
// set a special prompt
prompt("demo1 suspended... >> ");

// suspend the current batch
suspend;

// set the prompt back to normal
prompt();

cprint("Welcome back to demo1");
a = show(e2);
cprint("'a' is now equal to e2");
cprint("This is the end of demo1");
}
```

Using **suspend** is not allowed outside the global scope.

4.13 Autocolor.

You may have noticed that not all variables are drawn in the same color. Grade 1 variables are red, grade 2 variables are blue, etc. This color is set by a built-in function called **autocolorfunc()**.

Every time variable in the global scope that is assigned a value, it is send is through **autocolorfunc()** to give it a distinctive look before. The default built-in **autocolorfunc()** changes the color of the variable depending on it grade (but only if the color has not been set explicitly by one of the drawings function (section 4.9.4)). It also turns on stippling for imaginary conformal model objects (such as the circles of intersection of two spheres that do not intersect).

You can turn the auto-color feature of by setting the global variable **autocolor** to **false**. If **autocolor** is **false**, all variables get the same default foreground color.

4.13.1 Writing your own autocolor() function.

You can also write your own **autocolorfunc()** if you are not satified with the default. Writing your own **autocolorfunc()** is pretty low level, so be prepared. Check out the *autocolor.g* that is included in the GAViewer documentation package to see what the default **autocolorfunc()** looks like. Also remember that the default built-in **autocolorfunc()** executes faster than ordinary *.g* files because it was written in C++ instead and compiled to machine code, while *.g* files are interpreted.

Two important things you'll want to know is:

- How do I get information about how an object is interpreted? You can find out by calling **get_interpretation(a)**. This returns a bitfield that contains information about the interpretation of **a**. You can use the bitwise boolean functions to extract information from the bitfield (see *autocolor.g* for an example).

- How do I know whether some drawing property of a variable was already set explicitly by the user. Of course, you don't want to override what the user has explicitly set. So call **get_draw_flags(a)**. This will return a bitfield that contains information about which drawing properties were set by the user.

Chapter 5

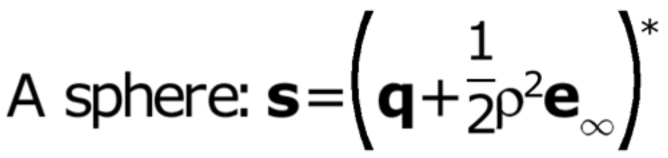
Typesetting labels.

GAViewer features a simple typesetting language which resembles Latex. It was implemented by the author as an introduction to parsing and interpreting languages, and of course because of its functionality. Here is a list of some of the features the the typesetting system provides:

- text and equation modes,
- sub- and superscripts,
- four scalable fonts (regular, italic, bold, greek),
- some special 'GA'-symbols,
- tabulars,
- left, right, center, and justifyable alignment,
- hats,
- scalable parenthesis,
- (square) roots,
- fractions,
- custom commands (macros), and
- (custom) colors.

The implementation of the typesetting system was kept simple, which makes it unsuitable for typesetting large amounts of text. The input is parsed, checked for syntax errors, and converted to a parse tree. After the parse pass have been completed, the parse tree is interpreted an turned into drawing commands. Then the parse tree can be released, and the drawing commands sent to OpenGL (or some other graphics API). Because parsing, interpreting and drawing do not occur in one pass, large amounts of memory will be required in large amounts of text are supplied as input. As a side note, lex and yacc are used for parsing the input.

The font for the typesetting is a 1024×1024 pixel 'GL_ALPHA' texture. It is included directly in the source code, in *fontdata.cpp*. The font is antialiased



A sphere: $\mathbf{s}=\left(\mathbf{q}+\frac{1}{2}\rho^2\mathbf{e}_{\infty}\right)^{*}$

Figure 5.1: An example of the use of text and equation modes.

and contains all ASCII symbols of four font: regular, italic, bold, greek. It also contains several special GA-only symbols.

Figure 5.1 shows an example of a complex typeset label. This figure is representative for the complexity the typesetting system was designed. The system can handle multiple lines and such, but don't use it to typeset entire pages like you can do with Latex.

5.1 txt and eqn modes.

There are two parse modes with subtle differences. **txt** mode, the default, is used to typeset ordinary text, like this sentence. **eqn** mode is intended for typesetting equations. Whitespaces, numbers and operators are treated differently in both modes, and some commands, such as **sqrt** and **frac** are only valid in **eqn** mode. Here is an example of switching between modes:

```
\txt{This is ordinary sentence}
```

```
\eqn{a = b + c - d}
```

The former line switches to text mode. This causes all text between the curly parentheses to be parsed as ordinary text. The latter line uses equation mode. The following, more complex line combines the uses of **txt** and **eqn** modes:

```
\txt{A sphere: \eqn{\bold{s} = \par{()}
{\bold{q} + \frac{1}{2}\rho^{2}\bold{e}_{\infty}}^{*}}
```

As you can see, the **eqn** part is inside embedded in the **txt** part. Because the default parsing mode is **txt**, the following is equivalent to the previous line (unless you have changed the default parsing mode):

```
\txt{A sphere: \eqn{\bold{s} = \par{()}
{\bold{q} + \frac{1}{2}\greek{r}^{2}\bold{e}_{\infty}}^{*}}
```

Both these sample lines result in the output shown figure 5.1.

5.1.1 txt mode details:

In **txt** mode, multiple instances of spaces, newline and tabs are interpreted as a single instance of whitespace. Whitespace at the end of a line is ignored and deleted. To force the insertion whitespace, use the **ws** command (section 5.4).

Sequences of letters, numbers, underscores, plus and minus signs are interpreted as words. Other ASCII characters are interpreted as single-character

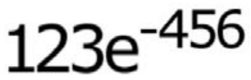


Figure 5.2: An example of a number in **eqn** mode.

words. A backslash followed by letters is interpreted as a command. A backslash followed by numbers is interpreted as a custom command argument (section 5.13). Two backslashes next to each other is short for the **newline** command. A backslash followed by any other character is interpreted as an escape sequence for that character. To produce an actual backslash in the output, use the **backslash** command.

The **frac**, **sqrt**, **har** and **par** commands are not valid in **txt** mode.

5.1.2 eqn mode details:

In **eqn** mode, all whitespace is ignored (unless it was forced using the **ws** command). Characters and symbols that are recognized as operators are automatically surrounded by a small amount of whitespace.

A sequence of letters is interpreted as a word. These are printed in italics by default. A sequence of numbers, possibly followed by an exponent are interpret as numbers. The following line results in the output shown in figure 5.2.

```
\eqn{123e-456}
```

The **frac**, **sqrt**, **har** and **par** commands can be used to typeset (square) roots, fractions, to put hats on top of expressions and, and to surround an expression in parentheses.

5.2 Fonts

Four fonts are available, as illustrated in figure 5.3. The input used to generate the figure speaks for itself:

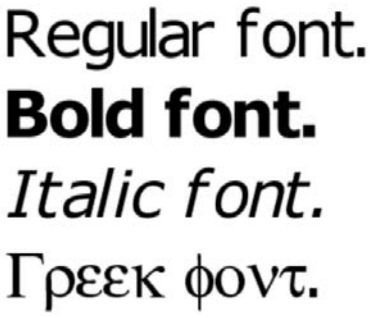


Figure 5.3: The four fonts.

Greek symbol	command	ASCII mapping
α	alpha	a
β	beta	b
γ	gamma	g
δ	delta	d
ϵ	epsilon	e
ζ	zeta	z
η	eta	h
θ	theta	q
ι	iota	i
κ	kappa	k
λ	lambda	l
μ	mu	m
ν	nu	n
ξ	xi	x
o	omikron	o
π	pi	p
ρ	rho	r
σ	sigma	s
τ	tau	t
υ	upsilon	u
ϕ	phi	f
χ	chi	c
ψ	psi	y
ω	omega	w

Figure 5.4: Mapping of the Greek to the ASCII alfabet.

