Surface Evolver Manual

Version 2.14
August 18, 1999

Kenneth A. Brakke

Mathematics Department
Susquehanna University
Selinsgrove, PA 17870

brakke@susqu.edu

Table of Contents

Chapter 1. Introduction.

§1.1. General description

The Surface Evolver is an interactive program for the study of surfaces shaped by surface tension and other energies. A surface is implemented as a simplicial complex, that is, a union of triangles. The user defines an initial surface in a datafile. The Evolver evolves the surface toward minimal energy by a gradient descent method. The evolution is meant to be a computer model of the process of evolution by mean curvature, which was studied in [B1] for surface tension energy in the context of varifolds and geometric measure theory. The energy in the Evolver can be a combination of surface tension, gravitational energy, squared mean curvature, user-defined surface integrals, or knot energies. The Evolver can handle arbitrary topology (as seen in real soap bubble clusters), volume constraints, boundary constraints, boundary contact angles, prescribed mean curvature, crystalline integrands, gravity, and constraints expressed as surface integrals. The surface can be in an ambient space of arbitrary dimension, which can have a Riemannian metric, and the ambient space can be a quotient space under a group action. The user can interactively modify the surface to change its properties or to keep the evolution well-behaved. The Evolver was written for one and two dimensional surfaces, but it can do higher dimensional surfaces with some restrictions on the features available. Graphical output is available as screen graphics and in several file formats, including PostScript.

The Surface Evolver program is freely available (see chapter 2) and is in use by a number of researchers. Some of the applications of the Evolver so far include modelling the shape of fuel in rocket tanks in low gravity [Te], calculating areas for the Opaque Cube Problem [B4],[B6], computing capillary surfaces in cubes [MH] and in exotic containers [C], simulating grain growth [FT][WM], studying grain boundaries pinned by inclusions, finding partitions of space more efficient than Kelvin's tetrakaidecahedra [WP][KS1], calculating the rheology of foams [KR1][KR2], modelling the shape of molten solder on microcircuits [RSB], studying polymer chain packing, modelling cell membranes [MB], knot energies [KS2], sphere eversion [FS], and classifying minimal surface singularities.

The strength of the Surface Evolver program is in the breadth of problems it handles, rather than optimal treatment of some specific problem. It is under continuing development, and users are invited to suggest new features.

This manual contains operational details and mathematical descriptions (please excuse the inconsistent fonts and formatting; the manual grows by bits and pieces). Much of the manual (the stuff without a lot of mathematical formulas) is also in HTML format, included in the distribution. A journal article description of the Evolver appeared in [B2].

Previous users of the Evolver should consult the History chapter for the new features.

§1.2. Portability

The Evolver is written in portable C and has been run on several systems: Silicon Graphics, Sun, HP, DEC, MS-DOS, Windows NT, Windows 95/98, and Macintosh. It is meant to be easily portable to any system that has C.

§1.3. Bug reports

Bug reports should be submitted by email to `brakke@susqu.edu`. Please include the Evolver version number, a description of the problem, the initial datafile, and the sequence of commands necessary to reproduce the problem.

§1.4. Web home page

My Web home page is `http://www.susqu.edu/facstaff/b/brakke/`. Evolver-related material and links will be posted there. In particular, there are many examples of surfaces. You may also want to check out The Geometry Center's home page `http://www.geom.umn.edu/`.

§1.5. Newsletter

The group of Surface Evolver users has grown large enough that I have started a newsletter. Mostly it consists of announcements of new Evolver versions. If you would like to be on the mailing list, send your email address to `brakke@susqu.edu`. Back issues are included in the HTML part of the distribution.

§1.6. Acknowledgements

The Evolver was originally written as part of the Minimal Surfaces Group of the Geometry Supercomputing Project (now The Geometry Center), sponsored by the National Science Foundation, the Department of Energy, Minnesota Technology, Inc., and the University of Minnesota. The program is available free of charge.

Chapter 2. Installation.

This chapter explains how to get and install the Evolver. Evolver is written to be portable between systems. The only requirement is a C compiler. Graphics on different systems may need customizing; see the section on graphics porting below.

§2.1. Acquisition.

The Surface Evolver program is available free of charge.

The Surface Evolver distribution package is available on the Web at

`http://www.susqu.edu/facstaff/b/brakke/evolver`

or at

`http://www.geom.umn.edu/software/download/evolver/evolver.html`

If you don't have a Web browser, and if you can connect to Internet and can use ftp, do a ftp login to `geom.umn.edu` (128.101.25.35) as "anonymous" with your email address as password. From the `pub/software/evolver` directory, get `evolver.tar.Z`. This archive contains source code, datafiles, and this manual, and unpacks to about 5.3MB. See below for special versions for Macs and PCs.

The program is distributed in a compressed tar file `evolver.tar.Z` (for unix systems; see below for others). Get this file into a working directory. Uncompress it with

```
uncompress evolver.tar
```

(yes, leave off the `.Z`). Extract the files with

```
tar xvf evolver.tar
```

This will unpack into three subdirectories: src (source code), doc (the manual), and fe (sample datafiles). The packed archive is about 1MB, unpacks to about 3MB. You will probably need another 3 or 4 MB to compile.

If you cannot use ftp, you can get Evolver on a floppy disk. Send email to `brakke@susqu.edu` or write to

Kenneth A. Brakke
Mathematics Department
Susquehanna University
Selinsgrove, PA 17870
USA

Indicate what format you can handle: tar, gzipped, zip, compressed, MS-DOS, etc.

Manual. This complete manual is in TeX DVI format in `manual.dvi`. Print it out however you print DVI files. There is also a PostScript version available from `http://www.susqu.edu/facstaff/b/brakke/evolver/manual.ps`. If you can't print it, you can ask for a hardcopy from `brakke@susqu.edu`. The figures to accompany the tutorial examples in the manual are in PostScript files `*.ps`, which can be sent direct to any PostScript printer. There is also an HTML version of most of the manual (except the parts with mathematical formulas) in `http://www.susqu.edu/facstaff/b/brakke/evolver/manual-html.tar.Z`. Get this even if you don't have a Web browser, since Evolver's on-line help uses extracts from these files. When using a browser directly, start with `default.htm`.

Macintosh. For those with Macintoshes, there is a Mac PowerPC version available as `http://www.susqu.edu/facstaff/b/brakke/evolver/EvolverPPC.sea.Hqx`. These are Stuffit Lite self-extracting archives treated with BinHEx. They include a README file with Mac specific information, datafiles in Mac format, and the HTML version of the documentation. You should also download the Postscript file for this manual, `manual.ps`. The Mac version may not be the current version, depending on the access I have to a Mac.

Windows NT and Windows 95/98. The file `http://www.susqu.edu/facstaff/b/brakke/evolver/evolver-NT.zip` has the executable file. Just unzip it and place the executable somewhere in your path. You must get the documentation (`manual.ps, ev-html.zip`), sample datafiles (`fe.zip`), and sample command files (`cmd.zip`) separately. This is an OpenGL version that has screen graphics much like geomview. Early versions of Windows 95 do not support OpenGL, but you can add that support. See `http://www.susqu.edu/facstaff/b/brakke/evolver/`. There is also a non-OpenGL version available, and even a Windows 3.1 version.

DOS. For those with PC compatibles without Windows, `http://www.susqu.edu/facstaff/b/brakke/evolver/` has a 32-bit DOS version. This PKZIP archive contains evolver.exe and the DOS extender program needed to run it, and a few datafiles. It was made with the Free Software Foundation's DJGPP compiler. For further instructions, see evolver.doc in evol-dos.zip. I hope no one out there still has to use this version.

Graphics. Screen graphics generally uses an external graphics program. There is an X-Windows graphics interface. The best graphics, however, are done on Silicon Graphics systems with the `geomview` graphics program. `geomview` is available by anonymous ftp from `geom.umn.edu`. There is also a version of `geomview` for X-Windows. Beware, though, that X-geomview has to emulate all 3D graphics in software, as X-windows is a 2D system. Hence X-geomview is slower, and does not use any built-in 3D graphics hardware a machine may have. X-geomview is available by anonymous ftp from geom.umn.edu in `/pub/software/geomview`. There are precompiled binaries for several systems, and source for other systems. One caution: `geomview` does not deal well with object sizes below 1e-5, so displaying micron-size objects using MKS units is ill-advised. The Windows NT/95/98 version graphics are pretty good, especially if you have OpenGL hardware support on your graphics card. Hit the 'h' key with the mouse focus on the graphics window to get a summary of the commands.

For those with surfaces already defined for `geomview`, files in `geomview`'s OFF format may be converted to the Evolver's datafile format by the program `polymerge`, which is included in the `geomview` distribution. Use the `-b` option. Files in other `geomview` formats may be converted to OFF with `anytooff`, also included with `geomview`.

NOTE: The X windows versions of `geomview` seem to require some event to occur in the `geomview` display window before it will redraw after loading a new datafile. So if `geomview` seems hung, just run the mouse over the window to give it an event to wake it up.

§2.2. Environment.

Evolver needs to find the initial datafile and sometimes other files (e.g. command files for the "read" command, or the HTML version of the manual). If the file is not in the current directory, then an environment variable called `EVOLVERPATH` will be consulted for a directory search list. The datafile directory and the documentation directory with the HTML files should definitely be included. The format is the same as the usual `PATH` variable. Set it up as usual in your system:

Unix C shell:

```
setenv EVOLVERPATH /usr/you/evolver/fe:/usr/you/evolver/doc
```

Bourne shell:

```
EVOLVERPATH=/usr/you/evolver/fe:/usr/you/evolver/doc
export EVOLVERPATH
```

MS-DOS:

```
set EVOLVERPATH=c:\evolver\fe;c:\evolver\doc
```

For the MS-DOS version, see the file `evolver.doc` included in `evolver.zip` for some optional environment variables.

Windows 95/98, Windows NT: Use the Control Panel System window to set the EVOLVERPATH environment variable.

Macintosh: Doesn't seem to have environment variables. Pending finding out how these things are supposed to be done on Macs, I have hardwired in a path consisting of the current directory, the fe subdirectory, and the doc subdirectory.

All systems: Output files are written to the filename given by the user in response to a prompt, so include the path you want.

§2.3. Files.

These are the files in the `evolver.tar.Z` archive.

Documentation in subdirectory `doc`:

`manual.dvi`	TeX DVI version of manual
`*.ps`	Postscript figures for manual
`*.htm`	HTML version of documentation

In subdirectory `src`:

For datafile and command parsing (user should not remake these):

`datafile.lex`	lex source for lexyy.c
`command.yac`	yacc source for ytab.c

For compilation:

`Makefile`
`*.c`
`*.h`

Screen graphics interfaces:

`xgraph.c`	X-Windows graphics interface
`geomgraph.c`	`geomview` interface for Silicon Graphics
`oglgraph.c`	OpenGL interface for Windows NT/95/98
`iriszgraph.c`	Silicon Graphics GL graphics
`irisstereo.c`	does stereo displays on Silicon Graphics so equipped
`hpzgraph.c`	HP 98731 graphics
`gnugraph.c`	graphics for DOS DJGPP version
`nulgraph.c`	no screen graphics

Evolver datafiles in subdirectory `fe`:

`*.fe`	(initial datafiles)
`*.cmd`	(sample command files for "read" command)
`*.wlf`	(Wulff vector files)
`jcmap`	(sample color map file)

You may put the data and colormap files in various directories. Evolver checks the environment variable `EVOLVERPATH` for data file paths.

§2.4. Unix Compilation.

First, you need to modify `Makefile` for your system. `Makefile` begins with sets of macro definitions for various systems. If your system is listed, remove the comment symbols '`#`' from start of your definitions. If your system is not there, use the `GENERIC` defines, or set up your own, and leave the graphics file as `xgraph` if you have X-windows, `nulgraph` otherwise. If you do define your own system, be sure to put a corresponding section in `include.h`. After you get a working program you can write a screen graphics interface if you can't use one of those provided. Edit `CFLAGS` to have the proper options (optimization, floating point option, etc.). `GRAPH` should be the name of a screen graphics interface file (see list under **Files** above, or write your own as discussed below). `GRAPHLIB` should be the appropriate graphics library plus any other libraries needed. Check all paths and change if necessary for your system.

If you want Evolver to be able to use geomview, include `-DOOGL` in CFLAGS.

If you want Evolver to operate in a higher space dimension n than the default maximum of 4, include `-DMAXCOORD=`n in CFLAGS. This sets the upper limit of dimensionality, and is used for allocating space in data structures. You can use `-DMAXCOORD=3` to save a little memory.

If your system supports the long double data type, you can compute in higher precision by compiling with `-DLONGDOUBLE` in CFLAGS.

You can let the compiler optimize better if you hard-wire the space dimension into the code with the compiler option `-DSDIM=`n, where n is the desired dimension of space. But such an Evolver can handle only the given space dimension, no more, no less.

Silicon Graphics systems with multiple processors may compile a version that will use all processors for some calculations by including `-DSGI_MULTI` in CFLAGS. This version will run fine with one processor, also. Currently, the only calculations done in parallel are the "named quantities". The number of processes actually done in parallel can be controlled with the `-p`n command line option.

The file `include.h` lists the include files for various systems. If your system isn't listed, you will have to put in a list of your own. Try copying the generic list (or one of the others) and compiling. Your compiler will no doubt be very friendly and helpful in pointing out unfound header files.

`include.h` also has various other system-specific defines. See the `GENERIC` section of it for comments on these, and include the appropriate ones for your system.

Now try compiling. Hopefully, you will only have to change the system-specific parts of `Makefile` and `include.h` to get things to work. If significant changes to other files are needed, let me know. For example, the return value of `sprintf()` varies among systems, and that caused problems once upon a time.

