Gnu Triangulation Software for *rlabplus* Version 1

Reference Manual

Marijan Koštrun, mkostrungmai.com

Printed February 28, 2026

Citation Info

If you find libgts for `rlab2` or `rlab3` useful, you may cite them in your research work as following:

> Marijan Koštrun, *rlabplus* (`rlab`2.2 through 2.5; `rlab3`), `http://rlabplus.sourceforge.net`, 2004-2026. Ian Searle and coworkers, `rlab` (through `rlab2.1`), `http://rlab.sourceforge.net`, 1992-2001.
> Stéphane Popinet and contributors, The GNU Triangulation Software. Version 0.7.6. March. 29, 2006. URL: `https://gts.sourceforge.net`, and various `github.com` releases the latest being tagged 2012-07-08.

Contents

Chapter 1

libgts - the GTS library for rlab

1.1 Introduction

From Ian Searle's RLAB Reference Manual, cir. 1999, pp.11-12.

Rlab stands for "our" lab. It is available to almost everyone who needs a computational tool for scientific and engineering computationss, because it is freely available and runs on many platforms.

For many years scientists and engineers have developed and implemented programs in low-level programming languages such as algol, fortran, pascal and c. As computers got faster, and more people program them, high level languages have become more popular. Most high level languages are tailored towards specific task, or class of tasks, this is not shortcoming, but a direct consequence of the high level nature of language. If a language is supposed to be easy, convenient, for a particular set of tasks, some assumptions about those tasks must be made. There is a cost associated with the assumptions made with any high level language. In the realm of scientific languages the price is usually slower program execution time. However, as tasks become more complex, and computers get faster, the penalty for slower execution time is less. Often the slower execution time is more than compensated for by the significantly reduced development time.

Rlab, a high level language for engineers and scientists makes several assumptions:

- The user is interested in computational exercise. Rlab offers if any convinience for those who need help with symbolic programming.
- The operations, and conventions of linear algebra are useful in accomplishing most tasks. Rlab offers simple access to the most popular linear algebra libraries, BLAS and LAPACK. Furhtermore, Rlab's basic data structures are matrix oriented, with the vector dot-product an integral part of the built-in operations.
  Due to the array oriented operations, and the high-level interface to FFTPACK, and a discrete IIR filtering function, Rlab serves well as an environment for signal analysis and exploration.
- The language should be simple, yet predictable, with its origins in popular scientific languages that most engineers/scientists are likely to already be familiar with. Thus Rlab takes after the C and FORTRAN programming languages, and to some extent Matlab, a popular high level matrix language.

I hope you find it useful . . .

1.1.1 About `rlab`

Rlab does not try to be a Matlab clone. Instead, it borrows what I believe are the best features of the Matlab languageand provides improved language syntax and semantics. The syntax has been improved to allow users more expression and reduce ambiguities. The variable scoping rules have been improvedto facilitate creation of larger programs and program libraries. A heterogeneous associative array has been added to allow users to create and operate on arbitrary data structures.

1.1.2 About *rlabplus*

rlabplus provides the third release of the environment for 32- and 64-bit linux systems on Intel and ARM and RaspberryPi architectures. The environment integrates large number of numerical solvers and functions from various sources, most notably from the Gnu Scientific Library (GSL) and from the netlib. Within the environment it is possible to visualize data using gnuplot, xmgrace, and pgplot xor plplot; get and post data using uniform resource locator implementing HDF5 or world wide web; and control serial, GPIB or TCP/IP connection. `rlab3` supports embedded python, java and ngspice interpreters.

1.2 Organization of libgts shared object library in *rlabplus*

The library source tree is organized as follows. Say user's name is *user* and their home directory is `/home/user/`:

- `rlab/` user specific `rlab` system directory
  - `lib.so/` user specific `rlab` system directory
    - `libgts.so.r3` *link to the same named file in `gts/` subdirectory*
    - `gts/`
      - `convhull_3d.h` *convex hull library*
      - `doc/` *location of this manual*
      - `gts/` *link to mk's version of gts-snapshot-121130*
      - `gts-0.7.6-mk/` *latest gts with mk mods*
      - `gts-snapshot-121130/` *latest gts*
      - `include` *gts headers pulled here for rlab build*
      - `libgts.a` *gts compiled for inclusion with rlab*
      - `libgts.so.r3` *loader of rlab functions for gts*
      - `r_gts_intersection.c` *add-in functionality*
      - `r_gts_util.c` *rlab library utility functions*
      - `rlab` *example scripts, see further sections of the manual*
      - `rlabplus_gts.c` *rlab gts library main body*
      - `rlabplus_gts.h` *its headers*
      - `rlabplus_lib64gts.so` *built shared object library for rlab*
      - `rmake` *rlab specific build script for gts and rlab link*

