

FEATPOST macros

L. Nobre G., <http://matagalatlante.org>

January 2006

Abstract

This document intends to be an explanation of the MetaPost macros defined in the FEATPOST package. Its purpose is to draw two or three-dimensional physics diagrams. The FEATPOST package is supposed to help you draw figures containing 3D dots, vectors, flat arrows, angles, parametric lines, circles, ellipses, cones, cylinders, spheres, globes, hemispheres, toruses, elliptical frusta, polygons, polyhedra, functional and parametric surfaces, direction fields, field lines and trajectories in vector fields, schematic automobiles, electric charges, etc. The fact that it is a programming language instead of Computer Aided Design (CAD) helps the user to experiment different figure layouts without changing specified geometric relationships among figure elements. Two of the intrinsic MetaPost features which may be important for physics diagrams are: (i) the typesetting power of TeX is easily called when needed and (ii) besides a sufficient number of mathematical operators, geometric relationships may be expressed by linear equations, without explicit assignments.

1 Introduction

3D in MetaPost is not a new idea. Denis Roegel

[dog](#)

contributed related packages to CTAN some years ago. It was the subject of a 1997 TUG-boat article (V18, N4, 274-283). Recently he developed *MP2GL* <http://www.loria.fr/~roegel/TeX/mousson2005.pdf>. Anthony Phan has been developing a very elegant package called *m3D* <http://www-math.univ-poitiers.fr/~phan/m3Dp>. Due to his skilled coding, *m3D* can handle larger objects and can produce more realistic renderings than FEATPOST. Other possibilities include *GNU 3DLDF* <http://directory.fsf.org/GNU/3DLDF.html>, *Bob's Info* <http://robert.silve.free.fr/latex/metapost3D/> or *Asymptote* <http://asymptote.sourceforge.net/>. In any case we are talking about vector-based abstract diagrams, so the functionality of these packages is nearer to *GNU PLOT* than to *OpenGL*. We could also talk about *POV-ray* <http://www.povray.org/> or *blender* <http://www.blender3d.org/> but these are focused on ray-traced images, a completely different thing.

Regarding software FEATPOST requires only MetaPost but recommends *LaTeX*, *bash*, *ImageMagick*, *ghostscript*, *Linuxdoc*, *Textutils*, *dvips*, *epstopdf*, *sed*, *gv*, *plaympeg*, *dog* <http://jl.photodex.com/dog/>, *galrey* <http://www.flexer.it/galrey/> and *xcmd* <http://lince.cii.fc.ul.pt>. Also, it is highly beneficial to be able to understand and cope with MetaPost error messages as FEATPOST has no protection against mistaken inputs. One probable cause of errors is the use of variables with the name of procedures, like

`X, Y, Z, N, rp, cb, ps`

All other procedure names have six or more characters.

The user must be aware that MetaPost has a limited arithmetic power and that the author has limited programming skills, which may lead to unperfect 3D figures, very long processing time or shear bugs. It's advisable not to try very complex diagrams at first and it's recommended to keep 3D coordinates near order 1 (default MetaPost units).

All three-dimensional FEATPOST macros are build upon the MetaPost `color` variable type. It looks like this:

`(red,green,blue)`

Its components may, nevertheless, be arbitrary numbers, like:

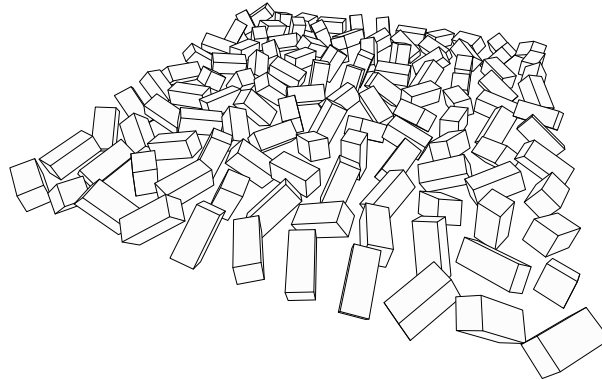


Figure 1: Example that uses `kindofcube`.

(X,Y,Z)

So, the `color` type is adequate to define not only colors but also 3D points and vectors.

2 Small Tutorial

One very minimalistic example program could be:

```
beginfig(1);
  cartaxes(1,1,1);
endfig;
end;
```

where `cartaxes` is a FEATPOST macro that produces the Cartesian referential.

One small example program may be:

```
f := 5.4*(1.5,0.5,1);
Spread := 30;
beginfig(1);
  numeric gridstep, sidenumber, i, j, coord, aa, ab, ac;
  color pa;
  gridstep = 0.9;
  sidenumber = 10;
  coord = 0.5*sidenumber*gridstep;
  for i=0 upto sidenumber:
    for j=0 upto sidenumber:
      pa := (-coord+j*gridstep,-coord+i*gridstep,0);
      aa := uniformdeviate(360);
      ab := uniformdeviate(180);
      ac := uniformdeviate(90);
      kindofcube( false, false, pa, aa, ab, ac, 0.4, 0.4, 0.9 );
    endfor;
  endfor;
endfig;
end.
```

where `kindofcube` is a FEATPOST macro that produces a rectangular prism (cuboid).

The main variable of any three-dimensional figure is the point of view. FEATPOST uses the variable `f` as the point of view. `Spread` is another global variable that controls the size of the projection.

Another example may be:

```
f := (13,7,3.5);
Spread := 35;
beginfig(1);
```

```

numeric i, len, wang, reflen, frac, coordg;
numeric fws, NumLines, inray, outay;
path conepath, cira, cirb, ella, ellb, tuba, tubb, tubc;
color axe, aroc, cubevertex, conecenter, conevertex;
color allellaxe, ellaaxe, ellbaxe, pca, pea, pcb, peb;
frac := 0.5;
len := 0.6;
wang := 60;
axe := (0,cosd(90-wang),sind(90-wang));
fws := 4;
reflen := 0.2*fws;
outay := 0.45*fws;
inray := 0.7*outay;
coordg := frac*fws;
NumLines := 30;
HoriZon := -0.5*fws;
setthestage( 0.5*NumLines, 2*fws );
cubevertex = (0.12*fws,-0.5*fws,-0.5*fws);
kindofcube(false,true,cubevertex,180,0,0,0.65*fws,0.2*fws,fws);
aroc := outay*(0,cosd(wang),sind(wang))-0.5*(0,fws,fws);
rigorousdisc( inray, true, aroc, outay, axe*len );
allellaxe := reflen*( 0.707, 0.707, 0 );
ellaaxe := reflen*( 0.707, -0.707, 1.0 );
ellbaxe := reflen*( -0.707, 0.707, 1.0 );
conecenter = ( coordg, coordg, -0.5*fws );
pca := ( coordg, -coordg, -0.5*fws );
pcb := ( -coordg, coordg, -0.5*fws );
pea := ( coordg, -coordg, 0.9*fws );
peb := ( -coordg, coordg, 0.9*fws );
cira := goodcirclepath( pca, blue, reflen );
cirb := goodcirclepath( pcb, blue, reflen );
ella := ellipticpath( pea, allellaxe, ellaaxe );
ellb := ellipticpath( peb, allellaxe, ellbaxe );
tuba := twocyclestogether( cira, ella );
tubb := twocyclestogether( cirb, ellb );
tubc := twocyclestogether( ella, ellb );
unfill tubb; draw tubb;
unfill tubc; draw tubc;
unfill tuba; draw tuba;
conevertex = conecenter + ( -3.5*reflen, 0, 0.8*fws );
verygoodcone(false,conecenter,blue,reflen,conevertex);
endfig;
end.