```
\regular{Regular font.}\newline
\bold{Bold font.}\newline
\italic{Italic font.}\newline
\greek{Greek font.}
```

If you find **regular**, **bold**, **italic**, **greek**, too long to type, you may consider using the equivalent abbreviations **fr**, **fb**, **fi**, **fg**, which stand for 'font' followed by the first letter of the name of the font.

Of couse, fonts can be embedded in each other as with the **txt** and **eqn** modes. So the following lines also result in the output shown in figure 5.3:

```
\regular{Regular font.\newline
\bold{Bold font.\newline
\italic{Italic font.\newline
\greek{Greek font.}}}}
```

The default font in math mode is italic for letters (variables) and regular for all other symbols.

The greek alphabet is mapped to the ASCII alfabet as shown in figure 5.4. Their are two ways to produce greek characters. The first way is to switch to the **greek** font, the second is to use the command supplied for each character, e.g., **alpha** to produce the symbol 'α':

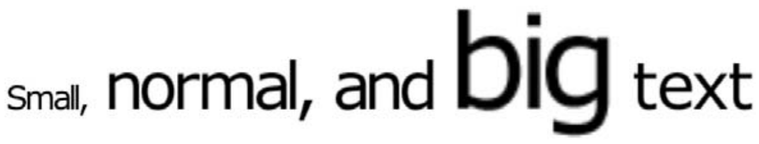


Figure 5.5: Scaling the font using the **scale** command.

```
\greek{a}
```

```
\alpha
```

The greek symbol commands work in both **txt** and **eqn** modes.

5.3 Scaling of fonts

The typesetting system always start typesetting with a default font size. The size of the font can change implicitly through the use of certain commands and constructions. For example, the size of the font of a sub- or superscript is 0.6× the size of the parent text.

You can explicitly scale the size of the font by using the **scale** command, as shown in the next example (output is in figure 5.5:

```
\scale{0.5}{Small,} \scale{1.0}{normal, and}
\scale{2.0}{big} text
```

5.4 Forced whitespace, forced newlines

To force whitespace between words or at the end of a line, use the **ws** command. Suppose the typesetting system typesets something badly, e.g.:

```
\italic{ape}:nut
```

(see output in figure 5.6a), then you could use whitespace commands to force some space at both sides of the colon:

```
\italic{ape}\ws{0.05}:\ws{0.1}nut
```

(see output in figure 5.6b)

If you don't supply an argument to the **ws** command, a default of 0.25 is used.

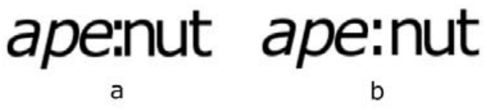
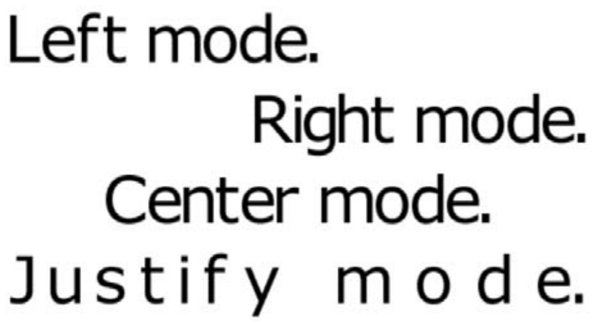


Figure 5.6: Forcing whitespace at both sides of the semicolon using the **ws** command.



Left mode.

Right mode.

Center mode.

Justify mode.

Figure 5.7: The four alignment modes.

Newline are automatically inserted when text is too wide to fit on the current line. To force a newline to be inserted, you can use the **newline** command, as was already shown in the four fonts example above (section 5.2). Two backslashes is a short synonym for a newline.

5.5 Alignment

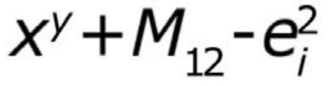
Four alignment modes are offered to place text on a line as required, and to fill out the line if required.

The first mode is **left**. This places all text as far to the left as possible. **right** modes places all text as far to the right as possible. **center** modes places all text in the center of the line. **justify** mode spaces all text such that it nicely fits the maximum width of the line. The ouput in figure 5.7 was generated from the following input:

```
\left{Left mode}\newline
\right{Right mode}
\newline\center{Center mode}
\newline\justify{Justify mode}
```

5.6 Sub- and superscript

In **eqn** mode, sub- and superscripts can be added to the previous 'word' in the sentence. The syntax of the sub- and superscript constructors is pretty straightforward, and should become clear from the following input and its output (figure 5.8):



$$x^y + M_{12} - e_i^2$$

Figure 5.8: Script example.

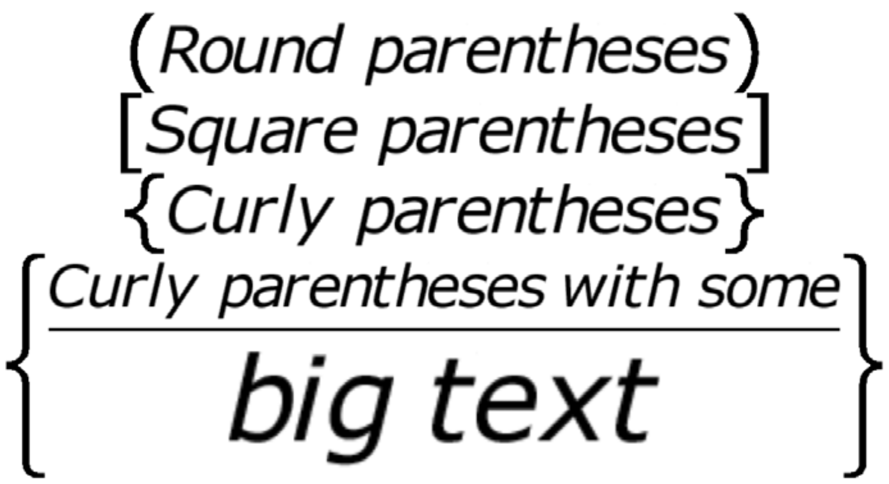


Figure 5.9: The three types of parentheses, and an example of how parentheses scale with their contents.

```
\eqn{x^{y} + M_{12} - e_{1}^{2}}
```

If you try to append multiple subscripts or multiple superscripts to a single word, the system will complain and ignore everthing but the first sub- or superscript.

5.7 Parentheses

Three types of parentheses can be placed around pieces of text in **eqn** mode: square parentheses [], round parentheses () and curly parentheses {}. The brackets automatically scale with the size of the input. Here are some examples of their use, the output is in figure 5.9:

```
\center{
\eqn{\par{()}{\txt{Round parentheses}}}\newline
\eqn{\par{[]}{\txt{Square parentheses}}}\newline
\eqn{\par{\{\}}{\txt{Curly parentheses}}}\newline
\eqn{\par{\{\}}{\frac{\txt{Curly parentheses with some}}
{\scale{2.0}{\txt{big text}}}}}}
```

As you can see, the first argument to the **par** command is the parentheses you would like. These can be any mix of types, e.g.: [], (] or even something weird like }{. The second arguments specifies the contents of the parentheses. Note that you always have to be in **eqn** mode (for no apparent reason...), but you can cheat you way around this by including **txt** commands inside the parentheses command.

5.8 Tabulars

Tabulars can be used for all kinds of purposes where you want to arrange text in row and columns. Examples of their use are tables and matrices. We will give an example of both.

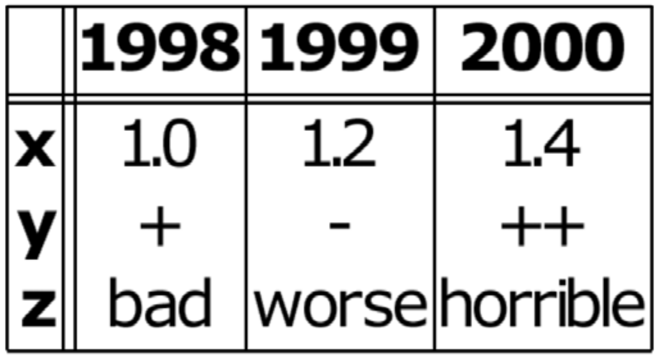
The **tabular** command first argument specify the number of columns, the alignment of content inside the columns, and optional vertical lines between them. All following arguments specify rows. Inside a row, each column is seperated by an ampersand: ’&’. The alignment of a column can be Here is an example of a table with 4 columns and 4 rows (output in figure 5.10):