Please note, in *rlabplus*, `rlab/lib.r` and `rlab/lib.so` are system directories. Typically, the library is downloaded as a tarred archive, which should be unpacked in user's `rlab/lib.so/` directory. There it will create `gts` subdirectory, which content is given above.

1.2.1 Installation

Pull link to `libgts.so.r3` one directory up, so it appears in `rlab/lib.so`.

1.2.2 How to Use the Library

Put the line

```
rfile libgts.so.r3
```

at the beginning of your script. If the library was installed properly, the functions provided by the library should be available for use after you run your script the first time.

Note: As a part of *rlabplus* two libraries are distributed with the system `rlab`, `libgeomview.r3` and `libgts.r3`. User should familiarize themselves with their content. In particular, `geomview` utility should be installed on the system and available from command line: `rlab` library uses it to visualize `rlab`/GTS data structures. Also, the system library `libgnuplot.r3` can take rlab/GTS data structures and plot them.

1.3 System script library `libgts.r3`

1.3.1 validgts

Format: $u = validgts(s)$

Arguments:

1. $s =\ll v; e; f; c \gg$, an `rlab` list containing GTS compliant surface.

Result: $0, 1$ depending whether the `rlab` variable s is contains GTS compliant surface or not.

Note: This function is a part of `libgts` library which is distributed with `rlab` irrespectively of shared object library.

1.3.2 __gts__to__xyz

Format: $A = \_gts\_to\_xyz(s)$

Arguments:

1. $s =\ll v; e; f; c \gg$, an `rlab` list containing GTS compliant surface.

Result: Internal function used for plotting programs to plot GTS surface in 3-D. Currently, Gnuplot is supported for both, plotting the 3-D meshes of surfaces (as chopped lines), and their sides (as 'polygons'). In Fig. (1.1) one can see a Gnuplot generated output of a few surfaces.

Note: This function is a part of `libgts` library which is distributed with `rlab` irrespectively of shared object library.

1.4 System script library `libgeomview.r3`

About: Geomview is an interactive 3D viewing program. Geomview lets you view and manipulate three-dimensional objects: you use the mouse to rotate, translate, zoom in and out, and so on. Geomview can be used as a standalone viewer for static objects, or as a display engine for other programs which produce dynamically changing geometry. Geomview can display objects described in a variety of file formats. Geomview comes with a wide selection of example objects, and you can create your own objects too.

1.4.1 plot_gts_surface

Format: $geomview.plot\_gts\_surface(scene, obj)$

Arguments:

1. scene, string, GCL compliant name of the scene.
2. obj, string, name of the valid surface object from GTS workspace.

Abstract: Plot an GTS surface object on a scene using `geomview`. In Fig. (1.1) one can see a Gnuplot generated output of a few surfaces.

1.4.2 snapshot

Format: $geomview.snapshot(fn/, fmt/)$

Arguments:

1. fn, string, filename in which the current Camera view will be saved in post-script format.
2. fmt, string, “ps” or “ppm” for two image formats that geomview supports.

Abstract: Plot current camera view from geomview to a file in post-script format.

Note: In Fig. (1.1) one can see a Gnuplot and `geomview` generated outputs of few surfaces.

1.5 `rlab` library `rlabplus_lib64gts`

GTS stands for The GNU Triangulated Surface Library. It provides workspace/environment for creating and manipulating surfaces, and their elemental parts: triangles (faces), line segments (edges) and points (vertices).

The functions in this library allow `rlab` to create and manipulate single surfaces and perform boolean operations between the surfaces.

The library is organized around surfaces as named objects in the GTS-managed workspace. User can create, delete or manipulate surface(s) as one could do that within the GTS. In addition there exist functions which export named GTS surfaces into `rlab` variables, or convert proper `rlab` variables into GTS surface objects. In that sense, `rlab` approach is different from, say python approach as in `pygts`, in which the GTS objects are tied to python variables.