```

where we find a `rigorousdisc` and a `verygoodcone` (the nicest FEATPOST macros) in addition to `setthestage`, `twocyclestogether` and `kindofcube`.

2.1 Mechanics

The following is the sequence of steps necessary to produce the diagrams.

1. There are two methods that one can use to instruct MetaPost to process the TeX parts of the input file with LaTeX instead of plain TeX. One is to put a first line containing just

```
%LaTeX
```

in the file. This is the simplest method, but it doesn't work on all systems. Alternately, you can set the environment variable `TEX` to `"latex"`, like:

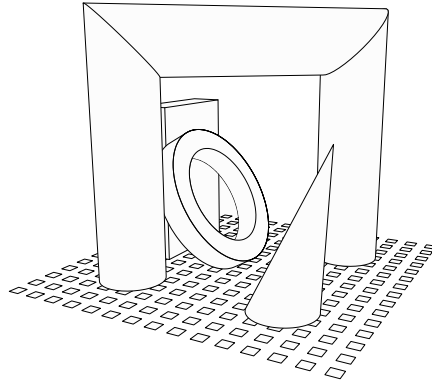


Figure 2: Example that uses `rigorousdisc` and `verygoodcone`.

```
export TEX=latex
```

or run `mpost` with a command-line option, like:

```
mpost --tex=latex myfile.mp
```

(this text was originally written by Brooks Moses)

2. Make your MetaPost program know FEATPOST macros. This may be accomplished in two ways:

- Insert the line:

```
input featpost3Dplus2D;
```

at the beginning of your program and then enter the shell command

```
mpost yourprogram.mp
```

. This works only if the file `featpost3Dplus2D.mp` is in a standard location.

- Pre-compile the macros into a `mem` file, for instance:

```
mpost -ini featpost
```

and then run your program with

```
mpost -mem featpost yourprogram.mp
```

3. If your program contains LaTeX text you may insert, at the beginning, the lines

```
verbatimtex
\documentclass{article}
\begin{document}
etex
```

and at the end

```
verbatimtex
\end{document}
etex
```

or make use of the

`latexmp` <http://www.ctan.org/tex-archive/graphics/metapost/contrib/macros/latexmp/> package.

4. Pass the produced figure(s) through `dvips` if they contain any text. You may do this with the provided shell script `laproof`.

```
laproof yourprogram N
```

where `N` is the figure number. This script produces an EPS figure that may be viewed with `gv` but that should not be inserted in LaTeX documents. Insert the original figure in LaTeX documents. Instead of `laproof` you may just as well use `MPS2EPS`

<http://www.ida.liu.se/~joned/download/mps2eps/>, `mproof` <http://www.ctan.org/tex-archive/graphics/mproof/> or `metapost.pl` <http://www.iee.et.tu-dresden.de/~fimmel/metapost.html> .

2.2 Main Features

2.2.1 Perspectives

FEATPOST can do three kinds of perspective.

```
f := ( 1.2 , 2.0 , 1.6 );
Spread := 75;

V1 := (1,1,1);
V2 := (-1,1,1);
V3 := (-1,-1,1);
V4 := (1,-1,1);
V5 := (1,1,-1);
V6 := (-1,1,-1);
V7 := (-1,-1,-1);
V8 := (1,-1,-1);
makeface1(1,2,3,4);makeface2(5,6,7,8);
makeface3(1,2,6,5);makeface4(2,3,7,6);
makeface5(3,4,8,7);makeface6(4,1,5,8);
makeline1(1,7);makeline2(2,8);
makeline3(3,5);makeline4(4,6);

beginfig(1);
  ParallelProj := true;
  SphericalDistortion := false;
  draw_all_test(red,true);
endfig;

beginfig(2);
  ParallelProj := false;
  SphericalDistortion := false;
  draw_all_test(green,true);
endfig;

beginfig(3);
  ParallelProj := false;
  SphericalDistortion := true;
  PrintStep := 5;
  draw_all_test(blue,true);
endfig;

end;
```

2.2.2 From 3D to 2D

The most important macro is `rp` that converts 3D points to two-dimensional (2D) rigorous, orthogonal or fish-eye projections. To draw a line in 3D-space try

```
draw rp(a)--rp(b);
```

where `a` and `b` are points in space (of `color` type). But if you're going for fish-eye it's better to

```
draw pathofstraightline(a,b);
```

If you don't know, leave it as

```
drawsegment(a,b);
```

2.2.3 Angles

When FEATPOST was created its main ability was to mark and to calculate angles. This is done with the macros `angline` and `getangle` as in the following program:

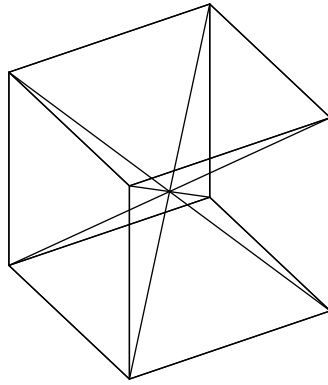


Figure 3: Orthogonal perspective.

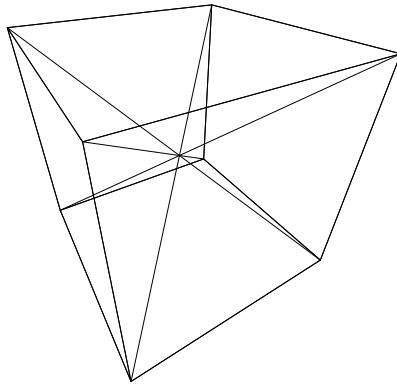


Figure 4: Rigorous perspective.