Test by running on the `cube.fe` sample file as described in the **Tutorial** chapter.

§2.4.1. X-Windows graphics.

To have Evolver use X-Windows graphics for its native graphics, edit `Makefile` to have `xgraph` as the graphics module and have all the proper libraries linked in. The X window will appear when you do the "`s`" command. Evolver does not continually poll for window redraw events, so if you move or resize the window you will have to do "`s`" again to redraw the surface. You may also be interested in the X version of geomview mentioned above.

§2.5. Customizing graphics.

This section is provided for those people who need to write their own graphics interface module. All device-specific graphics output has been collected into a few basic routines. There are two styles of interface. One provides facets in random order for routines that can do their own hidden surface removal. The other provides facets in back to front order (painter algorithm).

§2.5.1. Random-order interface.

The random-order interface has several global pointers to functions which should be set to the user's functions. This permits several sets of graphics routines in one program. The facets are generated by `graphgen()`, which call the user functions. The function `display()` is called as the overall display function; it should set the function pointers and call `graphgen()`. Example: `iriszgraph.c`

The function pointers are:

`void (*graph_start)(void)`

This is called at the start of each display. It should do whatever device initialization and screen clearing is needed.

`void (*graph_facet)(struct graphdata *,facet_id)`

This is called to graph one triangular facet. See `extern.h` for the definition of `graphdata`. The second argument is a facet identifier for routines that are not happy with just the information in `graphdata`; it is for Evolver gurus only. Facets are presented in random order. The coordinates are not transformed; the current transformation matrix is in `view[][]`, which is in homogeneous coordinates.

`void (*graph_edge)(struct graphdata *)`

This function is called to graph one edge in the string model.

`void (*graph_end)(void)`

This function is called after all data has been given.

§2.5.2. Painter interface.

The painter model is similar, except there is an extra layer of functions. Example: `xgraph.c`, `psgraph.c`, `gnugraph.c`. In `display()`, the user should set

```
graph_start = painter_start;
graph_facet = painter_facet;
graph_end   = painter_end;
```

The user should also set these function pointers:

`void (*init_graphics)(void)`

Called by `painter_start()` to do device initialization.

`void (*display_facet)(struct tsort *)`

Called by `painter_end()` to graph sorted facets one at a time. See `extern.h` for `struct tsort`.

`void (*finish_graphics)(void)`

Called at the end of all facets.

The user should set `graph_edge` as in the random interface.

§2.6. Dynamic load libraries.

Many Evolver features, such as level set constraints, parametric boundaries, named method integrands, and Riemannian metrics require user-defined functions of a set of arguments. The expressions for these functions are ordinarily stored as a parse tree and interpreted each time needed, which can be much slower that evaluating compiled expressions. There is a way to use a set of compiled functions specific to a datafile through a mechanism known as dynamic loading. Here a library of functions for a datafile is separately compiled, and then loaded at runtime when a the datafile is loaded. Currently, the Evolver only implements a dynamic loading mechanism found on many unix systems, whose presence can be tested by looking for the existence of the file `/usr/include/dlfcn.h`. If it exists, you can enable dynamic loading by including `-DENABLE_DLL` in the `CFLAGS` line in the `Makefile`. On some systems, you may need to include `-ldl` on the `GRAPHLIB` line also, to link Evolver with functions such as `dlopen()`.

To create the library for a datafile, write a source file containing C code for the desired functions, compile it, and link it into a shared library. The function should be able to compute the value and the partial derivatives of the function, and its second partials if you are going to use any Hessian features. A sample source file for a 2-dimensional datafile:

```
#define FUNC_VALUE  1
#define FUNC_DERIV  2
#define FUNC_SECOND 3
#define MAXCOORD 4   /* must be same as in Evolver!! */
#define REAL double  /* long double if Evolver compiled with -DLONGDOUBLE */
struct dstack { REAL value;
                REAL deriv[2*MAXCOORD];
                REAL second[2*MAXCOORD][2*MAXCOORD]; } ;


void func1 ( mode, x, s )
int mode; /* FUNC_VALUE, FUNC_DERIV, FUNC_SECOND */
REAL *x;  /* pointer to list of arguments */
struct dstack *s;  /* for return values */
{ REAL value;

  s->value = x[0] + x[1]*x[1];

  if ( mode == FUNC_VALUE ) return;

  /* first partials */
  s->deriv[0] = 1.0;
  s->deriv[1] = 2*x[1];

  if ( mode == FUNC_DERIV ) return;

  /* second partials */
  s->second[0][0] = 0.0;
  s->second[0][1] = 0.0;
  s->second[1][0] = 0.0;
  s->second[1][1] = 2.0;

  return;
}
```

Supposing the sourcefile name to be foo.c, compile and link on SGI systems (IRIX 5.0.1 or above) with

```
cc -c foo.c
ld -shared foo.o -o foo.so
```

Sun systems are the same, but with -s in place of -shared. For other systems, consult the ld documentation for the option to make a shared library or dynamic load library.

To use the functions in a datafile, include a line at the top of the datafile before any of the functions are used:

```
load_library "foo.so"
```

Up to 10 libraries may be loaded. Afterwards, any of the functions may be invoked just by using their name, without an explicit argument list because the argument list is always implicit where these functions are legal. Examples, supposing func2 is also defined with one argument:

```
constraint 1
formula:  func1
```

```
boundary 1 parameters 2
x1:  func2
x2:  3*func2 + sin(p1)
```

It is up to you to make sure the number of arguments your function expects is the same as the number implicit in the use of the function. You do not need to explicitly declare your functions in the datafile. Any undefined identifier is checked to see if it is a dynamically loaded function.

NOTE: This implementation of dynamic loading is experimental, and the interface described here may change in the future.

Chapter 3. Tutorial.

This chapter introduces some basic concepts and then guides the user through a series of examples. Not all the sample datafiles that come with Evolver are discussed; users may browse them at their liesure. This chapter can be read before the following chapters. The following chapters assume you have read the Basic Concepts section of this chapter. If you are a little rusty on Advanced Calculus, there is a short tutorial on it at the end of this chapter.

§3.1. Basic Concepts.

The basic geometric elements used to represent a surface are vertices, edges, facets, and bodies. Vertices are points in Euclidean 3-space. Edges are straight line segments joining pairs of vertices. Facets are flat triangles bounded by three edges. A surface is a union of facets. (Actually, there is no separate surface entity in the program; all facets essentially belong to one logical surface.) A body is defined by giving its bounding facets.

The term "surface", when used to refer to the entity upon which the Evolver operates, refers to all the geometric elements plus auxiliary data such as constraints, boundaries, and forces.

There are no limitations on how many edges may share a vertex nor on how many facets may share an edge. Thus arbitrary topologies are possible, including the triple junctions of surfaces characteristic of soap films.

Edges and facets are oriented for bookkeeping purposes, but there are no restrictions on the orientation of neighboring facets. Unoriented surfaces are thus possible.

A surface is deemed to have a total energy, arising from surface tension, gravitational energy, and possibly other sources. It is this energy which the Evolver minimizes.

No particular units of measurement are used. The program only deals with numerical values. If you wish to relate the program values to the real world, then all values should be within one consistent system, such as cgs or MKS.

The initial surface is specified in a text file (hereafter referred to as the datafile) that may be created with any standard text editor. (The `.fe` extension I always use for datafiles stands for facet-edge, which refers to the internal data structure used to represent the surface. You may use any name you wish for a datafile.)

The basic operation of the Evolver is to read in a datafile and take commands from the user. The main command prompt is

```
Enter command:
```

Basic commands are one letter (case is significant), sometimes with a numerical parameter. The most frequently used commands are:

`g` *n*	do *n* iterations
`s`	show surface on screen (or `P` option 8 for geomview)
`r`	refine triangulation of surface
`P`	graphics output (option 3 for PostScript)
`q`	quit

There is also a more elaborate command language (in which case is not significant). Commands must be followed with the `ENTER` key; Evolver only reads complete lines.

An iteration is one evolution step. The motion for the step is calculated as follows: First, the force on each vertex is calculated from the gradient of the total energy of the surface as a function of the position of that vertex. The force gives the direction of motion. Second, the force is made to conform to whatever constraints are applicable. Third, the actual motion is found by multiplying the force by a global scale factor.

§3.2. Example 1. Cube evolving into a sphere.

A sample datafile `cube.fe` comes with Evolver. The initial surface is a unit cube. The surface bounds one body, and the body is constrained to have volume 1. There is no gravity or any other force besides surface tension. Hence the minimal energy surface will turn out to be a sphere. This example illustrates the basic datafile format and some basic commands.

Let's look at the datafile that specifies the initial unit cube (see Figure 1 for a sketch):

```
// cube.fe
// Evolver data for cube of prescribed volume.

vertices  /* given by coordinates */
1  0.0 0.0 0.0
2  1.0 0.0 0.0
3  1.0 1.0 0.0
4  0.0 1.0 0.0
5  0.0 0.0 1.0
6  1.0 0.0 1.0
7  1.0 1.0 1.0
8  0.0 1.0 1.0

edges  /* given by endpoints */
1   1 2
2   2 3
3   3 4
4   4 1
5   5 6
6   6 7
7   7 8
8   8 5
9   1 5
10  2 6
11  3 7
12  4 8

faces  /* given by oriented edge loop */
1   1 10 -5  -9
2   2 11 -6 -10
3   3 12 -7 -11
4   4  9 -8 -12
5   5  6  7   8
6  -4 -3 -2  -1

bodies  /* one body, defined by its oriented faces */
1   1 2 3 4 5 6  volume 1
```

The datafile is organized in lines, with one geometric element defined per line. Vertices must be defined first, then edges, then faces, then bodies. Each element is numbered for later reference in the datafile.

Comments are delimited by `/* */` as in C, or follow `//` until the end of the line as in C++. Case is not significant, and all input is lower-cased immediately. Hence error messages about your datafiles will refer to items in lower case, even when you typed them in upper case.

The datafile syntax is based on keywords. The keywords `VERTICES`, `EDGES`, `FACES`, and `BODIES` signal the start of the respective sections. Note that the faces are not necessarily triangles (which is why they are called `FACES` and not `FACETS`). Any non-triangular face will be automatically triangulated by putting a vertex at its center and putting in edges to each of the original vertices. Faces don't have to be planar. Note that a minus sign on an edge means that the edge is traversed in the opposite direction from that defined for it in the `EDGES` section. The face oriented normal is defined by the usual right hand rule. The cube faces all have outward normals, so they all are positive in the body list. In defining a body, the boundary faces must have outward normals. If a face as defined has an inward normal, it must be listed with a minus sign.

That the body is constrained to have a volume of 1 is indicated by the keyword `VOLUME` after the body definition, with the value of the volume following. Any attributes or properties an element has are given on the same line after its definition.

Start Evolver and load the datafile with the command line

```
evolver cube.fe
```

You should get a prompt

```
Enter command:
```

If you have `geomview`, do command `P` option 8 to get a display. It takes a couple of seconds to initialize. You can manipulate the `geomview` display as usual independently of the Evolver. Evolver will automatically update the image whenever the surface changes.

If you do not have `geomview` and if you have screen graphics linked in, give the command `s` to show the surface. You should see a square divided into four triangles by diagonals. This is the front side of the cube; hidden surfaces are removed by default. On some systems, you can manipulate the displayed surface with the mouse (particularly Windows NT/95/98; hit the 'h' key with the mouse focus in the graphics window to get a summary of the possibilities). The graphics command prompt is

```
Graphics command:
```

It takes strings of letters, each letter making a viewing transformation on the surface: The most used ones are

`r`	rotate right by 6 degrees
`l`	rotate left by 6 degrees
`u`	rotate up by 6 degrees
`d`	rotate down by 6 degrees
`R`	reset to original position
`q`	quit back to main command prompt

Try `rrdd` to get an oblique view of the cube. Any transformations you make will remain in effect the next time you show the surface. Now do `q` to get back to the main prompt.

Now do some iterations. Give the command `g 5` to do 5 iterations. You should get this:

```
5.  area:    5.11442065156005 energy:    5.11442065156005  scale:  0.186828
4.  area:    5.11237323810972 energy:    5.11237323810972  scale:  0.21885
3.  area:    5.11249312304592 energy:    5.11249312304592  scale:  0.204012
2.  area:    5.11249312772740 energy:    5.11249312772740  scale:  0.20398
1.  area:    5.11249312772740 energy:    5.11249312772740  scale:  0.554771
```

Note that after each iteration a line is printed with the iterations countdown, area, energy, and current scale factor. By default, the Evolver seeks the optimal scale factor to minimize energy. At first, there are large motions, and the volume constraint may not be exactly satisfied. The energy may increase due to the volume constraint taking hold. Volume constraints are not exactly enforced, but each iteration tries to bring the volume closer to the target. Here that results in increases in area. You can find the current volumes with the `v` command:

```
Body       target volume         actual volume        pressure
  1     1.000000000000000     0.999999779366360   3.408026016427987
```

The pressure in the last column is actually the Lagrange multiplier for the volume constraint. Now let's refine the triangulation with the `r` command. This subdivides each facet into four smaller similar facets. The printout here gives the counts of the geometric elements and the memory they take:

```
Vertices:  50 Edges:  144 Facets:  96  Facetedges:  288 Memory:  27554
```

Iterate another 10 times:

```
10.  area:  4.908899804670224 energy:  4.908899804670224  scale:  0.268161
9.  area:  4.909526310166165 energy:  4.909526310166165  scale:  0.204016
8.  area:  4.909119925577212 energy:  4.909119925577212  scale:  0.286541
7.  area:  4.908360229118204 energy:  4.908360229118204  scale:  0.304668
6.  area:  4.907421919968726 energy:  4.907421919968726  scale:  0.373881
5.  area:  4.906763705259419 energy:  4.906763705259419  scale:  0.261395
4.  area:  4.906032256943935 energy:  4.906032256943935  scale:  0.46086
3.  area:  4.905484754688263 energy:  4.905484754688263  scale:  0.238871
2.  area:  4.904915540917190 energy:  4.904915540917190  scale:  0.545873
1.  area:  4.904475138593070 energy:  4.904475138593070  scale:  0.227156
```

You can continue iterating and refining as long as you have time and memory.