All functions in this library are in a list, `gts.surface` to emphasize the `rlab` approach as named-surfaces centric.
In `rlab` parlance a GTS compliant surface s is a list $s = \ll v; e; f; c \gg$, which entries are

- v, 3-col matrix of vertices comprising the surface.
- e, 2-col integer matrix of edges comprising the surface, where the row entries are the indices of its vertices in v.
- f, 3-col integer matrix of the triangles comprising the faces of the surface, where the row entries are the indices of its edges in e.
- c, 3-col matrix containing the color of each or of all faces. If omitted the default value $[1, 1, 1]$ is used.

Administrative Functions

1.5.1 ls

Format: $u = gts.surface.ls()$

Arguments:

Result: u, string array, names of surfaces currently defined in the GTS workspace.

1.5.2 info

Format: $u = gts.surface.info(sname)$

Arguments:

1. *sname*, string, name of the surface.

Result: u, list $\ll nr\_edge; nr\_face; nr\_vertex \gg$, count of edges, faces and vertices of the surface which name is given by the variable *sname* in the GTS workspace.

1.5.3 stat

Format: $u = gts.surface.stat(sname)$

Arguments:

1. *sname*, string, name of the surface.

Result: u, list $\ll n\_boundary\_edges; n\_duplicate\_edges; n\_duplicate\_faces; n\_incompatible\_faces; n\_non\_manifold\_edges; nr\_edge; nr\_face; nr\_vertex \gg$, which provides some statistical properties of the vertices, edges and faces comprising the surface which name is given by the variable *sname* in the GTS workspace.

1.5.4 qstat

Format: $u = gts.surface.qstat(sname)$

Arguments:

1. *sname*, string, name of the surface.

Result: u, list $\ll edge\_angle; face\_area; face\_length; face\_quality \gg$, which provides some more statistical properties of the vertices, edges and faces comprising the surface which name is given by the variable *sname* in the GTS workspace. Each entry in the list is another list with entries *max*, *min*, *avg*, *std* and *n* with obvious interpretation.

Functions for Definition of Surfaces

1.5.5 new

I. **Format:** $u = gts.surface.new(s/, prop/)$

   Arguments:

   1. *s*, string, unique name of the surface to be created in the workspace. If the surface with that name already exists in the workspace, its elements are deleted and the surface with new properties put in its place.
   2. *prop*, list, properties that describe the surface. In its absence, and empty surface is created. The expected entries of the list depend on its most important entry *desc*
      - *desc = "sphere"*
        - *level*, integer, level of detail of the sphere (default is 4).
      - *desc = "plane"* or *"rectangle"*
        - *bbox_x*, *bbox_y*, and *bbox_z*, of which two are pair of values, and one of them has to be single value. Creates rectangle in one of the planes $x = const$, $y = const$ or $z = const$, where the range is determined by the bounding box values.
      - *desc = "box"*
        - *bbox_x*, *bbox_y*, and *bbox_z*, all of are pairs of values. Creates box with respective bounding boxes.
        - *remove_face*, integer array of 0's and 1's for each of 12 triangles making the surface of the box. It explicitely includes or excludes each face. If omitted, all faces are included.
      - *desc = "mesh"*

  - ∗ *x*, *y*, and *z*, where *x* is a real vector of length *nx* containing the x-coordinate of the mesh, *y* is a real vector of length *ny* containing the y-coordinate of the mesh, and *z* is a matrix of size *nx*-by-*ny* where $z[i;j] = z(x[i], y[j])$.

    Note: To exclude a rectangular area out of the surface - that is, to put a hole in the surface - choose the mesh appropriately, and set the values at the holes to `nan()`;

In addition, all surfaces allow specifications of the point-wise transforms, which are all applied after the surface is constructed in x,y,z coordinates.

- *transl*, real 3-vector, translation of the surface after construction.
- *center*, real 3-vector, presumed center of surface, which is invariant under all transformations except translation.
- *scale*, real 3-vector, scaling of the surface in each of the tree directions.
- *rot_x*, *rot_y*, *rot_z* - real 3x3 matrix, presumably of rotations along each of the 3 axes.
- *color*, real 3-vector, single property of the entire surface, presumably its color.