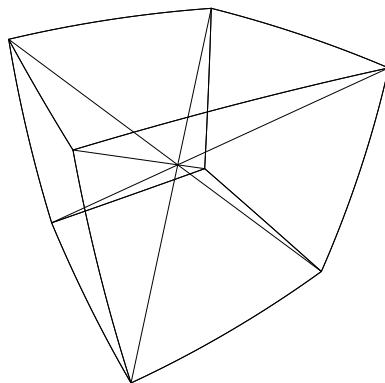


Figure 5: Fish-eye perspective.

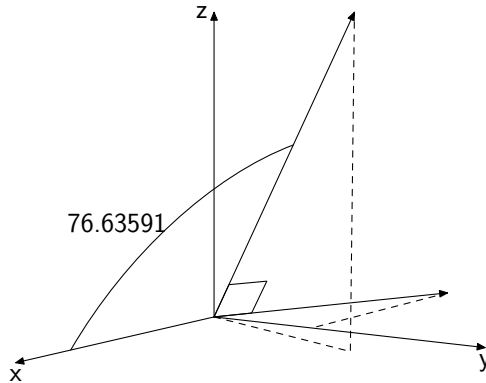


Figure 6: Example that uses `cartaxes`, `angline` and `getangle`.

```
f := (5,3.5,1);
beginfig(2);
  cartaxes(1,1,1);
  color va, vb, vc, vd;
  va = (0.29,0.7,1.0);
  vb = (X(va),Y(va),0);
  vc = N((-Y(va),X(va),0));
  vd = (0,Y(vc),0);
  drawarrow rp(black)--rp(va);
  draw rp(black)--rp(vb)--rp(va) dashed evenly;
  draw rp(vc)--rp(vd) dashed evenly;
  drawarrow rp(black)--rp(vc);
  squareangline( va, vc, black, 0.15 );
  angline(va,red,black,0.75,decimal getangle(va,red),lft);
endfig;
```

2.2.4 Intersections

The most advanced feature of FEATPOST is the ability to calculate the intersections of planar and convex polygons¹. It can draw the visible part of arbitrary sets of polygons as in the following program:

```
numeric phi;
phi = 0.5*(1+sqrt(5));
V1 := ( 1, phi,0);V2 := (-1, phi,0);
V3 := (-1,-phi,0);V4 := ( 1,-phi,0);
V5 := ( 0, 1, phi);V6 := ( 0,-1, phi);
V7 := ( 0,-1,-phi);V8 := ( 0, 1,-phi);
V9 := ( phi,0, 1);V10:= ( phi,0,-1);
V11:= (-phi,0,-1);V12:= (-phi,0, 1);
makeface1(1,2,3,4);makeface2(5,6,7,8);
makeface3(9,10,11,12);
beginfig(1);
  sharpraytrace;
endfig;
end
```

2.2.5 Coming back to 3D from 2D

It is now possible to do an "automatic perspective tuning" with the aid of macro `photoreverse` which is under development. Please, refer to example `photoreverse.mp` and to *FeatPost*

¹Unfortunately, this is also the most "bugged" feature.

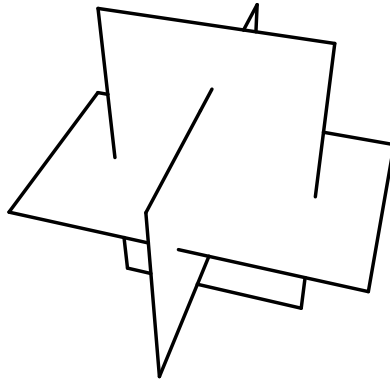


Figure 7: Example that uses `sharpraytrace`.

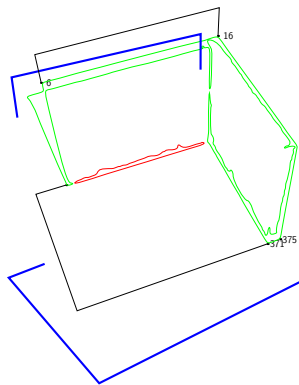


Figure 8: Example that uses `photoreverse`. It may not work when vertical lines are not vertical in average on the photo.

Deeper Technicalities <http://matagalatlante.org/nobre/hyt/technicaldrawfromphoto.html>.

Coming back to 3D from 1D Using the same algorithm of `photoreverse`, the macro `improvertex` allows one to approximate a point in 3D-space with given distances from three other points (an initial guess is required).

3 Reference Manual

Some words about notation. The meaning of macro, function, procedure and routine is the same. Global variables are presented like this:

```
vartype var, anothervar
anothervartype yetanothervar
```

Explanation of `var`, `anothervar` and `yetanothervar`. `vartype` can be any one of MetaPost types but the meaning of `color` is a three-dimensional point or vector, not an actual color like yellow, black or white. If the meaning is an actual color then the type will be `colour`. All global variables have default values.

Functions are presented like this:

- `returntype function()` Explanation of function. `returntype` can be any one of MetaPost types plus global, draw or drawlabel. `global` means that the function changes some of the global variables. `draw` means that the function changes the currentpicture. `drawlabel` means that the function changes the currentpicture and adds text to it.

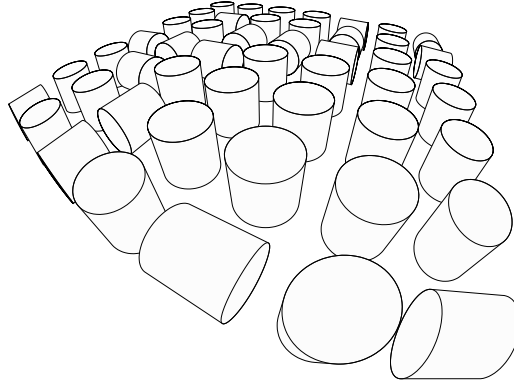


Figure 9: Figure that uses `SphericalDistortion:=true` and `rigorousdisc`.

1. `type1` Explanation of the first argument. The type of one argument can be any one of MetaPost types plus `suffix` or `text`.
2. `type2` Explanation of the second argument. There is the possibility that the function has no arguments. In that case the function is presented like "returntype **function**".
3. Etc.