```
\tabular{|r||c|c|c|}
{\hline}
{ &\bold{1998}&\bold{1999}&\bold{2000}}
{\hline}
{\hline}
{\bold{x}&1.0&1.2&1.4}
{\bold{y}&+&-&++}
{\bold{z}&bad&worse&horrible}
{\hline}
```

The width of a column is determined by the width of the widest item in the column. The alignment can be ’l’, ’r’, ’c’ or ’j’ (left, right, center, or justify), and behaves as described in section 5.5. A vertical bar ’—’ in the first argument produces a vertical line. A row containing only an **hline** command produces a horizontal line. Empty or unspecified entries in row are filled with blanks; surplus columns in row are ignored with a warning issued.

Now for an example of a matrix:

```
\eqn{\bold{M} =
\par{[]}{
\tabular{cccc}
{\fr{R_{11}}\ws{0.2}&\fr{R_{12}}\ws{0.2}&\fr{R_{13}}\ws{0.2}&0}
{\fr{R_{21}}&\fr{R_{22}}&\fr{R_{23}}&0}
{\fr{R_{31}}&\fr{R_{32}}&\fr{R_{33}}&0}
```



	1998	**1999**	**2000**
x	1.0	1.2	1.4
y	+	-	++
z	bad	worse	horrible

Figure 5.10: Tabular example 1.

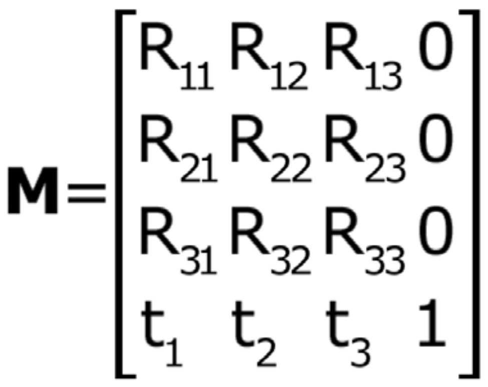


Figure 5.11: Tabular example 2.

```
{\fr{t_{1}}&\fr{t_{2}}&\fr{t_{3}}&1}
}}
```

This produces figure 5.11. Note the use of the **ws** command to force some extra whitespace between columns.

5.9 (Square) roots

The construct a (square) root, use the **sqrt** command. It takes at least one argument (the contents of the root), and optionally a second argument, which is the 'power' of the root. For example:

```
\eqn{\sqrt{x + y}\ws{1.0}
\sqrt{\pi}{3}\ws{1.0}
\sqrt{\fb{Z}}{\scale{0.5}{\frac{1}{2}}}}
```

This produces the output shown in figure 5.12. As you can see in the last line, sometimes you have to manually scale the font size of the 'power' if you put something weird or large in there.

5.10 Fractions

To construct complicated fractions where the numerator and the denominator are separated by a horizontal line, use the **frac** command. It takes two arguments (the contents of the numerator and the denominator). The size of the

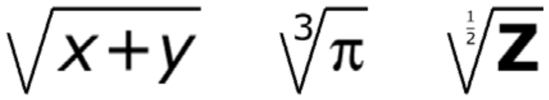


Figure 5.12: Square root example.

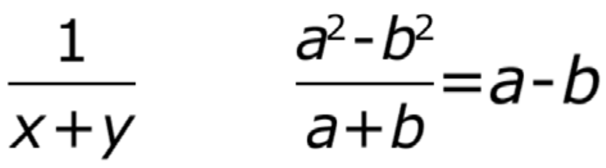


Figure 5.13: Fraction example.

font is implicitly scaled by 0.9 in a fraction, but this can be corrected by using the **scale** command (section 5.3). The output in figure 5.13 was generated from the following input:

```
\eqn{\frac{1}{x+y}\ws{2.0}
\frac{a^{2} - b^{2}}{a + b} = a - b}
```

5.11 Hats

You can add hats, bars and tildes on top of text by using the **hat**, **widehat**, **bar**, **widebar**, **tilde** or **widetilde** commands. The **wide** variants scale with the size of the content, as is illustrated by the following examples, and its output in figure 5.14.