Eventually, you will want to quit. So give the `q` command. You get

```
Enter new datafile name (none to continue, q to quit):
```

You can start a new surface by entering a datafile name (it can be the same one you just did, to start over), or continue with the present surface by hitting `ENTER` with no name (in case you pressed `q` by accident, or suddenly you remember something you didn't do), or you can really quit with another `q`.

§3.3. Example 2. Mound with gravity.

This example is a mound of liquid sitting on a tabletop with gravity acting on it. The contact angle between the drop surface and the tabletop is adjustable, to simulate the different degrees to which the liquid wets the table. This example illustrates macros, variables, constraints with energy, and omitting faces from body surfaces.

The drop starts as a cube with one face (face 6 of example 1) on the tabletop (the $z = 0$ plane). The most straightforward way to specify a contact angle is to declare face 6 to be constrained to stay on the tabletop and give it a surface tension different than the default of 1. But this leads to problems described below. The way the contact angle is handled instead is to omit face 6 and give the edges around face 6 an energy integrand that results in the same energy we would get if we did include face 6. If we let the interface energy density for face 6 be T, then we want a vectorfield $\vec{w}$ such that

$$\int\int_{face\ 6} T\vec{k}\cdot d\vec{S} = \int_{\partial(face\ 6)} \vec{w}\cdot d\vec{l}.$$

So by Green's Theorem, all we need is $curl\ \vec{w} = T\vec{k}$, and I have used $\vec{w} = -Ty\vec{i}$.

I have chosen to parameterize the contact angle as the angle in degrees between the table and the surface on the interior of the drop. This angle can be adjusted with the `A` command at run time. I could have made T the parameter directly, but then I wouldn't have had an excuse to show a macro.

Here is the datafile `mound.fe` (see figure 2 for a sketch):

```
// mound.fe
// Evolver data for drop of prescribed volume sitting on plane with gravity.
// Contact angle with plane can be varied interactively with A command.

PARAMETER angle = 90    // interior angle between plane and surface, degrees

#define T  (-cos(angle*pi/180))  // virtual tension of facet on plane

constraint 1   /* the table top */
formula:  z = 0
energy:    // for contact angle
e1:  -T*y
e2:  0
e3:  0

vertices
1   0.0  0.0 0.0  constraint 1  /* 4 vertices on plane */
2   1.0  0.0 0.0  constraint 1
3   1.0  1.0 0.0  constraint 1
4   0.0  1.0 0.0  constraint 1
5   0.0  0.0 1.0
6   1.0  0.0 1.0
7   1.0  1.0 1.0
8   0.0  1.0 1.0
9   2.0  2.0 0.0  fixed   /* for table top, display purposes only */
10  2.0 -1.0 0.0  fixed
11 -1.0 -1.0 0.0  fixed
12 -1.0  2.0 0.0  fixed

edges  /* given by endpoints and attribute */
1   1 2    constraint 1 /* 4 edges on plane */
2   2 3    constraint 1
3   3 4    constraint 1
4   4 1    constraint 1
5   5 6
6   6 7
7   7 8
8   8 5
9   1 5
10  2 6
11  3 7
12  4 8
13  9 10   fixed  /* for table top */
14 10 11   fixed
15 11 12   fixed
16 12  9   fixed

faces  /* given by oriented edge loop */
1   1 10 -5  -9
2   2 11 -6 -10
3   3 12 -7 -11
```

```
4   4  9 -8 -12
5   5  6  7   8
7  13 14 15  16  density 0 fixed /* table top for display */

bodies  /* one body, defined by its oriented faces */
1   1 2 3 4 5   volume 1  density 1
```

The mound itself was basically copied from `cube.fe`, but with face 6 deleted. The reason for this is that face 6 is not needed, and would actually get in the way. It is not needed for the volume calculation since it would always be at $z = 0$ and thus not contribute to the surface integral for volume. The bottom edges of the side faces are constrained to lie in the plane $z = 0$, so face 6 is not needed to keep them from catastrophically shrivelling up. We could have handled the contact angle by including face 6 with a surface tension equal to the interface energy density between the liquid and surface, but that can cause problems if the edges around face 6 try to migrate inward. After refinement a couple of times, interior vertices of the original face 6 have no forces acting on them, so they don't move. Hence it would be tough for face 6 to shrink when its outer vertices ran up against its inner vertices. The tabletop face, face 7, is entirely extraneous to the calculations. Its only purpose is to make a nice display. You could remove it and all its vertices and edges without affecting the shape of the mound. It's constraint 1 that is the tabletop as far as the mound is concerned. To see what happens with the bottom face present, load moundB.fe and do "run".

Now run Evolver on mound.fe. Refine and iterate a while. You should get a nice mound. It's not a hemisphere, since gravity is on by default with $G = 1$. If you use the `G` command to set `G 0` and iterate a while, you get a hemisphere. Try using the `A` command to change the contact angle, to 45 degrees or 135 degrees for example. You can also play with gravity. Set `G 10` to get a flattened drop, or `G -5` to get a drop hanging from the ceiling. `G -10` will cause the drop to try to break loose, but it can't, since its vertices are still constrained.

§3.4. Example 3. Catenoid.

The catenoid is the minimal surface formed between two rings not too far apart. In cylindrical coordinates, its equation is $r = (1/a)\cosh(az)$. In `cat.fe`, both the upper and lower rings are given as one-parameter boundary wires. The separation and radius are parameters, so you can change them during a run with the `A` command. The initial radius given is the minimum for which a catenoid can exist for the given separation of the rings. To get a stable catenoid, you will have to increase this value. However, if you do run with the original value, you can watch the neck pinch out. The initial surface consists of six rectangles forming a cylinder between the two circles. The vertices on the boundaries are fixed, elsewise they would slide along the boundary to short-cut the curvature; two diameters is shorter than one circumference. The boundary edges are fixed so that vertices arising from subdividing the edges are likewise fixed.

Here is the catenoid datafile:

```
// cat.fe
// Evolver data for catenoid.

PARAMETER  RMAX = 1.5088795   // ring radius; here critical value
PARAMETER  ZMAX = 1.0         // ring distance from midplane

boundary 1 parameters 1     //  upper ring
x1:    RMAX * cos(P1)
x2:    RMAX * sin(P1)
x3:    ZMAX

boundary 2 parameters 1    //   lower ring
```

```
x1:    RMAX * cos(P1)
x2:    RMAX * sin(P1)
x3:    -ZMAX

vertices   // given in terms of boundary parameter
1    0.00  boundary 1   fixed
2    pi/3  boundary 1   fixed
3  2*pi/3  boundary 1   fixed
4    pi    boundary 1   fixed
5  4*pi/3  boundary 1   fixed
6  5*pi/3  boundary 1   fixed
7    0.00  boundary 2   fixed
8    pi/3  boundary 2   fixed
9  2*pi/3  boundary 2   fixed
10   pi    boundary 2   fixed
11 4*pi/3  boundary 2   fixed
12 5*pi/3  boundary 2   fixed

edges  /* given by endpoints */
1    1  2  boundary 1   fixed
2    2  3  boundary 1   fixed
3    3  4  boundary 1   fixed
4    4  5  boundary 1   fixed
5    5  6  boundary 1   fixed
6    6  1  boundary 1   fixed
7    7  8  boundary 2   fixed
8    8  9  boundary 2   fixed
9    9  10 boundary 2   fixed
10   10 11 boundary 2   fixed
11   11 12 boundary 2   fixed
12   12 7  boundary 2   fixed
13   1  7
14   2  8
15   3  9
16   4  10
17   5  11
18   6  12

faces  /* given by oriented edge loops */
1   1 14 -7 -13
2   2 15 -8 -14
3   3 16 -9 -15
4   4 17 -10 -16
5   5 18 -11 -17
6   6 13 -12 -18
```

The parameter in a boundary definition is always `P1` (and `P2` in a two-parameter boundary). The Evolver can handle periodic parameterizations, as is done in this example.

Try this sequence of commands (displaying at your convenience):

`r` (refine to get a crude, but workable, triangulation)
`u` (equiangulation makes much better triangulation)

```
g 120   (takes this many iterations for neck to collapse)
t       (collapse neck to single vertex by eliminating all edges shorter than 0.05)
0.05
o       (split neck vertex to separate top and bottom surfaces)
g       (spikes collapse)
```

The catenoid shows some of the subtleties of evolution. Suppose the initial radius is set to `RMAX` = 1.0 and the initial height to `ZMAX` = 0.55. Fifty iterations with optimizing scale factor result in an area of 6.458483. At this point, each iteration is reducing the area by only .0000001, the triangles are all nearly equilateral, everything looks nice, and the innocent user might conclude the surface is very near its minimum. But this is really a saddle point of energy. Further iteration shows that the area change per iteration bottoms out about iteration 70, and by iteration 300 the area is down to 6.4336. The triangulation really wants to twist around so that there are edges following the lines of curvature, which are vertical meridians and horizontal circles. Hence the optimum triangulation appears to be rectangles with diagonals.

The evolution can be speeded up by turning on the conjugate gradient method with the '`U`' command. For conjugate gradient cognoscenti, the saddle point demonstrates the difference between the Fletcher-Reeves and Polak-Ribiere versions of conjugate gradient (§7.8.3). The saddle point seems to confuse the Fletcher-Reeves version (which used to be the default). However, the Polak-Ribiere version (the current default) has little problem. The `U` toggles conjugate gradient on and off, and `ribiere` toggles the Polak-Ribiere version. With Fletcher-Reeves conjugate gradient in effect, the saddle point is reached at iteration 17 and area starts decreasing again until iteration 30, when it reaches 6.4486. But then iteration stalls out, and the conjugate gradient mode has to be turned off and on to erase the history vector. Once restarted, another 20 iterations will get the area down to 6.4334. In Polak-Ribiere mode, no restart is necessary.

Exercise for the reader: Get the Surface Evolver to display an unstable catenoid by declaring the catenoid facets to be the boundary of a body, and adjusting the body volume with the `b` command to get zero pressure.

§3.5. Example 4. Torus partitioned into two cells.

This example has a flat 3-torus (i.e. periodic boundary conditions) divided into two bodies. The unit cell is a unit cube, and the surface has the topology of Kelvin's partitioning of space into tetrakaidecahedra [TW], which was the least area partitioning of space into equal volumes known until recently [WP]. The datafile handles the wrapping of edges around the torus by specifying for each direction whether an edge wraps positively (+), negatively (-), or not at all (*).

The display of a surface in a torus can be done several ways. The `connected` command (my favorite) shows each body as a single unit. The `clipped` command shows the surface clipped to the fundamental parallelpiped. The `raw_cells` command shows the unedited surface.

The Weaire-Phelan structure [WP] is in the datafile `phelanc.fe`. It has area 0.3% less than Kelvin's.

```
// twointor.fe
// Two Kelvin tetrakaidecahedra in a torus.

TORUS_FILLED   // signals that domain is a torus and bodies fill it.

periods
1.000000 0.000000 0.000000
0.000000 1.000000 0.000000
0.000000 0.000000 1.000000

vertices // values from another program
1  0.499733 0.015302 0.792314
2  0.270081 0.015548 0.500199
3  0.026251 0.264043 0.500458
```

```
4  0.755123 0.015258 0.499302
5  0.026509 0.499036 0.794636
6  0.500631 0.015486 0.293622
7  0.025918 0.750639 0.499952
8  0.499627 0.251759 0.087858
9  0.256701 0.499113 0.087842
10 0.026281 0.500286 0.292918
11 0.500693 0.765009 0.086526
12 0.770240 0.499837 0.087382

edges // with wraps in axis directions
1     1 2  * * *
2     2 3  * * *
3     1 4  * * *
4     3 5  * * *
5     2 6  * * *
6     2 7  * - *
7     1 8  * * +
8     4 6  * * *
9     5 9  * * +
10    3 10 * * *
11    3 4  - * *
12    6 8  * * *
13    6 11 * - *
14    7 4  - + *
15    8 12 * * *
16    9 8  * * *
17    9 11 * * *
18    10 7 * * *
19    11 1 * + -
20    12 5 + * -
21    5 7  * * *
22   11 12 * * *
23   10 12 - * *
24    9 10 * * *

faces
1     1 2 4 9 16 -7
2     -2 5 12 -16 24 -10
3     -4 10 18 -21
4     7 15 20 -4 11 -3
5     -1 3 8 -5
6     6 14 -11 -2
7     5 13 -17 24 18 -6
8     -12 13 19 7
9     -16 17 22 -15
10    -10 11 8 12 15 -23
11    -21 9 17 19 1 6
12    -14 -18 23 -22 -13 -8
13    -24 -9 -20 -23
14    -19 22 20 21 14 -3

bodies
```

```
1    -1 -2 -3 -4 -5 9 7 11 -9 10 12 5 14 3 volume 0.500
2     2 -6 -7 8 -10 -12 -11 -13 1 13 -14 6 4 -8 volume 0.500
```

§3.6. Example 5. Ring around rotating rod.