3.1 Global variables

```
boolean ParallelProj
boolean SphericalDistortion
```

Kind of projection calculated by `rp`. By default projections are rigorous but if `ParallelProj` is set `true` then parallel lines remain parallel in the projection. It is the same as placing the point of view infinitely far without losing sight. If `SphericalDistortion` is set `true` there will be a distortion coming from: (i) the projection being done on a sphere of center `f` and (ii) this sphere being plaited onto the paper page.

```
color f, viewcentr
```

The point of view is `f`. The plane or sphere of projection contains the center of view `viewcentr`. The axis parallel to `zz` that contains the `viewcentr` is projected on a vertical line.

```
numeric Spread
pair ShiftV, OriginProjPagePos
numeric PageWidth
numeric PageHeight
```

These variables control the placement of the projection on the paper. `Spread` is the magnification and `ShiftV` is the position of the `viewcentr` projection on the paper. But, if at some point in your program you introduce `produce_auto_scale` then the `currentpicture` will be centered at `OriginProjPagePos` and scaled to fit inside a rectangle of `PageWidth` by `PageHeight`.

```
color V[]
color L[]p[]
color F[]p[]
```

Vertexes, lines and faces. The idea here is to draw polygons and/or arbitrary lines in 3D space. Defining the polygons and the lines can be a bit tedious as FEATPOST is not interactive. First, one defines a list of the vertexes (`V[]`) that define the polygons and/or the lines. There is a list of polygons and a list of lines. Each polygon (`F[]p[]`) or line (`L[]p[]`) is itself a list of vertexes. All vertexes of the same polygon should belong to the same plane.

```
numeric NL
numeric npl[]
numeric NF
numeric npf[]
```

Number of lines, number of vertexes of each line, number of faces, number of vertexes of each face.

numeric PrintStep

Printstep is the size of iterative jumps along lines. Used by `lineraytrace`, `faceraytrace` and `pathofstraightline`. Big Printsteps make fast raytracings.

boolean FCD[]
colour TableC[]
numeric TableColors
numeric FC[]
colour HigColor
colour SubColor
color LightSource

FCD means "face color defined". The `draw_invisible` macro draws polygons in colour, if it is defined. The colour must be selected from the table of colours `TableC` that has as many as `TableColors`. The colour FC of each polygon will depend on its position relatively to `LightSource` where we suppose there is a lamp that emits light coloured `HigColor`. Furthermore the colour of each polygon may be modified if it belongs to a functional or parametric surface. In this case, if we are looking at the polygon from below than `SubColor` is subtracted from its colour.

numeric RopeColorSeq[]
numeric RopeColors

These are used by `ropepattern`.

numeric TDAtipLen
numeric TDAhalfTipBase
numeric TDAhalfThick

These control the shape of Three-Dimensional Arrows.

boolean ShadowOn
numeric Horizon

When `ShadowOn` is set `true`, some objects can cast a black shadow on a horizontal plane of Z coordinate equal to `Horizon` (an area from this plane may be drawn with `setthestage`) as if there is a punctual source of light at `LightSource`. The macros that can produce shadows, in addition to their specific production, are

- `emptyline`
- `rigorousdisc`
- `verygoodcone`
- `tropicalglobe`
- `whatisthis`
- `kindofcube`
- `draw_all_test`
- `fill_faces`
- `positivecharge`

All macros that contain `shadow` in their name calculate the location of shadows (using `cb`).

path VGAborder

This path and the macro `produce_vga_border` are meant to help you clip the `currentpicture` to a 4:3 rectangle as in a movie frame.

pair PhotoPair[]
color PhotoPoint[]
numeric PhotoMarks

These are used by `photoreverse`.

```

pen ForePen, BackPen
path CLPath
numeric NCL

```

These are used by `closedline`.

3.2 Definitions

- global `makeline@#(text1)`
- global `makeface@#(text1)`

Both of these functions ease the task of defining lines and polygons. Just provide a list of vertices in the right sequence for each polygon and/or line. Suppose a tetrahedron

```

V3:=(+1,-1,-1);V2:=(-1,+1,-1);
V4:=(+1,+1,+1);V1:=(-1,-1,+1);
makeface2(1,2,3);makeface3(1,2,4);
makeface1(3,4,1);makeface4(3,4,2);

```

The number in the last `makeface` or last `makeline` procedure name must be the number of polygons or lines. All polygons and lines from 1 upto this number must be defined but the sorting may be any of your liking.

3.3 Macros

3.3.1 Very Basic Macros

- numeric `X()` Returns the first coordinate of a point or vector (of color type). Replaces `redpart`.
- numeric `Y()` Returns the second coordinate of a point or vector. Replaces `greenpart`.
- numeric `Z()` Returns the second coordinate of a point or vector. Replaces `bluepart`.
- draw `produce_auto_scale` The currentpicture is centered in, and adjusted to the size of, an A4 paper page. This avoids the control of `Spread` and `ShiftV`.
- string `cstr()` Converts a color into its string. Usefull in combination with `getready`.
- string `bstr()` Converts a boolean expression into its string. Usefull in combination with `getready`.

3.3.2 Vector Calculus

- color `N()` Unit vector. Returns `black` (the null vector) when the argument has null norm. The "N" means "normalized".
- numeric `cdotprod()` Dot product of two vectors.
- color `ccrossprod()` Cross product of two vectors.
- numeric `ndotprod()` Cossine of the angle beetween two vectors.
- color `ncrossprod()` Normalized cross product of twovectors.
- numeric `conorm()` Euclidean norm of a vector.
- numeric `getangle()` Angle between two vectors.
- pair `getanglepair()` Orientation angles of a vector. The first angle (`xpart`) is measured between the vector projection on the `XY` plane and the `X` axis. The second angle (`ypart`) is measured between the vector and its projection on the `XY` plane. This may be usefull to find the arguments of `kindofcube`
- color `eulerrotation()` Three-dimensional rotation of a vector. See the figure explaining `kindofcube` to visualize the following movement: (i) grab the `X` component of the vector; (ii) rotate it on the `XY` plane as much as the first argument; (iii) raise it up as much as the second argument; and (iv) turn it around as much as the third argument.
 1. numeric Angle of rotation around the `Z` component.

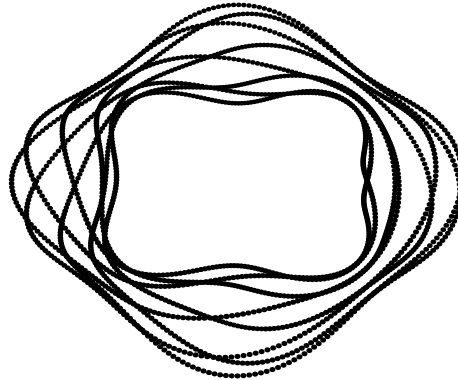


Figure 10: Figure that uses `signalvertex`.