II. **Format:** $u = gts.surface.new(svar)$

Arguments:

1. *svar*, variable containing GTS-complient description of the surface in terms of a list with four entries:
   - *v*, 3-col matrix of vertices comprising the surface.
   - *e*, 2-col integer matrix of edges comprising the surface, where the row entries are the indices of its vertices in *v*.
   - *f*, 3-col integer matrix of the triangles comprising the faces of the surface, where the row entries are the indices of its edges in *e*.
   - *c*, 3-col matrix containing the color of each or of all faces. If omitted the default value $[1, 1, 1]$ is used.

Result: *u*, an integer, 0 for success, 1 for failure.

Abstract: *gts.surface.new* creates new surface in the GTS workspace with the given properties. The list $\ll v; e; f; c \gg$ can be created, e.g., by `svar=gts.surface.export(s)`.

```
>> gts.surface.new("s", <<desc="sphere";level=2>>);
>>
```

1.5.6 from_vertices

Format: $gts.surface.from\_vertices(sname, v, fv/, c/)$

Arguments:

1. *sname*, string, unique name of the surface to be created in the workspace. If the surface with that name already exists in the workspace, its elements are deleted and the surface with new properties put in its place.

2. *v*, 3-column real matrix of vertices comprising the surface.

3. fv, 3-column integer matrix of indices of vertices comprising surface. Notice fv is <u>NOT</u> a GTS compliant face.
   This is a helper function that makes it easier to construct a surface when only point information may be available.

Result: u, an integer, 0 for success, 1 for failure.

Abstract: Creates a surface *sname* in GTS workspace from the vertices and ad hoc face data.

1.5.7 readm,writem

Format: $u = gts.surface.readm(s, fn)$
Format: $u = gts.surface.writem(s, fn/, opts/)$

Arguments:

1. s, string, unique name of the surface to be created in the workspace. If the surface with that name already exists in the workspace, its elements are deleted and the surface with new properties put in its place.
2. fn, string, file name containing GTS-compliant surface definition. If writing to file then from the file name the format is derived: oogl and gts.

Result: u, an integer, 0 for success, 1 for failure.

Abstract: Loads/saves definition of surface in file.

1.5.8 export

Format: $s = gts.surface.export(sname)$

Arguments:

1. *sname*, string, unique name of the surface to be created in the workspace.

Result: $s = \ll v; e; f; c \gg$, an `rlab` list containing GTS compliant surface.

Abstract: Exports GTS surface to `rlab`, where so defined surface can be re-imported to the GTS, e.g., in this example by using `gts.surface.new(newname, s)`.