This example consists of a ring of liquid forming a torus around a rod rotating along its long axis (z axis) in weightlessness. The liquid has controllable contact angle with the rod. The interesting question is the stability of the ring as the spin increases.

The effect of the rotation is incorporated in the energy through an integral using the divergence theorem:

$$E = -\int\int\int_B \frac{1}{2}\rho r^2\omega^2 dV = -\int\int_{\partial B} \frac{1}{2}\rho\omega^2(x^2+y^2)z\vec{k}\cdot \vec{dA}$$

where B is the region of the liquid, ρ is the fluid density and ω is the angular velocity. Note the energy is negative, because spin makes the liquid want to move outward. This has to be countered by surface tension forces holding the liquid on the rod. If ρ is negative, then one has a toroidal bubble in a rotating liquid, and high spin stabilizes the torus. The spin energy is put in the datafile using the named quantity syntax (see below). "`centrip`" is a user-chosen name for the quantity, "`energy`" declares that this quantity is part of the total energy, "`global_method`" says that the following method is to be applied to the whole surface, "`facet_vector_integral`" is the pre-defined name of the method that integrates vector fields over facets, and "`vector_integrand`" introduces the components of the vectorfield.

The rod surface is defined to be constraint 1 with equation $x^2 + y^2 = R^2$, where R is the radius of the rod. The contact energy of the liquid with the rod is taken care of with an edge integral over the edges where the liquid surface meets the rod:

$$E = \int\int_S -T\cos(\theta)dA = -T\cos(\theta)\int_{\partial S} z ds = T\cos(\theta)\int_{\partial S}\frac{z}{R}(y\vec{i} - x\vec{j})\cdot \vec{ds}$$

Here S is the rod surface not included as facets in ∂B, T is the surface tension of the free surface, and θ is the internal contact angle.

Constraint 2 is a horizontal symmetry plane. By assuming symmetry, we only have to do half the work.

Constraint 3 is a one-sided constraint that keeps the liquid outside the rod. Merely having boundary edges on the rod with constraint 1 is not enough in case the contact angle is near 180° and the liquid volume is large. Constraint 3 may be put on any vertices, edges, or faces likely to try to invade the rod. However, it should be noted that if you put constraint 3 on only some vertices and edges, equiangulation will be prevented between facets having different constraints.

Constraint 4 is a device to keep the vertices on the rod surface evenly spaced. Edges on curved constraints often tend to become very uneven, since long edges short-cutting the curve can save energy. Hence the need for a way to keep the vertices evenly spread circumferentially, but free to move vertically. One way to do that is with another constraint with level sets being vertical planes through the z axis at evenly spaced angles. Constraint 4 uses the real modulus function with arctangent to create a periodic constraint. Each refinement, the parameters need to be halved to cut the period in half. This is done with the special "`rr`" refinement command at the end of the datafile. The regular "`r`" refinement command should never be used on this file. Note that autorecalc is temporarily turned off to prevent projecting vertices to the constraint when it is in an invalid state. Also note the $\pi/6$ offset to avoid the discontinuity in the modulus function. $\pi/6$ was cleverly chosen so that all refinements would also avoid the discontinuity.

For detecting stability, one can perturb the torus by defining a command

```
perturb := set vertex y y+.01 where not on_constraint 1
```

This sets the y coordinate of each vertex to $y + .01$. For convenience, this command is defined in the "`read`" section at the end of the datafile, where you can put whatever commands you want to execute immediately after the datafile is loaded. To detect small perturbations, and get numerical values for the size of perturbations, the y moment of the liquid is calculated in the named quantity "`ymoment`". It is not part of the energy, as indicated by the "`info_only`" keyword. You can see the value with the "`v`" command.

```
// ringblob.fe

// Toroidal liquid ring on a rotating rod in weightlessness.
// Half of full torus
// Using second periodic constraint surface intersecting rod to
// confine vertices on rod to vertical motion.

// Important note to user:  Use only the 'rr' command defined at
// the end of this file to do refinement.    This is due to the
// nature of constraint 4 below.

// This permits drawing both halves of the ring
view_transforms 1
1 0 0 0
0 1 0 0
0 0 -1 0
0 0 0 1

// Basic parameters.    These may be adjusted at runtime with the
// 'A' command or by assignment.
parameter rodr = 1  // rod radius
parameter spin = 0.5 // angular velocity
parameter angle = 90 // internal contact angle with rod
parameter tens = 1   // surface tension of free surface
#define rode (tens*cos(angle*pi/180))  // liquid-rod contact energy
parameter dens = 1  // density of liquid, negative for bubble

// spin centripetal energy
quantity centrip energy global_method facet_vector_integral
vector_integrand:
q1:  0
q2:  0
q3:  -0.5*dens*spin*spin*(x^2+y^2)*z

// y moment, for detecting instability
quantity ymoment info_only global_method facet_vector_integral
vector_integrand:
q1:  0
q2:  0
q3:  y*z

// Constraint for vertices and edges confined to rod surface,
// with integral for blob area on rod
constraint 1
formula:  x^2 + y^2 = rodr^2
energy:
e1:  rode*z*y/rodr
e2:  -rode*z*x/rodr
e3:  0
```

```
// Horizontal symmetry plane
constraint 2
formula:  z = 0

// Rod surface as one-sided constraint, to keep stuff from caving in
// Can be added to vertices, edges, facets that try to cave in
constraint 3 nonnegative
formula:  x^2 + y^2 = rodr^2

// Constraint to force vertices on rod to move only vertically.
// Expressed in periodic form, so one constraint fits arbitrarily
// many vertices.  Note offset to pi/6 to avoid difficulties with
// modulus discontinuity at 0.
parameter pp = pi/2    /* to be halved each refinement */
parameter qq = pi/6    /* to be halved each refinement */
constraint 4
formula:    (atan2(y,x)+pi/6) % pp = qq

//initial dimensions
#define ht 2
#define wd 3

vertices
1  0   -wd 0  constraints 2    // equatorial vertices
2  wd    0 0  constraints 2
3  0    wd 0  constraints 2
4  -wd   0 0  constraint 2
5  0   -wd ht                 // upper outer corners
6  wd    0 ht
7  0    wd ht
8  -wd   0 ht
9  0 -rodr ht constraints 1,4   // vertices on rod
10 rodr  0 ht constraints 1,4
11 0  rodr ht constraints 1,4
12 -rodr  0 ht constraints 1,4

edges
1   1 2 constraint 2  // equatorial edges
2   2 3 constraint 2
3   3 4 constraint 2
4   4 1 constraint 2
5   5 6               // upper outer edges
6   6 7
7   7 8
8   8 5
9   9  10 constraint 1,4   // edges on rod
10  10 11 constraint 1,4
11  11 12 constraint 1,4
12  12  9 constraint 1,4
13   1  5        // vertical outer edges
14   2  6
15   3  7
```

```
16   4  8
17   5  9        // cutting up top face
18   6 10
19   7 11
20   8 12

faces  /* given by oriented edge loop */
1   1 14 -5 -13 tension tens // side faces
2   2 15 -6 -14 tension tens  // Remember you can't change facet tension
3   3 16 -7 -15 tension tens  // dynamically just by changing tens; you have
4   4 13 -8 -16 tension tens  // to do "tens := 2; set facet tension tens"
5   5 18 -9 -17 tension tens  // top faces
6   6 19 -10 -18 tension tens
7   7 20 -11 -19 tension tens
8   8 17 -12 -20 tension tens

bodies  /* one body, defined by its oriented faces */
1   1 2 3 4 5 6 7 8  volume 25.28

read  // some initializations
transforms off     // just show fundamental region to start with

// special refinement command redefinition
r :::=  autorecalc off;  pp := pp/2; qq := qq % pp; 'r'; autorecalc on;

// a slight perturbation, to check stability
perturb := set vertex y y+.01 where not on_constraint 1


hessian_normal // to make Hessian well-behaved
```

§3.7. Example 6. Column of liquid solder.

Here we have a tiny drop of liquid solder that bridges between two parallel, horizontal planes, the lower at $z = 0$ and the upper at $z =$ `ZH`. On each plane there is a circular pad that the solder perfectly wets, and the solder is perfectly nonwetting off the pads. This would be just a catenoid problem with fixed volume, except that the pads are offset, and it is desired to find out what horizontal and vertical forces the solder exerts. The surface is defined the same way as in the catenoid example, except the upper boundary ring has a shift variable "`SHIFT`" in it to provide an offset in the y direction. This makes the shift adjustable at run time. Since the top and bottom facets of the body are not included, the constant volume they account for is provided by content integrals around the upper boundary, and the gravitational energy is provided by an energy integral. One could use the volconst attribute of the body instead for the volume, but then one would have to reset that every time ¡tt¿ZH¡/tt¿ changed.

The interesting part of this example is the calculation of the forces. One could incrementally shift the pad, minimize the energy at each shift, and numerically differentiate the energy to get the force. Or one could set up integrals to calculate the force directly. But the simplest method is to use the Principle of Virtual Work by shifting the pad, recalculating the energy without re-evolving, and correcting for the volume change. Re-evolution is not necessary because when a surface is at an equilibrium, then by definition any perturbation that respects constraints does not change the energy to first order. To adjust for changes in constraints such as volume, the Lagrange multipliers (pressure for the volume constraint) tell how much the energy changes for given change in the constraints:

$$DE = L^T DC$$

where DE is the energy change, L is the vector of Lagrange multipliers and DC is the vector of constraint value changes. Therefore, the adjusted energy after a change in a parameter is

$$E_{adj} = E_{raw} - L^T DC$$

where E_{raw} is the actual energy and DC is the vector of differences of constraint values from target values. The commands `do_yforce` and `do_zforce` in the datafile do central difference calculations of the forces on the top pad, and put the surface back to where it was originally. Note that the perturbations are made smoothly, i.e. the shear varies linearly from bottom to top. This is not absolutely necessary, but it gives a smoother perturbation and hence a bit more accuracy.

```
// column.fe
// Example of calculating forces exerted by a
// column of liquid solder in shape of skewed catenoid.

// All units cgs
parameter RAD = 0.05     // ring radius
parameter ZH = 0.08      // total height
parameter SHIFT = 0.025    // shift
#define SG 8      // specific gravity of solder
#define TENS 460  // surface tension of solder
#define GR  980   // acceleration of gravity

gravity_constant GR

BOUNDARY 1  PARAMETERS 1
X1:  RAD*cos(P1)
X2:  RAD*sin(P1) + SHIFT
X3:  ZH
CONTENT  // used to compensate for missing top facets
c1:  0
c2:  -ZH*x
c3:  0
ENERGY  // used to compensate for gravitational energy under top facets
e1:  0
e2:  GR*ZH^2/2*x
e3:  0

BOUNDARY 2  PARAMETERS 1
X1:  RAD*cos(P1)
X2:  RAD*sin(P1)
X3:  0

vertices   // given in terms of boundary parameter
1    0.00  boundary 1   fixed
2    pi/3  boundary 1   fixed
3  2*pi/3  boundary 1   fixed
4    pi    boundary 1   fixed
5  4*pi/3  boundary 1   fixed
```

```
6  5*pi/3  boundary 1   fixed
7    0.00  boundary 2   fixed
8    pi/3  boundary 2   fixed
9  2*pi/3  boundary 2   fixed
10   pi    boundary 2   fixed
11 4*pi/3  boundary 2   fixed
12 5*pi/3  boundary 2   fixed

edges
1    1  2  boundary 1   fixed
2    2  3  boundary 1   fixed
3    3  4  boundary 1   fixed
4    4  5  boundary 1   fixed
5    5  6  boundary 1   fixed
6    6  1  boundary 1   fixed
7    7  8  boundary 2   fixed
8    8  9  boundary 2   fixed
9    9  10 boundary 2   fixed
10   10 11 boundary 2   fixed
11   11 12 boundary 2   fixed
12   12 7  boundary 2   fixed
13   1  7
14   2  8
15   3  9
16   4  10
17   5  11
18   6  12

faces
1   1 14  -7 -13   density TENS
2   2 15  -8 -14   density TENS
3   3 16  -9 -15   density TENS
4   4 17 -10 -16   density TENS
5   5 18 -11 -17   density TENS
6   6 13 -12 -18   density TENS

bodies
1    -1 -2 -3 -4 -5 -6 volume 0.00045 density SG

read

// horizontal force on upper pad by central differences
dy := .0001
do_yforce := { oldshift := shift; shift := shift + dy;
               set vertex y  y+dy*z/zh; // uniform shear
               recalc;
               energy1 := total_energy -
                     body[1].pressure*(body[1].volume - body[1].target);
               oldshift := shift; shift := shift - 2*dy;
               set vertex y  y-2*dy*z/zh; // uniform shear
               recalc;
               energy2 := total_energy -
                     body[1].pressure*(body[1].volume - body[1].target);
```

```
                  yforce := -(energy1-energy2)/2/dy;
                  printf "restoring force:  %20.15f\n",yforce;
                  // restore everything
                  oldshift := shift; shift := shift + dy;
                  set vertex y  y+dy*z/zh; // uniform shear
                  recalc;
                }

// vertical force on upper pad by central differences.
dz := .0001
do_zforce := { oldzh := zh; zh := zh + dz;
                  set vertex z  z+dz*z/oldzh; recalc; // uniform stretch
                  energy1 := total_energy -
                        body[1].pressure*(body[1].volume - body[1].target);
                  oldzh := zh; zh := zh - 2*dz;
                  set vertex z  z-2*dz*z/oldzh; recalc; // uniform stretch
                  energy2 := total_energy -
                        body[1].pressure*(body[1].volume - body[1].target);
                  zforce := -(energy1-energy2)/2/dz;
                  printf "vertical force:    %20.15f\n",zforce;
                  // restore everything
                  oldzh := zh; zh := zh + dz;
                  set vertex z  z+dz*z/oldzh; recalc; // uniform stretch
                }
```

§3.8. Example 7. Rocket fuel tank.