- 2. `numeric` Angle of rotation around the rotated `Y` component.
- 3. `numeric` Angle of rotation around the two times rotated `X` component.
- 4. `color` Vector to be rotated.
- `color` **randomfear** Generates a randomly oriented unit vector.

3.3.3 Projection Macros

- pair **rp()** Converts spatial positions into planar positions on the paper page. The conversion considers the values of the following global variables: `viewcentr`, `ParallelProj`, `SphericalDistortion`, `Spread` and `ShiftV`. When both `ParallelProj` and `SphericalDistortion` are `false` it won't work if either (i) the vectors `f-viewcentr` and `f-R` are perpendicular (`R` is the argument) or (ii) `f` and `viewcentr` share the same `X` and `Y` coordinates.
 1. `color` Spatial position.
- `color` **cb()** Calculates the position of the shadow of a point. Uses `Horizon` and `LightSource`.
 1. `color` Point position.
- `color` **projectpoint()** Calculates the intersection between a plane and a straight line. The plane contains a given point and is perpendicular to the line connecting the `LightSource` and this same point. The line is defined by another given point and the `LightSource`. Summary: **projectpoint** returns the projection of the second argument on a plane that contains the first argument. Can be used to draw shadows cast on generic planes.
 1. `color` Origin of the projection plane.
 2. `color` Point to be projected.
- `color` **lineintersectplan()** Calculates the intersection between a generic plane and a straight line. The plane contains a given point and is perpendicular to a given vector. The line contains a given point and is parallel to a given vector.
 1. `color` Point of the line.
 2. `color` Vector parallel to the line.
 3. `color` Point of the projection plane.
 4. `color` Vector perpendicular to the projection plane.

3.3.4 Plain Basic Macros

- draw **signalvertex()** Draws a dot sized inversely proportional to its distance from the viewpoint `f`.
 1. `color` Location.
 2. `numeric` Factor of proportionality ("size of the dot").
 3. `colour` Colour of the dot.

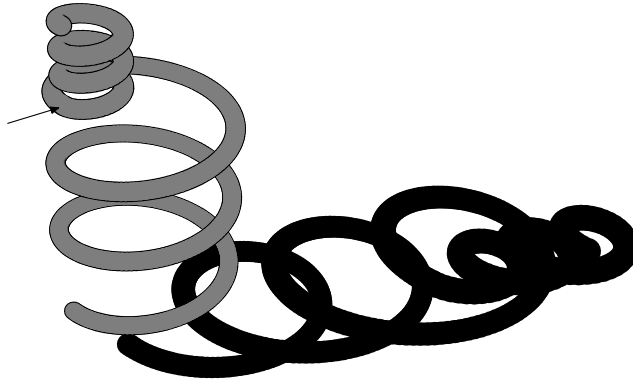


Figure 11: Figure that uses `emptyline`. The junction point of two different lines is indicated by an arrow. Note the unperfection on the top right, inside the upper turn.

- path `pathofstraightline()` When using `SphericalDistortion:=true`, straight lines look like curves. This macro returns the curved path of a straight line between two points. This path will have a greater `length` ("time") when `PrintStep` is made smaller.
- draw `drawsegment()` Alternative `pathofstraightline` that avoids the calculation of all the intermediate points when `SphericalDistortion:=false`.
- drawlabel `cartaxes()` Cartesian axis with prescribed lengths and appropriate labels.
 1. `numeric` Length of the X axis.
 2. `numeric` Length of the Y axis.
 3. `numeric` Length of the Z axis.
- draw `emptyline()` This procedure produces a sort of a tube that can cross over itself. It facilitates the drawing of, for instance, thick helical curves but it won't look right if the curves are drawn getting apart from the point of view. Please, accept this inconvenience. As like many other FEATPOST macros this one can produce visually correct diagrams only in limited conditions. Can cast a shadow.
 1. `boolean` Choose `true` to join this line with a previously drawn line.
 2. `numeric` Factor of proportionality ("diameter of the tube"). The tubes are just sequences of dots drawn by `signalvertex`.
 3. `colour` Colour of the tube border.
 4. `colour` Colour of the tube.
 5. `numeric` Total number of dots on the tube line.
 6. `numeric` Fraction of the tube diameter that is drawn with the tube colour.
 7. `numeric` This is the number of dots that are redrawn with the colour of the tube for each drawn dot with the color of the tube border. Usually 1 or 2 are enough.
 8. `text` This is the name a function that returns a 3D point of the line for each value of a parameter in between 0 and 1.
- draw `closedline()` This procedure produces a tube that can cross over itself. It facilitates the drawing of, for instance, thick helical curves but it won't look right as its thickness does not change with the distance from the point of view. The drawing is entirely done in two dimensions, so the tube diameter depends on the global variables `ForePen` and `BackPen`. There can be more than one line in a figure but all get the same diameter. When calling `closedline()` in different figures of the same program you must reinitialize both `NCL` and `Nobjects` (because `closedline()` uses `getready()`).
 1. `boolean` Value of "the line is closed".
 2. `numeric` Total number of path segments on the tube line.
 3. `numeric` Use 0.5 or more.
 4. `numeric` Use 0.75 or more.

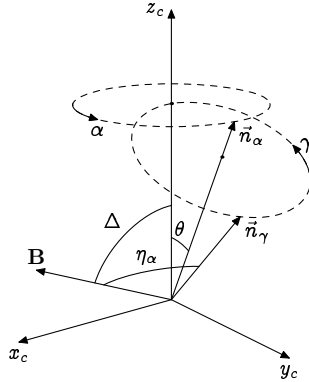


Figure 12: Figure that uses `angline` and `rigorouscircle`.