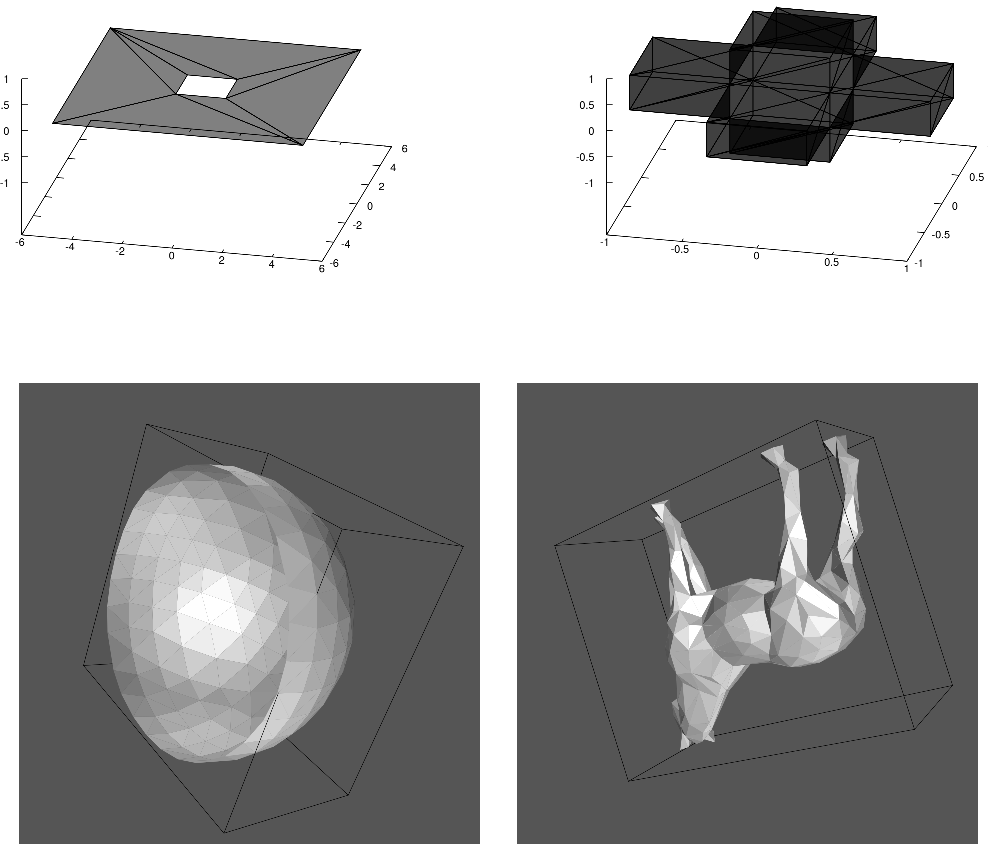

Figure 1.1: (Top row) Two examples of Gnuplot output of plotting, a rectangular surface with a rectangular hole (top left) and "cutter" surface (top right) provided with GTS source tree in directory `test/boolean/surfaces`. The mesh (black lines) is plotted as a 3-D scatter line plot, where user can specify line and point style in usual fashion, while the faces (gray surfaces) are plotted as Gnuplot objects 'polygons'. (Bottom row) Two examples of use of `geomview` for rendering 3-D plots, shaken hemisphere (bottom left); and horse4 (bottom right) in random rotation, also from `test/boolean/surfaces`.

Surface Manipulation Functions

1.5.9 foreach_vertex, foreach_edge, foreach_face

Format: $u = gts.surface.foreach\_vertex(s, vertexfn/, fnparams//, /opts/)$

Arguments:

1. *s*, string, name of the surface in the GTS workspace.
2. *vertexfn*, name of function that will be called for each vertex of the surface *s*. The function has to be defined as

   ```
   vertexfn = function (vertex/, fnparams/) { ... };
   ```

   where `vertex` is a 3-vector, and *fnparams* are the optional parameter passed to the function. If function returns a value, then the result of the last call of the function `vertexfn` will be returned upon traversing all the vertices.
3. *fnparams*, parameter that is passed directly to the function *vertexfn*.
4. *opts*, list of options. Suported options are
   - *modify_obj*, integer 0 or 1. If set to 1 then it expects that the vertexfn returns a 3-vector which will then replace the function argument vertex in the surface (replaces the vertex or the points in the edge, or points in the triangle with the result of the function call on the vertex, or edge, or triangle.
   - *stack_pts*, integer 0 or 1. If iterating over edges (triangles) choose if the function argument is going to be two points as row 6-vector (three points as row 9-vector), or two (three) points stacked on top of each other, that is a matrix with two (three) rows and three columns.

Result: u, depends whether *modify_obj* is set or not. If set, then returns 0 upon successfull completion of iteration. Conversely, if *modify_obj* is unset then the result of the last function call is returned. For obvious reasons, one cannot combine the two, that is, the solver cannot modify vertices (edges, faces) and return value.

1.5.10 foreach_face_remove

Format: $gts.surface.foreach\_face\_remove(sname, decidefn/, fnparams, opts/)$

Arguments:

1. *sname*, string, name of the surface in the GTS workspace.
2. *decidefn*, name of the function that will be called on each face of the surface *s*. The function has to be defined as

   ```
   decidefn = function (face/, fnparams/) { ... };
   ```

   where `face` can be a 9-vector containing the three vertices of the face, or a matrix 3-by-3 where each row is a vertex of the face. *fnparams* are the optional parameter passed to the function. The function returns 0 if the face stays with the surface, or 1 if the face is to be removed from the surface..

3. *fnparams*, parameter that is passed directly to the function *vertexfn*.

4. *opts*, list of options. Suported options are

   - *stack_pts*, integer 0 or 1. If iterating over triangles choose if the function argument is going to be three points as row 9-vector, or three points stacked on top of each other, that is a matrix with three rows and three columns.

Abstract: The solver iterates over all faces of the named surface *sname*, and for each face it executes the `rlab`-function *decidefn*. If *decidefn* returns 1 for particular face, this face is removed from the surface.