This example discusses a cylindrical flat-ended tank in low gravity with some liquid inside. Complete understanding of this example qualifies you as an Evolver Power User.

The initial configuration is shown in figure 3. The axis is horizontal with the base at the left. We will assume the fuel is in a ring around the base of the tank. The fuel comprises one body, and the boundary of the body has four components:

1. The covered part of the flat circular base at $y = 0$, implemented as constraint 1..
2. The cylindrical wall at $x^2 + z^2 = R^2$, implemented as constraint 2..
3. The triangular facets (8 of them in figure 3).
4. The gaps between the edges on the wall (the "gap edges", edges 5,6,7,8 in figure 3) and the wall. We will assume that the gap surfaces are generated by lines radially outward from the central axis (the y axis) through the points of the gap edges.

To specify contact angles, let θ_B be the interior contact angle with the base, and let θ_W be the interior contact angle with the cylindrical wall.

§3.8.1. Surface energy.

Base.

The surface energy here comes from the contact angle. The angle is represented by pretending the uncovered base has surface tension $T_B = cos(\theta_B)$. That we handle with an energy integral on constraint 1. We want

$$E = \int\int_{base} T_B \vec{j} \cdot d\vec{A}. \tag{1}$$

By Stokes' theorem, this is the integral around the edges in the base (edges 1,2,3,4)

$$E = \int_{edges} T_B z\vec{i} \cdot \vec{ds}, \tag{2}$$

where the edges are oriented oppositely from figure 3. Hence the energy integrand for constraint 1 should be $-T_B z\vec{i}$.

Wall.

The contact angle is represented by pretending the uncovered wall has surface tension $T_W = cos(\theta_W)$. The uncovered wall in infinite, so we will assume equivalently that the covered wall has tension $-T_W$. The energy is

$$E = \int\int_{wall} -T_W dA = \int\int_{wall} -T_W \frac{1}{R}(x\vec{i} + z\vec{k}) \cdot \vec{dA}. \tag{3}$$

We want to turn this into an integral over the boundary of the wall, which is made of two paths, the base circumference and the "gap trace" where the gap surfaces meet the wall. This calls for a vector potential, but our integrand above does not have zero divergence. But we could choose a divergenceless integrand that is equal on the wall:

$$E = \int\int_{wall} \vec{F} \cdot \vec{dA}. \tag{4}$$

where

$$\vec{F} = -T_W \frac{R}{(x^2 + z^2)}(x\vec{i} + z\vec{k}). \tag{5}$$

Now we can get a vector potential $\vec{w}$

$$\vec{w} = -T_W \frac{Ry}{(x^2 + z^2)}(-z\vec{i} + x\vec{k}), \quad \nabla \times \vec{w} = \vec{F}. \tag{6}$$

Hence

$$E = \int_{basecircum} \vec{w} \cdot ds + \int_{gap\ trace} \vec{w} \cdot ds. \tag{7}$$

The first of these is zero since $y = 0$ there. The second is equal to the integral over the gap edges, since $\nabla \times \vec{w}$ is parallel to the gap surfaces! So

$$E = \int_{gap\ edges} \vec{w} \cdot ds, \tag{8}$$

where the gap edges are oppositely oriented from figure 3. Hence the constraint 2 energy integrand includes $-\vec{w}$.

The facets.

The surface tension energy here is automatically taken care of by the Evolver. We will leave the surface tension as the default value of 1.

The gaps.

We will declare constraint 2 to be `CONVEX` so that Evolver will include a gap surface energy for all the gap edges. We put the spring constant to be 1, as that value give the best approximation to the true area for small gaps.

§3.8.2. Volume.

The body volume $V = \int\int\int dV$ can be calculated as a surface integral $V = \int\int \vec{F} \cdot \vec{dA}$ where $\nabla \cdot F = 1$. The integral over the facets is handled automatically, but there are two choices for $\vec{F}$. We will take $\vec{F} =$

$(x\vec{i}+y\vec{j}+z\vec{k})/3$ by putting `SYMMETRIC_CONTENT` in the datafile. (This choice works better for the tank on its side; for an upright tank, the default $\vec{F}=z\vec{k}$ is easier.) The surface integral has four components:

Base.

This part of the integral is zero, since $y=0$ and the normal is $\vec{j}$.

The wall.

Here the $\vec{j}$ component of $\vec{F}$ can be dropped, and the rest put in the equivalent divergenceless form

$$\vec{F}_W = \frac{R^2}{3(x^2+z^2)}(x\vec{i}+z\vec{k}). \tag{9}$$

This has vector potential

$$\vec{w} = \frac{R^2 y}{3(x^2+z^2)}(-z\vec{i}+x\vec{k}), \qquad \nabla\times\vec{w} = \vec{F}_W. \tag{10}$$

By the same reasoning as for wall area, this reduces to the integral of $-\vec{w}$ over the gap edges. Thus we include $-\vec{w}$ in the content integrand of constraint 2.

Facets.

Evolver takes care of these automatically.

Gaps.

We subtract some radial component from $\vec{F}$ to get a divergenceless vector $\vec{F}_G$ with the same surface integral over the gap surfaces:

$$\vec{F}_G = -\frac{1}{6}x\vec{i}+\frac{1}{3}y\vec{j}-\frac{1}{6}z\vec{k}. \tag{11}$$

This has vector potential

$$\vec{w} = \frac{1}{6}zy\vec{i}-\frac{1}{6}xy\vec{k}. \tag{12}$$

Hence

$$\int\int_{gaps}\vec{F}\cdot dA = \int_{gap\ trace}\vec{w}\cdot ds + \int_{gap\ edges}\vec{w}\cdot ds. \tag{13}$$

The second part we can put directly into the constraint 2 content integrand (with opposite orientation of the edges). The gap trace part we convert to an equivalent integrand $\vec{v}$:

$$\vec{v} = \frac{R^2 y}{6(x^2+z^2)}(z\vec{i}-x\vec{k}). \tag{14}$$

which has radial curl. Hence the gap trace integral is the same as $\int_{gap\ edges}\vec{v}\cdot ds$, this time with the edges in the same orientation as figure 3.

§3.8.3. Gravity.

Just for fun, we're going to put in gravity in such a way that the user can make it act in any direction. Let the acceleration of gravity be $\vec{G}=g_y\vec{j}+g_z\vec{k}$. g_y and g_z will be adjustable constants. Let the fuel density be ρ. Then the gravitational potential energy is

$$E = -\int\int\int_{body}\vec{G}\cdot\vec{x}\rho dV = -\int\int\int_{body}(g_y y+g_z z)\rho dV. \tag{15}$$

First we change this to a surface integral

$$E = -\rho\int\int_{\partial body}\vec{F}\cdot d\vec{A}, \qquad \nabla\cdot\vec{F} = g_y y+g_z z. \tag{16}$$

It turns out convenient to take

$$\vec{F} = (g_y \frac{y^2}{2} + g_z yz)\vec{j}. \tag{17}$$

The surface integral has four components:

Base.

This part is zero since $y = 0$.

Wall.

This part is also zero since $\vec{F}$ is parallel to the wall.

Facets.

This part is taken care of by installing $-\rho\vec{F}$ as the integrand for surface energy 1 for the facets.

Gaps.

We get an equivalent divergence-free integrand $\vec{F}_G$ by adding a radial component to $\vec{F}$:

$$\vec{F}_G = g_y \left(\frac{y^2}{2}\vec{j} - \frac{y}{2}(x\vec{i} + z\vec{k}) \right) + g_z \left(yz\vec{j} - \frac{z}{3}(x\vec{i} + z\vec{k}) \right). \tag{18}$$

This has vector potential $\vec{w}$

$$\vec{w} = \left(g_y \frac{y^2}{4} + g_z \frac{yz}{3} \right) (z\vec{i} - x\vec{k}). \tag{19}$$

This is integrated over the gap edges with negative orientation, so $\rho\vec{w}$ goes into the energy integrand of constraint 2, and over the gap trace. For the gap trace part, we construct an equivalent $\vec{w}_G$ that has radial curl:

$$\vec{w}_G = \left(g_y \frac{y^2}{4} \frac{R^2}{(x^2 + y^2)} + g_z \frac{yz}{3} \frac{R^3}{(x^2 + z^2)^{(3/2)}} \right) (z\vec{i} - x\vec{k}), \tag{20}$$

so

$$\int_{gap\ trace} \vec{w} \cdot \vec{ds} = \int_{gap\ edges} \vec{w}_G \cdot \vec{ds}. \tag{21}$$

Here the integral is over the same orientation as in figure 3, so $-\rho w_G$ also goes into the energy integrand of constaint 2.

§3.8.4. Running.

Refine once before iterating. Iterate and refine to your heart's content. You might try command `u` once in a while to regularize the triangulation. Play around with command `A` to change gravity and wall contact angles. Keep an eye on the scale factor. If it gets towards zero, check out the `t` and `w` histograms.

```
// tankex.fe

// Equilibrium shape of liquid in flat-ended cylindrical tank.

// Tank has axis along y-axis and flat bottom in x-z plane.    This
// is so gravity acting vertically draws liquid toward wall.

// Straight edges cannot conform exactly to curved walls.
// We need to give them an area so that area cannot shrink by straght edges
// pulling away from the walls.  The gaps are also accounted for
// in volume and gravitational energy.
```

```
SYMMETRIC_CONTENT     // for volume calculations

// Contact angles, initially for 45 degrees.
PARAMETER ENDT  = 0.707   /* surface tension of uncovered base   */
PARAMETER WALLT = 0.707   /* surface tension of uncovered side wall */

// Gravity components
PARAMETER GY = 0
PARAMETER GZ = -1

SPRING_CONSTANT 1  // for most accurate gap areas for constraint 2

#define  TR  1.00       /* tank radius */
#define  RHO 1.00       /* fuel density */

constraint 1  // flat base
function:    y = 0
energy:                              // for contact energy line integral
e1:      -ENDT*z
e2:      0
e3:      0

#define wstuff (WALLT*TR*y/(x^2 + z^2))   // common wall area term
#define vstuff (TR^2/3*y/(x^2 + z^2))     // common wall volume term
#define gstuff (GY*TR^2*y^2/4/(x^2 + z^2) + GZ*TR^3*y*z/3/(x^2+z^2)**1.5)
// common gap gravity term

constraint 2  CONVEX         // cylindrical wall
function:    x^2 + z^2 = TR^2
energy:                                // for contact energy and gravity
e1:    -wstuff*z  + RHO*GY*y^2*z/4 + RHO*GZ*y*z^2/3 - RHO*gstuff*z
e2:       0
e3:      wstuff*x  - RHO*GY*y^2*x/4 - RHO*GZ*y*z*x/3 + RHO*gstuff*x
content:                              // so volumes calculated correctly
c1:      vstuff*z - z*y/6 + vstuff*z/2
c2:       0
c3:    -vstuff*x + x*y/6 - vstuff*x/2

// named quantity for arbitrary direction gravity on facets
quantity arb_grav energy method facet_vector_integral global
vector_integrand:
q1:    0
q2:    -RHO*GY*y^2/2 - RHO*GZ*y*z
q3:    0


// Now the specification of the initial shape

vertices
1    0.5  0.0  0.5  constraint 1
2    0.5  0.0 -0.5  constraint 1
3   -0.5  0.0 -0.5  constraint 1
4   -0.5  0.0  0.5  constraint 1
```

```
5     1.0  0.5  0.0  constraint 2
6     0.0  0.5 -1.0  constraint 2
7    -1.0  0.5  0.0  constraint 2
8     0.0  0.5  1.0  constraint 2

edges
1     2  1  constraint 1
2     1  4  constraint 1
3     4  3  constraint 1
4     3  2  constraint 1
5     5  6  constraint 2
6     6  7  constraint 2
7     7  8  constraint 2
8     8  5  constraint 2
9     1  8
10    1  5
11    2  5
12    2  6
13    3  6
14    3  7
15    4  7
16    4  8

faces
1    13   6 -14
2     3  14 -15
3    15   7 -16
4     2  16  -9
5     9   8 -10
6     1  10 -11
7    11   5 -12
8     4  12 -13

bodies
1     1 2 3 4 5 6 7 8   volume 0.6  density 0  /* no default gravity */
```

§3.9. Example 8. Spherical tank.

This example discusses a sperical tank in low gravity with some liquid inside. Complete understanding of this example qualifies you as an Evolver Power User.

The initial configuration is shown in figure 4. The liquid comprises one body, and the boundary of the body has three components:

1. The sphere at $x^2 + y^2 + z^2 = R^2$, implemented as constraint 1.
2. The faceted surface (initially, the single face).
3. The gaps between the edges on the sphere (the "gap edges", edges 1,2,3,4 in figure 4) and the sphere. We will assume that the gap surfaces are generated by lines radially outward from the center of the sphere through the points of the gap edges.

We will often be looking for vectorfields that have radial curl, since their integral over the gap edges will be the same as over the gap trace. As a useful fact for all the curls below, note that if

$$\vec{w} = \frac{z^n}{(x^2+y^2)(x^2+y^2+z^2)^{n/2}}(y\vec{i} - x\vec{j}) \tag{1}$$

then

$$\nabla \times \vec{w} = \frac{nz^{n-1}}{(x^2+y^2+z^2)^{1+n/2}}(x\vec{i} + y\vec{j} + z\vec{k}). \tag{2}$$

To specify contact angles, let θ_W be the interior contact angle with the spherical wall.