5. `text` This is the name of a function that returns a 3D point of the line for each value of a parameter in between 0 and 1.
- `drawlabel angline()` Draws an arch between two straight lines with a common point and places a label near the middle of the arch (marks an angle). Note that the arch is not circular.
 1. `color` Point of one line.
 2. `color` Point of the other line.
 3. `color` Common point.
 4. `numeric` Distance between the arch and the common point.
 5. `picture` Label.
 6. `suffix` Position of the label relatively to the middle of the arch. May be one of `lft`, `rt`, `top`, `bot`, `ulft`, `urt`, `llft` and `lrt`.
- `drawlabel angline()` The same as the previous function but the sixth argument is numeric: 0=`rt`; 1=`urt`; 2=`top`; 3=`ulft`; 4=`lft`; 5=`llft`; 6=`bot`; 7=`lrt`; any other number places the label on the middle of the arch.
- `draw squareangline()` This is supposed to mark 90 degree angles but works for any angle value.
 1. `color` Point of one line.
 2. `color` Point of the other line.
 3. `color` Common point.
 4. `numeric` Distance between the "arch" and the common point.
- `path rigorouscircle()` 3D circle. The total "time" of this path is 8. This small number makes it easy to select parts of the path. The circle is drawn using the "left-hand-rule". If you put your left-hand thumb parallel the circle axis then the other left-hand fingers curl in the same sense as the circle path. This path always starts, approaching the view point, from a point on a diameter of the circle that projects orthogonally to its axis, and rotating around the axis in the way of the left-hand-rule.
 1. `color` Center of the circle.
 2. `color` Direction orthogonal to the circle (circle axis).
 3. `numeric` Radius of the circle.
- `draw tddarrow()` Draws a flat arrow that begins at the first argument and ends at the second. The shape of the arrow is controlled by the global variables `TDAtiplen`, `TDAhalftipbase`, `TDAhalfthick`.
- `path twocyclestogether()` This macro allows you to draw any solid that has no vertexes and that has two, exactly two, planar cyclic edges. In fact, it doesn't need to be a solid. Just provide the pathes of both cyclic edges as arguments but note that the returned path is polygonal. In order to complete the drawing of this solid you have to choose

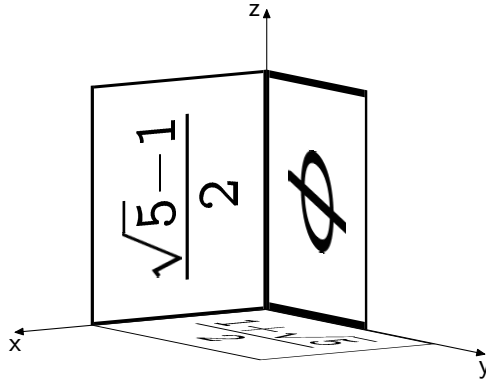


Figure 13: Example that uses `labelinspace`.

one of the edges to be drawn immediately afterwards. This is done automatically by the `whatisthis` macro for the case of two parallel and concentric ellipses.

- path `ellipticpath()` Produces an elliptic path in 3D space.
 1. `color` Position of the center.
 2. `color` Major or minor axis.
 3. `color` The other axis.
- drawlabel `labelinspace()` Draw some 2D picture on some 3D plane (only when `ParallelProj:=true`).
 1. `color` Position for the lower-left corner.
 2. `color` Orientation of the picture's bottom edge.
 3. `color` Orientation of the picture's left edge.
 4. `text` 2D picture's name.

3.3.5 Standard Objects

- path `goodcirclepath()` Another 3D circle macro. More rigorous than `rigorouscircle` but when the direction orthogonal to the circle is almost orthogonal to the line `viewpoint--center` it doesn't work correctly. The total "time" of this path is 36.
 1. `color` Center of the circle.
 2. `color` Direction orthogonal to the circle.
 3. `numeric` Radius of the circle.
- draw `spatialhalfsfear()` An hemisphere. Doesn't work with `f` inside it.
 1. `color` Center.
 2. `color` Vector orthogonal to the frontier circle and pointing out of the concavity.
 3. `numeric` Radius of the (hemi)sphere.
- path `spatialhalfcircle()` And yet another 3D circle macro. Only the visible or the hidden part. This is useful to mark sections of cylinders or spherical major circles.
 1. `color` Center of the circle.
 2. `color` Direction orthogonal to the circle.
 3. `numeric` Radius of the circle.
 4. `boolean` The visible part is selected with `true` and the hidden with `false`.
- draw `rigorousdisc()` 3D opaque cylinder with/without a hole. Can cast a shadow (without the hole).
 1. `numeric` Ray of an axial hole.
 2. `boolean` Option for completely opaque cylinder (`true`) or partial pipe (`false`) when there is no hole. When the cylinder has an hole this option should be `true`.

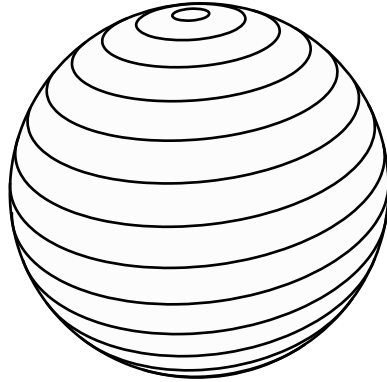


Figure 14: Figure that uses `tropicalglobe`.

- 3. `color` Center of one circular base.
- 4. `numeric` Radius of both circular bases.
- 5. `color` Vector that defines the length and orientation of the cylinder. The addition the third and fifth arguments should give the position of the center of the other circular base.
- draw `verygoodcone()` 3D cone. Can cast a shadow.
 1. `boolean` Option to draw dashed evenly the invisible edge (`true`) or not (`false`).
 2. `color` Center of the circular base.
 3. `color` Direction ortogonal to the circular base.
 4. `numeric` Radius of the circular base.
 5. `color` Position of the vertex
- path `rigorousfearpath()` 3D sphere. Simple but hard.
 1. `color` Center position.
 2. `numeric` Radius.
- draw `tropicalglobe()` Globe with minor circles. Can cast a shadow.
 1. `numeric` Number of marked latitudes.
 2. `color` Center position.
 3. `numeric` Radius
 4. `color` Axis orientation.
- draw `whatisthis()` An elliptic frustum. Both edges are elliptic an have the same orientation but one may be greater than the other. Can cast a shadow.
 1. `color` Reference edge center.
 2. `color` Major or minor axis.
 3. `color` The other axis.
 4. `numeric` Length of the original cylinder.
 5. `numeric` Edges axis length ratio.
- draw `kindofcube()` Polyhedron with six orthogonal faces (cuboid).
 1. `boolean` Also draw the invisible edges `dashed evenly` (`true`) or do not.
 2. `boolean` The reference point may be a vertex (`true`) or the center(`false`).
 3. `color` Reference point.
 4. `numeric` Alpha1.
 5. `numeric` Alpha2.
 6. `numeric` Alpha3.
 7. `numeric` L1. Length of the first side.

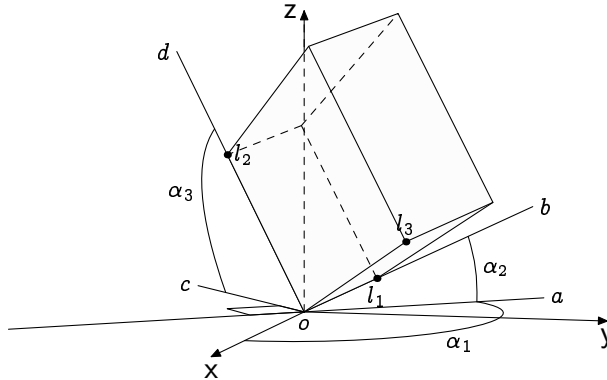


Figure 15: Figure that uses and explains `kindofcube`. Note that the three indicated angles may be used as arguments of `eulerrotation`.