```
\eqn{
\hat{A}\ws{1.0}\tilde{\frac{x}{y}}\ws{1.0}
\widehat{A + B + C}\ws{1.0}\widebar{A + B + C}}
```

All types of hats look rather bad when (OpenGL) antialiasing is turned off.

5.12 Colors

Text can be typeset in arbitrary colors. A fixed number of colors is known to the system, and custom colors can be defined using a special command described below. The default colors are: red, green, blue, magenta, yellow, cyan, black, white, grey and gray (these last two are synonyms).

The draw text in one of these colors (the default is black by the way), do something like the following (output in figure 5.15):

```
Black text, \red{red text}, \blue{blue text}
\newline and a green \green{frog}.
```

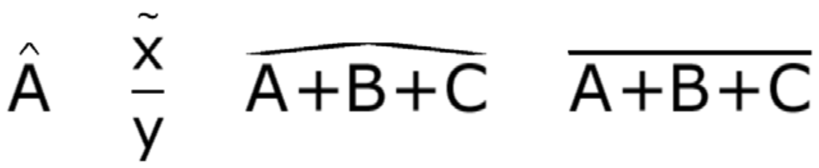


Figure 5.14: Hats example.

Figure 5.15: Color example.

5.12.1 Custom colors

Custom colors can be added using the **newcolor** command. The command takes one argument, which is a sentence that contains the name of the color, the red value (range: [0.0, 1.0]), the green value and the blue value, seperated by whitespace:

```
\newcolor{brightpink 1.0 0.0 0.6}
```

Once a new color is defined like this, it can be used as any other color:

```
\brightpink{Some bright pink text.}
```

If a color is already defined, issuing a **newcolor** command for it will override its current rgb value. Redefining an existing command as a color is possible but not recommended.

5.13 Custom commands

When you want to do the same thing a lot of times, it might be useful to create a custom command or 'macro'. A custom command defines a fixed input string, with variable arguments. An example might be when you always want to typeset your geometric algebra multivectors in bold font. You can define a new command **mv** (short for multivector), and use that everytime you want to typeset something as a multivector. Custom commands can save you a lot of typing work, make your input more readible, and if you later change your mind and decide that all you multivectors should be typeset in italics, it is

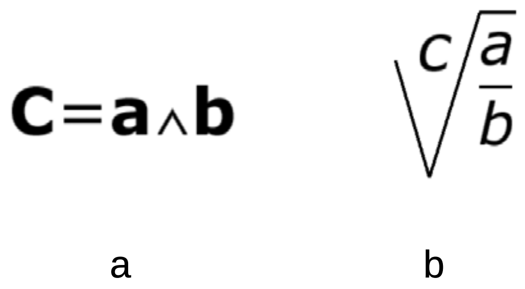


Figure 5.16: Custom command example.

easier and less error-prone to change a single custom command definition, than to replace every occurance of the **bold** command.

Here is an example of defining and using a custom command:

```
\newcommand{\mv}{\bold{\1}}
\eqn{\mv{C} = \mv{a} \op \mv{b}}
```

The output is in figure 5.16a. The first argument to the **newcommand** command is the name of the the new command (**mv** in this case). The second argument is what the command should be expanded to. Inside the second argument, a backslash followed by an positive number i will be replaced i^{th} argument of the new command. The **mv** command takes only one argument, but next we give an example of multiple arguments (output in figure 5.16b).

```
\newcommand{\wc}{\eqn{\sqrt{\frac{\1}{\2}}{\3}}}
\wc{1}{2}{3}
```

Note that the **eqn** command is included inside the custom command, to force equation mode every time the command is used (otherwise the **frac** command may not work). This is like **ensuremath** in Latex.

Commands with no arguments are also possible by simply not using any 'backslash positive numbers' inside the custom command string.

If a command is already defined, issuing a **newcommand** command for it will override its current value. Redefining an existing predefined command (such as **txt** or **par** is possible but not recommended.

5.14 Special symbols

Several special symbols for geometric algebra use are available, and more can be added quite easily as required. Currently the following symbols are available through commands:

Command	Symbol
gp	half space
op	$\wedge$
ip	$\cdot$
lc	$\rfloor$
rc	$\lfloor$
cp	$\times$
infty or **inf**	∞

Accessing the greek special characters directly was already discussed in section 5.2. Most ASCII characters that are not numbers or letters can also be accessed by typing a backslash followed by the character. For some characters, this is the only way of producing them, since they are reserved (e.g., the backslash).

Bibliography

[1] D. Fontijne, T. Bouma, L. Dorst *Gaigen: A Geometric Algebra Implementation Generator.* Available at http://carol.science.uva.nl/~fontijne/gaigen