§3.9.1. Surface energy.

Wall.

The contact angle is represented by pretending the uncovered wall has surface tension $T_W = cos(\theta_W)$. We will assume equivalently that the covered wall has tension $-T_W$. The energy is

$$E = \int\int_{wall} -T_W \, dA = \int\int_{wall} -T_W \frac{1}{R}(x\vec{i} + y\vec{j} + z\vec{k}) \cdot d\vec{A}. \tag{3}$$

We want to turn this into an integral over the boundary of the covered wall, the "gap trace" where the gap surfaces meet the wall. This calls for a vector potential, but our integrand above does not have zero divergence. But we could choose a divergenceless integrand that is equal on the wall:

$$E = \int\int_{wall} \vec{F} \cdot d\vec{A}, \tag{4}$$

where

$$\vec{F} = -T_W \frac{R^2}{(x^2+y^2+z^2)^{3/2}}(x\vec{i} + y\vec{j} + z\vec{k}). \tag{5}$$

Now we can get a vector potential $\vec{w}$ (see useful fact above with $n = 1$)

$$\vec{w} = -T_W \frac{R^2 z}{(x^2+y^2)(x^2+y^2+z^2)^{1/2}}(y\vec{i} - x\vec{j}), \quad \nabla \times \vec{w} = \vec{F}. \tag{6}$$

Hence

$$E = \int_{gap\ trace} \vec{w} \cdot ds. \tag{7}$$

This is equal to the integral over the gap edges, since $\nabla \times \vec{w}$ is parallel to the gap surfaces. So

$$E = \int_{gap\ edges} \vec{w} \cdot ds, \tag{8}$$

where the gap edges are oppositely oriented from figure 4. Hence the constraint energy integrand includes $-\vec{w}$. Note the potential $\vec{w}$ is zero at $z = 0$, so it is actually measuring area above the equator. So one would need to add a hemisphere's worth of energy $2T_W \pi R^2/3$.

The facets.

The surface tension energy here is automatically taken care of by the Evolver. We will leave the surface tension as the default value of 1.

The gaps.

We will declare constraint 1 to be `CONVEX` so that Evolver will include a gap surface energy for all the gap edges. We put the spring constant to be 1, as that value give the best approximation to the true area for small gaps.

§3.9.2. Volume.

The body volume $V = \int\int\int dV$ can be calculated as a surface integral $V = \int\int \vec{F} \cdot \vec{dA}$ where $\nabla \cdot F = 1$. The integral over the facets is handled automatically, but there are two choices for $\vec{F}$. We will take $\vec{F} = (x\vec{i} + y\vec{j} + z\vec{k})/3$ by putting `SYMMETRIC_CONTENT` in the datafile. Thus

$$V = \frac{1}{3}\int\int_{body\ surface}(x\vec{i} + y\vec{j} + z\vec{k}) \cdot dA. \tag{9}$$

The surface integral has three components:

The wall.

Like wall energy, this is first put in the equivalent divergenceless form

$$\vec{F} = \frac{1}{3}\frac{R^3}{(x^2+y^2+z^2)^{3/2}}(x\vec{i} + y\vec{j} + z\vec{k}). \tag{10}$$

This has vector potential (again useful fact above with $n = 1$)

$$\vec{w} = \frac{1}{3}\frac{R^3 z}{(x^2+y^2)(x^2+y^2+z^2)^{1/2}}(y\vec{i} - x\vec{j}), \quad \nabla \times \vec{w} = \vec{F}. \tag{11}$$

By the same reasoning as for wall area, this reduces to the integral of $-\vec{w}$ over the gap edges. Thus we include $-\vec{w}$ in the content integrand of constraint 1. However, we also note a potential problem at $z = -R$ where $x^2 + y^2 = 0$. We must be sure that this point always remains within the covered surface. As long as it does, it will have a constant effect. In fact, we see that at $z = 0$ the integrand $\vec{w}$ is zero, and the other volume contributions are zero at $z = 0$, so the singularity at $z = -R$ must have a contribution to the volume of half the volume of the sphere. We can account for this by using `VOLCONST 2*pi*R^3/3` with the body.

Facets.

Evolver takes care of these automatically.

Gaps.

The integrand is parallel to the gap surfaces, hence this contribution is zero.

§3.9.3. Gravity.

The sphere being very symmetric, all directions of gravity are the same. Therefore we will use Evolver's default gravity. Let the magnitude of gravity be G (downwards). Let the liquid density be ρ. Then the gravitational potential energy is

$$E = G\int\int\int_{body} z\rho dV. \tag{12}$$

First we change this to a surface integral

$$E = G\rho\int\int_{\partial body}\vec{F} \cdot \vec{dA}, \qquad \nabla \cdot \vec{F} = z. \tag{13}$$

The default gravity mechanism uses

$$\vec{F} = G\rho\frac{z^2}{2}\vec{k}. \tag{14}$$

The surface integral has three components:

Wall.

First, we convert to the equivalent divergence-free integrand

$$\vec{F} = G\rho\frac{R^4 z^3}{2(x^2+y^2+z^2)^3}(x\vec{i}+y\vec{j}+z\vec{k}). \tag{15}$$

Note this has the same dot product with $x\vec{i}+y\vec{j}+z\vec{k}$ on the surface of the sphere. The vector potential is

$$\vec{w} = G\rho\frac{R^4 z^4}{8(x^2+y^2)(x^2+y^2+z^2)^2}(y\vec{i}-x\vec{j}), \quad \nabla\times\vec{w} = \vec{F}. \tag{16}$$

So

$$E = \int_{gap\ trace} \vec{w}\cdot\vec{ds} - \frac{G\rho R^4}{4}. \tag{17}$$

The constant term here is the gravitaional energy of the lower hemisphere, since $\vec{w} = 0$ when $z = 0$. It will not show up in the Evolver energy; you must remember to adjust for it yourself. Due to the radial curl of $\vec{w}$, the integral is also the integral over the gap edges. The orientation is opposite that given for the gap edges, so $-\vec{w}$ goes in the energy for constraint 1.

Facets.

The integrals over facets are automatically taken care of by the Evolver.

Gaps.

We get an equivalent divergence-free integrand $\vec{F}_G$ by adding a radial component $-(z/4)(x\vec{i}+y\vec{j}+z\vec{k})$ to $\vec{F}$:

$$\vec{F}_G = \frac{G\rho}{4}(-xz\vec{i}+-yz\vec{j}+z^2\vec{k}). \tag{18}$$

This has vector potential $\vec{w}$

$$\vec{w} = -\frac{G\rho z^2}{8}(y\vec{i}-x\vec{j}). \tag{19}$$

This is integrated over the gap edges with opposite orientation, so $-\vec{w}$ goes into the energy integrand of constraint 1, and over the gap trace. For the gap trace part, we construct an equivalent $\vec{w}_G$ that has radial curl (using two versions of useful fact with $n = 2$ and $n = 4$ to show radial curl):

$$\begin{aligned}\vec{w}_G &= \frac{G\rho}{8}\frac{z^2(R^2-z^2)}{(x^2+y^2)}(y\vec{i}-x\vec{j}) \\ &= \frac{G\rho}{8}\left(\frac{z^2R^4}{(x^2+y^2)(x^2+y^2+z^2)} - \frac{z^4R^4}{(x^2+y^2)(x^2+y^2+z^2)^2}\right)(y\vec{i}-x\vec{j}) \\ &= \frac{G\rho}{8}\frac{z^2R^4}{(x^2+y^2+z^2)^2}(y\vec{i}-x\vec{j})\end{aligned} \tag{20}$$

so

$$\int_{gap\ trace}\vec{w}\cdot\vec{ds} = \int_{gap\ edges}\vec{w}_G\cdot\vec{ds}. \tag{21}$$

Here the integral is over the same orientation as in figure 4, so w_G also goes into the energy integrand of constaint 1.

§3.9.4. Running.

Refine once before iterating. Iterate and refine to your heart's content. You might try command `u` once in a while to regularize the triangulation. Play around with command `A` to change gravity and wall contact angles. Keep an eye on the scale factor. If it gets towards zero, check out the `t` and `w` histograms.

```
// sphere.fe

// Spherical tank.

SYMMETRIC_CONTENT   // natural for a sphere

/* sphere radius */
#define R  1.00

/* cosine of contact angle */
PARAMETER WALLT = 0.707

/* density of body */
#define RHO   1.0

#define wstuff (R^2*z/(x^2 + y^2)/sqrt(x^2+y^2+z^2))
#define gstuff (G*RHO/8*R^4*z^4/(x^2 + y^2)/(x^2+y^2+z^2)^2)
#define gapstuff (G*RHO*R^4/8*z^2/(x^2+y^2+z^2)^2)

constraint 1    /* the sphere */
formula:        x^2 + y^2 + z^2 = R^2
energy:
e1:        WALLT*wstuff*y - gstuff*y + G*RHO/8*y*z^2 + gapstuff*y
e2:      -WALLT*wstuff*x + gstuff*x - G*RHO/8*x*z^2 - gapstuff*x
e3:0
content:
c1:    -wstuff*y/3
c2:      wstuff*x/3
c3:      0

vertices:
1    1  0  0  constraint 1
2    0  1  0  constraint 1
3   -1  0  0  constraint 1
4    0 -1  0  constraint 1
5    0  0  1  FIXED   // to show back hemisphere
6    0  0 -1  FIXED
7    0  1  0  FIXED
8   -1  0  0  FIXED
9    0 -1  0  FIXED

edges:
1    1  2     constraint 1
2    2  3     constraint 1
3    3  4     constraint 1
4    4  1     constraint 1
5    5  9     constraint 1  FIXED // to show back hemisphere
6    9  6     constraint 1  FIXED
7    6  7     constraint 1  FIXED
8    7  5     constraint 1  FIXED
9    5  8     constraint 1  FIXED
10   9  8     constraint 1  FIXED
```

```
11   6  8     constraint 1  FIXED
12   7  8     constraint 1  FIXED

faces:
1    1 2 3 4
2    5 10 -9  constraint 1 density 0 FIXED   // to show back hemisphere
3   -10 6 11  constraint 1 density 0 FIXED
4   -11 7 12  constraint 1 density 0 FIXED
5    8 9 -12  constraint 1 density 0 FIXED

bodies:    // start with sphere half full
1    1    volume 2*pi*R^3/3  volconst 2*pi*R^3/3
```

§3.10. Example 9. Crystalline integrand.

This example shows how to use Wulff vectors for a crystalline integrand. In general, the Wulff vector at a point on a surface is a vector that is dotted with the surface normal to give the energy density of the surface. For ordinary area, it is the unit normal itself. But if a finite set of Wulff vectors is given, and the one used at a point is the one with the maximum dot product, then the minimal surface has only flat faces, i.e. it is crystalline. The crystal shape in this example is an octahedron. The Wulff vectors are the vertices of the octahedron, and are in `octa.wlf`. The datafile `crystal.fe` describes a cube with extra edges around the equator. These edges provide places for octahedron edges to form. Things work a lot better if the facet edges are where the crystal edges want to be.

To see the cube turn into an octahedron, refine once, use the `m` command to set the scale to 0.05, and iterate about 50 times. Fixed scale works much better than optimizing scale for crystalline integrands, since crystalline energy is not a smooth function of the scale factor. Sometimes you have to put in jiggling also to get over local obstacles to minimizing energy.

Here is the Wulff vector file `octa.wlf`:

```
0      0      1
0      0     -1
0.707  0.707  0
-0.707  0.707  0
0.707 -0.707  0
-0.707 -0.707  0
```

Here is the datafile:

```
// crystal.fe
// Evolver data for cube of prescribed volume with
//   crystalline surface energy.    Evolves into an octahedron.

Wulff "octa.wlf"    // Wulff vectors for octahedral crystal

vertices
```

```
1  0.0 0.0 0.0
2  1.0 0.0 0.0
3  1.0 1.0 0.0
4  0.0 1.0 0.0
5  0.0 0.0 1.0
6  1.0 0.0 1.0
7  1.0 1.0 1.0
8  0.0 1.0 1.0
9  0.0 0.0 0.5
10 1.0 0.0 0.5
11 1.0 1.0 0.5
12 0.0 1.0 0.5

edges  /* given by endpoints and attribute */
1   1 2
2   2 3
3   3 4
4   4 1
5   5 6
6   6 7
7   7 8
8   8 5
9   1 9
10  2 10
11  3 11
12  4 12
13  9 5
14  10 6
15  11 7
16  12 8
17   9 10
18  10 11
19  11 12
20  12  9

faces  /* given by oriented edge loop */
1   1 10 -17  -9
2   2 11 -18 -10
3   3 12 -19 -11
4   4  9 -20 -12
5   5  6  7   8
6  -4 -3 -2  -1
7  17 14 -5 -13
8  18 15 -6 -14
9  19 16 -7 -15
10 20 13 -8 -16

bodies  /* one body, defined by its oriented faces */
1   1 2 3 4 5 6 7 8 9 10  volume 1
```

§3.11. Tutorial in Advanced Calculus.

Some Evolver users may be a bit rusty in advanced calculus, so this section states the basic theorems and shows how to apply them to find appropriate edge integrals for area and volume.

Notation: If S is an oriented surface or plane region, then ∂S is its oriented boundary, oriented counterclockwise with respect to the unit normal vector $\vec{N}$.