8. `numeric` L2. Length of the second side.
9. `numeric` L3. Length of the third side.

These arguments are represented in the next figure.

- draw `setthestage()` Produces an horizontal square made of squares. Its Z coordinate is defined by `Horizon`.
 1. `numeric` Number of squares in each side.
 2. `numeric` Size of each side.
- draw `setthearena()` Produces an horizontal circle made of circles. Its Z coordinate is defined by `Horizon`. Due to the fact that the center of a circle is not on the center of its central perspective projection, this may look a bit strange.
 1. `numeric` Number of circles on a diameter.
 2. `numeric` Diameter.
- draw `smoothtorus()` Toxic donut (not to be eaten). Produces an error message when `f` is close to the table.
 1. `color` Center.
 2. `color` Direction orthogonal to the torus plane.
 3. `numeric` Big ray.
 4. `numeric` Small ray.

3.3.6 Composed Objects

- draw `positivecharge()` Draws a sphere with a plus or minus sign on the surface. The horizontal segment of the sign is drawn on the horizontal plane that contains the sphere center. The middle point of this segment is on a vertical plane containing the viewpoint.
 1. `boolean` Selects the sign (`true` means positive).
 2. `color` Position of the center.
 3. `numeric` Sphere ray.
- draw `simplecar()` Draws a cuboid and four discs in a configuration resembling an automobile. The first three arguments of `simplecar` are the same as the the last seven arguments of `kindofcube` but grouped in colors.
 1. `color` Center of the cuboid that constitutes the body of the car..
 2. `color` Angles defining the orientation of the car (see `kindofcube`).
 3. `color` Dimensions of the car.
 4. `color` Characteristics of the front wheels. `redpart`-distance from the front. `greenpart`-width of the front wheels (length of the cylinders). `bluepart`-wheel ray.
 5. `color` Same as above for the rear wheels

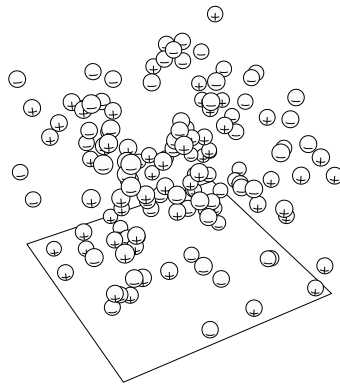


Figure 16: Figure that uses `positivecharge`, `getready` and `doitnow`.

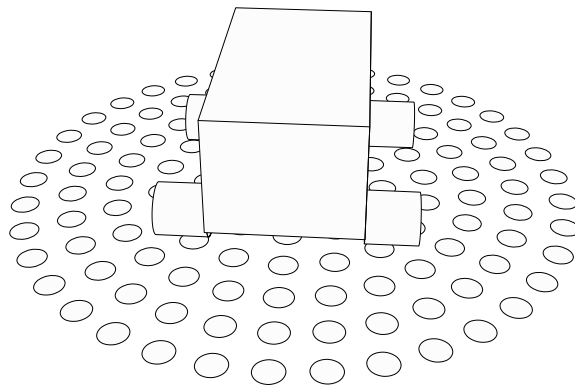


Figure 17: Figure that uses `setthearena` and `simplecar`.

3.3.7 Shadow Pathes

- draw **signalshadowvertex()** Draws the shadow of a **signalvertex** dot.
 1. **color** Location of the light-blocking dot.
 2. **numeric** Factor of proportionality ("size of the dot").
 3. **colour** Colour of the dot.
- path **ellipticshadowpath()** Produces the shadow of an elliptic path.
 1. **color** Position of the center.
 2. **color** Major or minor axis.
 3. **color** The other axis.
- path **circleshadowpath()** Produces the shadow of a circle.
 1. **color** Center of the circle.
 2. **color** Direction ortogonal to the circle.
 3. **numeric** Radius of the circle.
- path **rigorousfearshadowpath()** 3D sphere shadow.
 1. **color** Center position.
 2. **numeric** Radius.

3.3.8 Differential Equations

Before we proceed, be aware that solving differential equations (DE) is mainly an experimental activity. The most probable result of a procedure that attempts to solve a DE is garbage. The procedure may be unstable, the solution may be littered with singularities or something may go wrong. If you don't have a basic understanding of differential equations then skip this section, please.

- path **fieldlinepath()** A vectorial field line is everywhere tangent to the field vectors. Two different parallel fields have the same field lines. So the field only constrains the direction of the field lines, not any kind of "speed" and, therefore, it is recommended to normalize the field before using this macro that contains a second-order Runge-Kutta method implementation.
 1. **numeric** Total number of steps.
 2. **color** Initial position.
 3. **numeric** Step (arc)length.
 4. **text** Name of the function that returns a field vector for each 3D position.
- path **trajectorypath()** The acceleration of a particle in a conservative force field is equal to the ratio (conservative force)/(particle mass). The acceleration is also equal to the second order time derivative of the particle position. This produces a second order differential equation that we solve using a second-order Runge-Kutta method implementation.
 1. **numeric** Total number of steps.
 2. **color** Initial position.
 3. **color** Initial velocity.
 4. **numeric** Time step.
 5. **text** Name of the function that returns a (force/mass) vector for each 3D position.
- path **magnetictrajectorypath()** The acceleration of a charged particle in a magnetic field is equal to the ratio (magnetic force)/(particle mass) but the magnetic force depends on both the velocity and the magnetic field. The acceleration is also equal to the second order time derivative of the particle position. This produces a second order differential equation that we solve using a fourth-order Runge-Kutta method implementation.
 1. **numeric** Total number of steps.
 2. **color** Initial position.
 3. **color** Initial velocity.
 4. **numeric** Time step.
 5. **text** Name of the function that returns a (charge)*(magnetic field)/(particle mass) vector for each 3D position.