1.5.11 add_face

Format: $i = gts.surface.add\_face(sname, tri/, opts/)$

Arguments:

1. *sname*, string, name of the surface.

2. *tri*, 9-vector or 3x3 matrix, vertices of a face to be added to the surface.

3. *opts*, list of options, Currently supported is entry *color*, 3-vector of the color associated with the face.

Result: i, integer 0 or 1, if surface existed and the new face has been added, or if surface did not exist, and was created comprising the single face.

Abstract: Adds face to the surface: a surface is created comprising single face from three provided vertices, This surface is then merged with the surface named in *sname*.

1.5.12 remove_face

Format: $i = gts.surface.remove\_face(sname, tri)$

Arguments:

1. *sname*, string, name of the surface.

2. *tri*, 9-vector or 3x3 matrix, vertices of a face to be added to the surface.

3. *opts*, list of options, Currently supported is entry *color*, 3-vector of the color associated with the face.

Result: i, integer 0 or 1, if surface existed and the new face has been added, or if surface did not exist, and was created comprising the single face.

Abstract: Remove face from the surface by first adding the vertices provided in the array *tri*.

Note: This operation is 50% or better likely to succeed. Sometimes it just silently fails.

1.5.13 remove_surface

Format: $gts.surface.remove\_face(sname, rsname)$

Arguments:

1. *sname*, string, name of the surface.
2. *rsname*, string, name of the surface.

Abstract: Remove second surface from the first surface.

Note: This operation is 50% or better likely to succeed. Sometimes it just silently fails.

Example: Idelized ice cream cone created in `rlab` and visualized and printed through `geomview` interface. This script is provided in the test archive as `gts/rlab/eg10.r3` The geomview exported its rendition as postscript file, which was then converted to pdf for inclusion in this manual. This is given in Fig. (1.2).

```
rfile libgeomview
rfile libgts.so

gnuwins(2);

// ice cream cone parameters
b = 1;  // bottom radius - also radius of the ball
h = 4;  // height

A_0 = [0,0,h];


//
// construct half-sphere in space z<=0
//
// 1. create sphere of radius b
gts.surface.new("ice_cream_cone", <<desc="sphere";level=7;scale=b>>);
face_remove_fn = function(fc)
{
  yes_no = any(fc[;3,6,9]>0);
  return yes_no;
};
// remove all points which z>0
gts.surface.for_each_face_remove("ice_cream_cone", face_remove_fn);
// find all points which are on the rim: z=0
ice_cream_cone = gts.surface.export("ice_cream_cone");
idx_vrim = find(ice_cream_cone.v[;3] == 0);
if (isempty(idx_vrim))
{
  stop("Calculation failed: change == to something more robust!\n");
}
fi = atan2(ice_cream_cone.v[idx_vrim;2],ice_cream_cone.v[idx_vrim;1]);
s_fi_idx = sort(fi).idx;
for (i in range(s_fi_idx))
{
  ip1 = ifelse(i>=length(s_fi_idx),i+1-length(s_fi_idx),i+1);
  // construct a triangle to be added to the surface:
  //  v[i], v[ip1], A_0
  i_1 = idx_vrim[s_fi_idx[i]];
  i_2 = idx_vrim[s_fi_idx[ip1]];
  gts.surface.add_face("ice_cream_cone",...
      [ice_cream_cone.v[i_1;],ice_cream_cone.v[i_2;],A_0] );
}
```

```

geomview.plot_gts_surface("scene","ice_cream_cone");
pause("Rotate 3D view to choosing, then press enter");
geomview.snapshot("ice_cream_cone.ps");
```

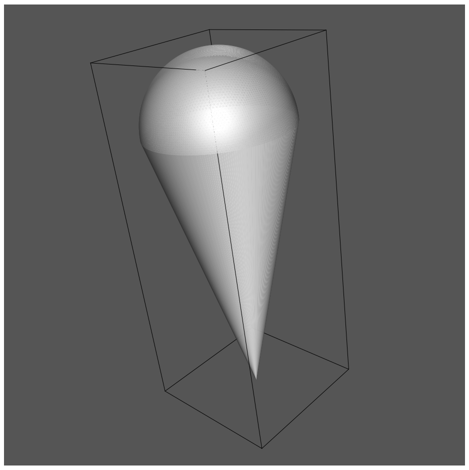

Figure 1.2: Idealized ice cream cone.