Green's Theorem: If S is a plane region and $\vec{w}(x,y) = M(x,y)\vec{i} + N(x,y)\vec{j}$ is a vectorfield, then

$$\int_{\partial S} \vec{w}\cdot\vec{dl} = \int\int_S \frac{\partial N}{\partial x} - \frac{\partial M}{\partial y} dx\, dy.$$

Green's Theorem is actually a special case of Stokes' Theorem:

Stokes' Theorem: If S is an oriented surface in R^3 and $\vec{w}(x,y,z) = w_x(x,y,z)\vec{i} + w_y(x,y,z)\vec{j} + w_z(x,y,z)\vec{k}$ is a vectorfield, then

$$\int_{\partial S} \vec{w}\cdot\vec{dl} = \int\int_S curl\, \vec{w}\cdot\vec{N}\, dA$$

where

$$curl\,\vec{w} = \nabla\times\vec{w} = \left(\left[\frac{\partial w_z}{\partial y} - \frac{\partial w_y}{\partial z}\right]\vec{i}, \left[\frac{\partial w_x}{\partial z} - \frac{\partial w_z}{\partial z}\right]\vec{j}, \left[\frac{\partial w_y}{\partial x} - \frac{\partial w_x}{\partial y}\right]\vec{k}\right).$$

Notation: If B is a region in R^3, then ∂B is its boundary surface, oriented with outward unit normal $\vec{N}$.

Divergence Theorem: If B is a region in R^3 and $\vec{w}$ is a vectorfield, then

$$\int\int\int_B div\,\vec{w}\, dx\, dy\, dz = \int\int_{\partial B} \vec{w}\cdot\vec{N}\, dA,$$

where

$$div\,\vec{w} = \nabla\cdot\vec{w} = \frac{\partial w_x}{\partial x} + \frac{\partial w_y}{\partial y} + \frac{\partial w_z}{\partial z}.$$

Theorem: $div\, curl\,\vec{w} = 0$.

Application to volume integrals.

Suppose B is a body we wish to calculate the volume of. The volume can be defined as the triple integral

$$V = \int\int\int_B 1\, dx\, dy\, dz.$$

Evolver can't do volume integrals, but it can do surface integrals, so Evolver uses the Divergence Theorem with $\vec{w} = z\vec{k}$, since $div\, z\vec{k} = 1$:

$$V = \int\int_{\partial B} z\vec{k}\cdot\vec{N}\, dA.$$

Taking an upward pointing normal to the surface, this can be written as

$$V = \int\int_{\partial B} z\, dx\, dy.$$

It is often inconvenient and unnecessary to integrate over all of ∂B. For example, any portion lying in the plane $z = 0$ may obviously be omitted, along with any portion that is vertical, i.e. has a horizontal tangent $\vec{N}$. Other parts of ∂B may be omitted if we specifically include compensating integrals. Suppose a portion S of ∂B is part of a constraint level set $F(x,y,z) = 0$. Then we want to use Stokes' Theorem to rewrite

$$\int\int_S z\vec{k}\cdot\vec{N}\, dA = \int_{\partial S} \vec{w}\cdot\vec{dl}$$

for some vectorfield $\vec{w}$ with $z\vec{k} = curl\,\vec{w}$. Unfortunately, $div\,z\vec{k} \neq 0$, so we can't do that. We have to get an equivalent form of $z\vec{k}$ valid on S. Clearly the equivalent form cannot contain z as such, so we have to solve the implicit function $F(x,y,z) = 0$ explicitly for $z = f(x,y)$. Then it remains to solve $f(x,y)\vec{k} = curl\,\vec{w}$ for $\vec{w}$. There are many possible solutions to this (all differing by the gradient of some scalar function), but one convenient solution is to take $w_x = M(x,y)$, $w_y = N(x,y)$, and $w_z = 0$, with

$$\frac{\partial N}{\partial x} - \frac{\partial M}{\partial y} = f(x,y).$$

There are still many solutions of this. One could take partial antiderivatives

$$M(x,y) = 0, \qquad N(x,y) = \int f(x,y)dx,$$

or

$$M(x,y) = -\int f(x,y)dy, \qquad N(x,y) = 0,$$

or some combination thereof. The choice is guided by the principle that $M\vec{i} + N\vec{j}$ should be perpendicular to the portions of ∂S that are not included in edge constraint integrals.

Often, the problem has rotational symmetry around the z axis. Then it is usually convenient to choose a vectorfield $\vec{w}$ with rotational symmetry, of the form

$$\vec{w} = g(t)(-y\vec{i} + x\vec{j}),$$

where $t = x^2 + y^2$. Using the Chain Rule on t, we get

$$\frac{\partial N}{\partial x} - \frac{\partial M}{\partial y} = g'(t) \cdot 2x \cdot x + g(t) + g'(t) \cdot 2y \cdot y + g(t) = 2g'(t)t + 2g(t) = 2(tg(t))'.$$

Writing $f(x,y) = h(t)$, the solution of

$$2(tg(t))' = h(t)$$

is

$$g(t) = \frac{1}{2t}\int h(t)dt.$$

Symbolic integration programs like Mathematica or Maple can be useful in doing this indefinite integral. The result needs to have t replaced with x^2+y^2, and the relation $F(x,y,z) = 0$ is often useful in simplifying the result. Great care may be needed to get the result into a form that is valid everywhere, particularly if there are square roots of ambiguous sign. Always check your results for reasonableness, and check any datafiles to make sure the initial values for volumes are what you expect.

Example: Hyperboloid constraint $x^2 + y^2 - z^2 = R^2$. Here the surface is a disk spanning the hyperboloid, and the body is the region between the $z = 0$ plane and the disk. The edge integral must compensate for the volume below the surface but outside the hyperboloid. We have $z = f(x,y) = (x^2+y^2-R^2)^{1/2}$, so $h(t) = (t-R^2)^{1/2}$, and integrating gives

$$g(t) = \frac{1}{2t}\frac{2}{3}(t-R^2)^{3/2}.$$

Hence

$$\vec{w} = \frac{(x^2+y^2-R^2)^{3/2}}{3(x^2+y^2)}(-y\vec{i} + x\vec{j}).$$

Using the hyperboloid equation, this simplifies to

$$\vec{w} = \frac{z^3}{3(x^2+y^2)}(-y\vec{i} + x\vec{j}).$$

Notice this form is smooth everywhere on the surface and zero when $z = 0$. When integrated over the disk boundary, it properly gives negative volumes for curves with negative z. In the datafile, this vector would go in the constraint integral with reversed signs, since we want to subtract the volume under the hyperboloid from the volume under the disk.

Application to area. Here we want to calculate the area on a constraint surface S bounded by a curve ∂S for the purpose of prescribing the contact angle of the curve. We want to use Stokes' Theorem to rewrite the area as a boundary line integral:

$$A = \int\int_S 1\,dA = \int\int_S \vec{v}\cdot\vec{N}\,dA = \int_{\partial S} curl\,\vec{w}\cdot\vec{dl}.$$

To this end, we need to find a vectorfield $\vec{v}$ with $\vec{v}\cdot\vec{N} = 1$ and $div\,\vec{v} = 0$. Again, one way to do this is to solve for the surface as the graph of $z = f(x,y)$ and use $\vec{v} = (1+f_x^2+f_y^2)^{1/2}\vec{k}$ to write the area as

$$\int\int_S (1+f_x^2+f_y^2)^{1/2}\,dx\,dy,$$

and use Green's Theorem. We need to solve

$$\frac{\partial N}{\partial x} - \frac{\partial M}{\partial y} = (1+f_x^2+f_y^2)^{1/2}.$$

The same remarks as in the volume case apply.

Example: Hyperboloid constraint $x^2+y^2-z^2 = R^2$. We have $z = f(x,y) = (x^2+y^2-R^2)^{1/2}$, so

$$(1+f_x^2+f_y^2)^{1/2} = \left[1+\frac{x^2+y^2}{x^2+y^2-R^2}\right]^{1/2}.$$

Using the rotationally symmetric approach as in the volume example, we have

$$h(t) = [1+t/(t-R^2)]^{1/2},$$

and

$$g(t) = \frac{1}{2t}\int [1+t/(t-R^2)]^{1/2}dt.$$

Mathematica gives the integral in the horrible form

$$\sqrt{\frac{R^2-2\,t}{R^2-t}}\,(-R^2+t) - \frac{R^2\,\log(-3\,R^2+2\,\sqrt{2}\,R^2\,\sqrt{\frac{R^2-2\,t}{R^2-t}}+4\,t-2\,\sqrt{2}\,\sqrt{\frac{R^2-2\,t}{R^2-t}}\,t)}{2\,\sqrt{2}}.$$

Taking advantage of the fact that $t \geq R^2$, this becomes

$$\sqrt{(2t-R^2)(t-R^2)} - \frac{R^2\,\log(-3\,R^2+2\,\sqrt{2}\,\sqrt{(2t-R^2)(t-R^2)}+4\,t)}{2\,\sqrt{2}}.$$

And using $t - R^2 = z^2$,

$$z\sqrt{2t-R^2} - \frac{R^2\,\log(-3\,R^2+2\,\sqrt{2}\,z\sqrt{2t-R^2})+4\,t)}{2\,\sqrt{2}}.$$

Adding a constant of integration to make this 0 at $z = 0$ gives

$$z\sqrt{2t-R^2} - \frac{R^2}{2\sqrt{2}}\left(\log\left(4t-3\,R^2+2\,\sqrt{2}\,z\sqrt{2t-R^2}\right)-\log(R^2)\right).$$

This is in fact an odd function of z. The final vectorfield is

$$\vec{w} = \left[\frac{2\sqrt{2}z\sqrt{2(x^2+y^2)-R^2} - R^2\left(\log\left(4(x^2+y^2)-3\,R^2+2\,\sqrt{2}\,z\sqrt{2(x^2+y^2)-R^2}\right)-\log(R^2)\right)}{4\sqrt{2}(x^2+y^2)}\right](-y\vec{i}+x\vec{j}).$$

Chapter 4. The Model.

This chapter describes the mathematical model of a surface that the Surface Evolver incorporates. See the **Datafile** chapter for how to set up the initial surface and the **Operations** chapter for how to manipulate the surface. Keywords are often given in upper case in this manual, even though case is not significant. In the Index, keywords are in typewriter font.

§4.1. Dimension of surface.

In the standard Evolver surface, the surface tension resides in the two-dimensional facets (this is referred to as the **soapfilm model**.). However, it is possible to have the tension reside in the edges. This is referred to as the **string model** and is declared by the keyword `STRING` in the datafile. See `slidestr.fe` for an example. It is possible in the soapfilm model to also have edges with tension in order to represent a situation where singular curves have energy. This chapter is written mainly for the soapfilm model. For differences in the string model, see the **String Model** section below. For a surface of more than two dimensions, see **Simplex model**.

§4.2. Geometric elements.

The surface is defined in terms of its geometric elements of each dimension. Each element has its own set of attributes. Some may be set by the user; others are set internally but may be queried by the user. It is also possible to dynamically define "extra attributes" for any type of element, which may be single values or vectors of values.

§4.2.1. Vertices.

A vertex is a point in space. The coordinates of the vertices are the parameters that determine the location of the surface. It is the coordinates that are changed when the surface evolves.

Attributes:

coordinates - Euclidean coordinates.

boundary - Vertex position is defined parametrically on a boundary. A vertex can be on at most one boundary.

boundary parameters - For a vertex on a boundary.

fixed - Vertex cannot be moved or eliminated.

constraints - Vertex may be subject to a set of level set constraints.

id - An identification number.

original - Number of the vertex in the original datafile, -1 if subsequently created.

bare - Vertex does not have an adjacent edge (string model) or adjacent facet (soapfilm model). Useful for avoiding warning messages.

valence - Number of adjacent edges.

axial_point - With a symmetry group, indicates that the vertex is a fixed point of the group, i.e. on an axis of rotation.

triple_point - For telling Evolver three films meet at this vertex. Used when `effective_area` is on to adjust motion of vertex by making the effective area around the vertex $1/\sqrt{3}$ of the actual area.

tetra_point - For telling Evolver six films meet at this vertex. Used when `effective_area` is on to adjust motion of vertex by making the effective area around the vertex $1/\sqrt{6}$ of the actual area.

vertexnormal[n] - Components of a normal vector.

extra attributes - User-defined values.

§4.2.2. Edges.

An edge is an oriented line segment between a tail vertex and a head vertex, in the linear model. In the quadratic model, an edge has a midpoint. In the Lagrange model, an edge has $lagrange\_order + 1$

Attributes:

vertices - Vertices of the edge.

midv - In the quadratic model, the midpoint of the edge.

length - Actual length of the edge.

fixed - All vertices generated from subdividing edge will be fixed. Edges fixed in the datafile will have their endpoints fixed.

constraints - Set of level set constraints edge satisfies. Constraints may be used to specify volume and energy integrands.

boundary - Edge may be in one parameterized boundary.

no_refine - Edge is not to be refined by the 'r' command.

torus wrapping - Whether edge crosses torus unit cell faces, and in which direction.

density or tension - Energy per unit length in string model.

color - For display.

id - An identification number.

oid - Signed identification number, for orientation.

original - Number of the original datafile edge this edge is descended from, -1 if not descended.

orientation - For the sign of oriented integrals on this edge. Value +1 or -1.

x,y,z - Components of vector from tail to head.

bare - Edge does not have adjacent facets. Useful for avoiding warning messages.

frontbody - (String model) The id of the body of which the edge is on the positively oriented boundary.

backbody - (String model) The id of the body of which the edge is on the negatively oriented boundary.

extra attributes - User-defined quantities.

§4.2.3. Facets.

A facet is an oriented polygon defined by its edges. In the soapfilm model, it is always a triangle. In the string model, it may have any number of sides. In the Lagrange soapfilm model, it will have interior vertices if lagrange_order exceeds 2.