3.3.9 Renderers

- draw **sharpraytrace** Heavy procedure that draws only the visible part of all edges of all defined faces. There's no point in using this procedure when there are no intersections between faces. Any how this will not work for non-convex faces nor when `SphericalDistortion:=true`.
- draw **lineraytrace()** Draws only the visible part of all defined lines using sequences of dots (`signalvertex` and `PrintStep`).
 1. `numeric` Dot size.
 2. `colour` Dot colour.
- draw **faceraytrace()** Draws only the visible part of all edges of all defined faces using sequences of dots (`signalvertex` and `PrintStep`).
 1. `numeric` Dot size.
 2. `colour` Dot colour.
- draw **draw_all_test()** Draws all defined edges (and lines) in a correct way independently of the kind of projection used. Can cast a shadow (but the shadow is not correct when `SphericalDistortion:=true`).
 1. `boolean` If `true` the lines are also drawn.
- draw **fill_faces()** Unfills and draws all faces in the order they were defined (without sorting). Can cast a shadow.
 1. `text` Like the argument of `drawoptions` but used only inside this macro and only for the edges.
- draw **draw_invisible()** This is a fast way of removing hidden lines that doesn't allow for intersecting polygons nor polygons of very different area. It works by +sorting all polygons by distance to `f` and then by "filling" the polygons. This routine may be used to draw graphs of 3D surfaces.
 1. `boolean` If `true` polygons are sorted relatively to nearest vertex and, if `false`, relatively to their mass center. Choose `false` for surface plots.
 2. `boolean` If `false` then the polygons are painted with their `FC` colour modified by `LightSource`. If `true` then the next two arguments are used and the polygons are darkened proportionally to their distance from `f`.
 3. `colour` Colour of faces.
 4. `colour` Colour of the edges.
- global **getready()** When you don't want to edit the source of the MetaPost program, to resort the objects so they'll be drawn correctly, use this macro and the next.
 1. `string` Command line that would draw some object. For instance: `"rigorousfearpath(black,1);"`.
 2. `color` Reference position of that object.
- draw **doitnow** The reference positions given as arguments of previous `getready` calls are used to sort and draw the objects also given as string arguments to previous `getready` calls. Remember to initialize `Nobjects:=0;` before a second figure.

3.3.10 Nematics (Direction Fields)

Nematics are the least ordered liquid crystals. Their configurations can be described by direction fields (vector fields without arrows). The two following routines ease the task of representing their configurations.

- global **generatedirline()** Defines a single straight line segment in a given position and with a given orientation.
 1. `numeric` Line index number.
 2. `numeric` Angle between the `X` axis and the projection of the line on the `XY` plane.
 3. `numeric` Angle between the line and the `XY` plane.
 4. `numeric` Line (arc)length.

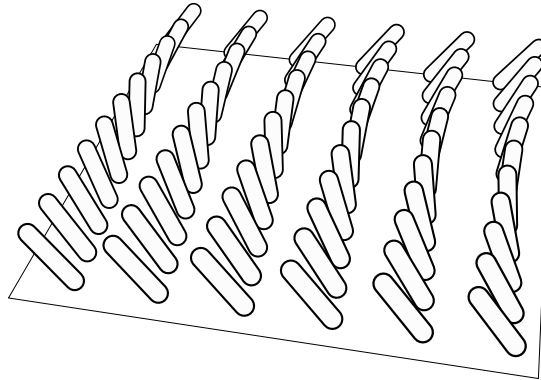


Figure 18: Figure that uses `director_invisible` and `generatedirline`.

5. `color` Position of the line middle point.

- draw **`director_invisible()`** This is a direction field renderer that can sort direction lines. This routine draws straight lines of given "thickness" between the first all the points of all the `L[]p[]` lines. It is supposed to help you draw vector fields without arrows but taking care of invisibility. The lines may be generated by `generatedirline` or by other macros.
 1. `boolean` When there is no need to sort lines you may use `false` here.
 2. `numeric` "Thickness" of the direction lines
 3. `boolean` Use `true` for cyclic "direction" lines.

3.3.11 Surface Plots

Many powerfull plotting packages like `gnuplot` <http://www.gnuplot.info/> and `gri` <http://gri.sourceforge.net/> are freely available. Because of this, FEATPOST surface plots are geared towards unusual features like equilateral triangular grid, hexagonal domain and merging together functional and parametric surface descriptions.

- draw **`hexagonaltrimesh()`** Plots a functional surface on a triangular or hexagonal domain. Uses the `LightSource`.
 1. `boolean` Select the kind of domain. `true` for hexagonal and `false` for triangular. The domain is centered on the origin (`black`). When the domain is hexagonal two of its corners are on the `-YY` axis. When the domain is triangular one of its corners is on the `X` axis.
 2. `numeric` Number of small triangles on each side of the triangular domain or three times the number of small triangles on each side of the hexagonal domain.
 3. `numeric` Length of the triangular domain side or three times the hexagonal domain side.
 4. `text` Name of the function that returns the `Z` coordinate of a surface point of coordinates `X` and `Y`.
- global **`partrimesh()`** Defines a parametric surface that can be drawn with `draw_invisible`. In the following descriptions `S` and `T` are the parameters. Remember to initialize `NF`. The surface is defined so that quadrangles are used whenever possible. If impossible, two triangles are used but their orientation is selected to maximize the surface smoothness. Also note that, unlike `hexagonaltrimesh()`, the spatial range you require to be visible is always first reshaped into a cube and second compressed or extended vertically. How much the cube is compressed or extended depends on the last `numeric` argument, the compression factor for `Z`, meaning that the final height of the cube is $2/(\text{compression factor})$. Thanks to Sebastian Sturm for pointing the need to explain this.
 1. `numeric` Number of `T` steps.

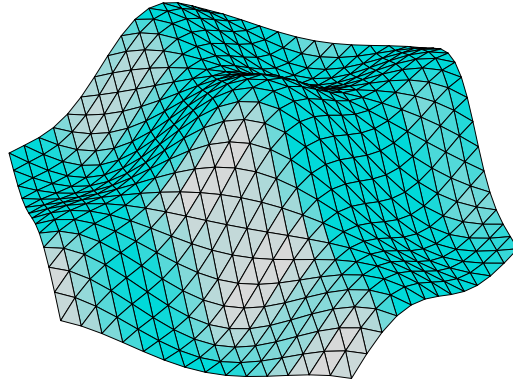


Figure 19: Figure that uses `hexagonaltrimesh`.

2. `numeric` Number of `S` steps.
3. `numeric` Minimal `T` value.
4. `numeric` Maximal `T` value.
5. `numeric` Minimal `S` value.
6. `numeric` Maximal `S` value.
7. `numeric` Minimal `X` value.
8. `numeric` Maximal `X` value.
9. `numeric` Minimal `Y` value.
10. `numeric` Maximal `Y` value.
11. `numeric` Minimal `Z` value.
12. `numeric` Maximal `Z` value.
13. `numeric` Compression factor for `Z` values.
14. `text` Name of the function that returns a surface point (of `color` type) for each pair `(S,T)`.