Attributes:

edges - Loop of edges connected head to tail.

area - Actual area of the facet.

fixed - Fixedness inherited by all elements descended from this facet.

constraints - Set of level set constraints facet satisfies.

boundary - Facet may be in one boundary.

density - Energy per unit area; surface tension.

id - An identification number.

oid - Signed identification number, for orientation.

orientation - For the sign of oriented integrals on this facet. Value +1 or -1.

original - Number of the original datafile facet this facet is descended from, -1 if not descended.

no_refine - Edges generated in the interior of the facet will inherit the no_refine attribute. But otherwise, this attribute has no effect on whether the facet will be refined by the 'r' command.

color - For display.

frontcolor, backcolor - For coloring the two sides differently.

phase - For determining boundary tension in string model.

orientation - For the sign of oriented integrals on this facet. Value +1 or -1.

bodies - Ids of the two bodies this facet forms positive or negative boundary with, respectively.

frontbody - The id of the body of which the facet is on the positively oriented boundary.

backbody - The id of the body of which the facet is on the negatively oriented boundary.

valence - Number of edges around facet.

x,y,z - components of normal vector.

extra attributes - User-defined quantities.

§4.2.4. Bodies.

A body is a three-dimensional region of space in the soapfilm model, and a two-dimensional region in the string model.

Attributes:

facets - A set of oriented facets defines the boundary of body. These facets are used for calculating body volume and gravitational energy. Only those facets needed for correct calculation need be given. In the string model, usually a body corresponds to one facet, and hte "volume" of the body is the area of the facet.

id - An identification number.

density - For gravitational potential energy.

volume - Actual volume.

target volume - Constrained volume value set by user.

volconst - An adjustment used in the torus model to do correct volume calculations. The value is added to whatever volume is otherwise calculated.

pressure - For prescribed pressure, or for pressure resulting from volume constraint.

phase - For determining boundary tension in soapfilm model.

volfixed - Read-only attribute, 1 if the volume of the body is fixed, 0 if not.

extra attributes - User-defined quantities.

§4.2.5. Facetedges.

A facetedge is an internal data structure used by the Evolver to hold connectivity information. The user need not be concerned about facetedges in the ordinary course of events.

Attributes:

id - An identification number.

edge - Generates the edge the facetedge is associated with.

facet - Generates the facet the facetedge is associated with.

extra attributes - User-defined quantities.

§4.3. Quadratic model.

By default, edges and facets are linear. It is possible to represent edges as quadratic curves and facets as quadratic patches by putting the `QUADRATIC` keyword in the datafile or by using the `M` command. If this is done, then each edge is endowed with a midpoint vertex. Some basic features are implemented for the quadratic representation, such as area, volume, constraints, and basic named quantity methods.

§4.4. Lagrange model.

For higher accuracy than the linear or quadratic models provide, there is a very limited implementation of higher order elements. In the Lagrange model of order n, each edge is defined by Lagrange interpolation on $n+1$ vertices evenly spaced in the parameter domain, and each facet is defined by interpolation on $(n+1)(n+2)/2$ vertices evenly spaced in a triangular pattern in the parameter domain. That is, the elements are Lagrange elements in the terminology of finite element analysis. The Lagrange model may be invoked with the command "lagrange n".

The Lagrange model is limited to certain named quantities and methods, so the Evolver must be started with the `-q` option, or you must give the `convert_to_quantities` command. No triangulation manipulations are available: no refining, equiangulation, or anything. Use the linear or quadratic model to establish your final triangulation, and just use the Lagrange model to get extra precision.

§4.5. Simplex model.

This model represents a surface solely with vertices and facets. It was put in to enable representation of arbitrary dimension surfaces, but many Evolver features are not available with it. Here each facet is represented as an oriented list of $k+1$ vertices, where k is the dimension of the surface. Edges may be specified as $k-1$ dimensional simplices, but they are used only to compute constraint integrals; a complete list of edges is not needed. Bodies still exist for hypersurfaces.

The datafile must have the keyword `SIMPLEX_REPRESENTATION` in the first section, and the phrase `SURFACE_DIMENSION` k if $k \neq 2$. If the domain is not 3-dimensional, then `SPACE_DIMENSION` n must also be included. The `EDGES` section is optional. Each facet should list $k+1$ vertex numbers. Non-simplicial facets are not allowed. See the sample datafile `simplex3.fe`.

Most features are not implemented. Vertices may be `FIXED`. Constraints are allowed, but no integrands of any kind. No `TORUS` domain. No `QUADRATIC` representation. No surface changing except iteration and refining. Refining subdivides each simplex edge, with the edge midpoint inheriting the common attributes of the edge endpoints. Refining will increase the number of facets by a factor of 2^k.

§4.6. Dimension of ambient space.

By default, surfaces live in 3 dimensional space. However, the phrase `SPACE_DIMENSION` n in the datafile sets the dimension to n. This means that all coordinates and vectors have n components. The only restriction is that Evolver has to be compiled with the `MAXCOORD` macro defined to be at least n in Makefile or in `model.h`. Change `MAXCOORD` and recompile if you want more than four dimensions.

Graphics will display only the first three dimensions of spaces with more than three dimensions, except for geomview, which has a four-dimensional viewer built in (although its use is awkward now).

§4.7. Riemannian metric.

The ambient space can be endowed with a Riemannian metric by putting the keyword `METRIC` or `CONFORMAL_METRIC` in the datafile followed by the elements of the metric tensor. A conformal metric tensor is a multiple of the identity matrix, so only one element is needed. Only one coordinate patch is allowed, but the quotient space feature makes this quite flexible. Edges and facets are linear in coordinates, they are not geodesic. The metric is used solely to calculate lengths and areas. It is not used for volume. To get a volume constraint on a body, you will have to define your own "quantity" constraint. See `quadm.fe` for an example of a metric.

One special metric is available built-in. It is the Klein model of hyperbolic space in n dimensions. The domain is the unit disk or sphere in Euclidean coordinates. Including the keyword `KLEIN_METRIC` in the top section of the datafile will invoke this metric. Lengths and areas are calculated exactly, but as with other metrics you are on your own for volumes and quantities.

§4.8. Torus domain.

By default, the domain of a surface is Euclidean space R^3. However, there are many interesting problems dealing with periodic surfaces. The Evolver can take as its domain a flat torus with an arbitrary parallelpiped as its unit cell, i.e. the domain is a parallelpiped with its opposite faces identified. This is indicated by the `TORUS` keyword in the datafile. The defining basis vectors of the parallelpiped are given in the `TORUS_PERIODS` entry of the datafile. See `twointor.fe` for an example.

Certain features are not available with the torus domain, namely quadratic representation, constraints, boundaries, quantities, and gravity. (Actually, you could put them in, but they will not take into account the torus wrapping.) However, there are new named quantity methods that will do quadratic model torus domain.

Volumes and volume constraints are available. However, if the torus is completely partitioned into bodies of prescribed volume, then the volumes must add up to the volume of the unit cell and the `TORUS_FILLED` keyword must be given in the datafile. Or just don't prescribe the volume of one body.

Volumes are somewhat ambiguous. The volume calculation method is accurate only to one torus volume, so it is possible that a body whose volume is positive gets its volume calculated as negative. Evolver adjusts volumes after changes to be as continuous as possible with the previous volumes as possible, or with target volumes when available. You can also set a body's volconst attribute if you don't like the Evolver's actions.

Level set constraints can be used in the torus model, but be cautious when using them as mirror symmetry planes with volumes. The torus volume algorithm does not cope well with such partial surfaces. If you must, then use y=const symmetry planes rather than x=const, and use the -q option or do `convert_to_quantities`. Double-check that your volumes are turning out correctly; use volconst if necessary.

Vertex coordinates are given as Euclidean coordinates within the unit cell, not as linear combinations of the basis vectors. The coordinates need not lie within the parallelpiped, as the exact shape of the unit cell is somewhat arbitrary. The way the surface wraps around in the torus is given by saying how the edges cross the faces of the unit cell. In the datafile, each edge has one symbol per dimension indicating how the edge vector crosses each identified pair of faces, and how the vector between the endpoints needs to be adjusted to get the true edge vector:

`*` does not cross face
`+` crosses in same direction as basis vector, so basis vector added to edge vector
`-` crosses in opposite direction, so basis vector subtracted from edge vector.

There are several commands for ways of displaying a torus surface:

`raw_cells` - Graph the facets as they are. Good for checking your datafile if it's all triangles. Faces that get subdivided may be unrecognizable.

`connected` - Each body's facets are unwrapped in the torus, so the body appears in one connected piece. Nicest option.

`clipped` - Shows the unit cell specified in the datafile. Facets are clipped on the parallelpiped faces.

§4.9. Quotient spaces and general symmetry.

As a generalization of the torus domain, you may declare the domain to be the quotient space of R^n with respect to some symmetry group. This cannot be completely specified in the datafile, but requires you to write some C routines to define group operations. Group elements are represented by integers attached to edges (like the wrap specifications in the torus model). You define the integer representation of the group elements. See the files `quotient.c` and `symtest.fe` for an example. See `khyp.c` and `khyp.fe` for a more intricate example modelling an octagon in Klein hyperbolic space identified into a genus 2 surface. The datafile requires the keyword `SYMMETRY_GROUP` followed by your name for the group in quotes. Edges that wrap have their group element specified in the datafile by the phrase "`wrap` n", where n is the number of the group element. The wrap values are accessible at run time with the `wrap` attribute of edges. The group operations are accessible by the functions `wrap_inverse(`w`)` and `wrap_compose(`$w1$`,`$w2$`)`.

Using any Hessian commands with any symmetry group other than the built-in torus model requires converting to all named quantities with either the `-q` startup option or the `convert_to_quantities` command.

Volumes of bodies may not be calculated correctly with a symmetry group. The volume calculation only knows about the built-in torus model. For other symmetry groups, if you declare a body, it will use the Euclidean volume calculation. It is up to you to design an alternate volume calculation using named quantities and methods.

Currently implemented symmetries:

§4.9.1. TORUS symmetry group

This is the underlying symmetry for the torus model. Although the torus model has a number of special features built in to the Evolver, it can also be accessed through the general symmetry group interface. The torus group is the group on n-dimensional Euclidean space generated by n independent vectors, called the period vectors. The torus group uses the torus periods listed in the datafile top section.

Datafile declaration:

```
symmetry_group "torus"
periods
2 0 0
0 3 0
0 0 3
```

Group element encoding: The 32-bit code word is divided into 6-bit fields, one field for the wrap in each dimension, with low bits for the first dimension. Hence the maximum space dimension is 5. Within each bitfield, 1 codes for positive wrap and 011111 codes for negative wrap. The coding is actually a 2's complement 5-bit integer,so higher wraps could be represented.

§4.9.2. ROTATE symmetry group

This is the cyclic symmetry group of rotations in the x-y plane, where the order of the group is given by the internal variable `rotation_order` **Note:** Since this group has fixed points, some special precautions are necessary. Vertices on the rotation axis must be labelled with the attribute `axial_point` in the datafile. Edges out of an axial point must have the axial point at their tail, and must have zero wrap. Facets including an axial point must have the axial point at the tail of the first edge in the facet.

Sample datafile declaration:

```
symmetry_group "rotate"
parameter rotate_order = 6
```

Group element encoding: An element is encoded as the power of the group generator.

§4.9.3. FLIP ROTATE symmetry group

This is the cyclic symmetry group of rotations in the x-y plane with a flip z -¿ -z on every odd rotation, where the order of the group is given by the internal variable `rotation_order`, which had better be even. **Note:** Since this group has points that are fixed under an even number of rotations, some special precautions are necessary. Vertices on the rotation axis must be labelled with the attribute `double_axial_point` in the datafile. Edges out of an axial point must have the axial point at their tail, and must have zero wrap. Facets including an axial point must have the axial point at the tail of the first edge in the facet.

Sample datafile declaration:

```
symmetry_group "flip_rotate"
parameter rotate_order = 6
```

Group element encoding: An element is encoded as the power of the group generator.

§4.9.4. CUBOCTA symmetry group

This is the full symmetry group of the cube. It can be viewed as all permutations and sign changes of (x,y,z).

Datafile declaration:

```
symmetry_group "cubocta"
```

Group element encoding: The low bit of wrap denotes reflection in x; second lowest bit reflection in y; third lowest bit reflection in z; next two bits form the power of the (xyz) permutation cycle; next bit tells whether to then swap x,y. (By John Sullivan; source in quotient.c under name pgcube)

§4.9.5. XYZ symmetry group

The orientation-preserving subgroup of cubocta. Included temporarily since it is referred to in a paper. See cubocta for details.

§4.9.6. GENUS2 symmetry group

This is a symmetry group on the Klein model of hyperbolic space whose quotient group is a genus 2 hyperbolic manifold. The fundamental region is an octagon.

Datafile declaration:

```
symmetry_group "genus2"
```

Group element encoding: There are 8 translation elements that translate the fundamental region to one of its neighbors. Translating around a vertex gives a circular string of the 8 elements. The group elements encoded are substrings of the 8, with null string being the identity. Encoding is 4 bits for start element, 4 bits for stop (actually the one after stop so 0 0 is identity). See `khyp.c` for more.

§4.9.7. DODECAHEDRON symmetry group

This is the symmetry group of translations of hyperbolic 3 space tiled with right-angled dodecahedra. The elements of the group are represented as integers. There are 32 generators of the group so each generator is represented by five bits. Under this scheme any element that is the composition of up to five generators can be represented. If you want to use this group, you'll have to check out the source code in dodecgroup.c, since somebody else wrote this group and I don't feel like figuring it all out right now.

Datafile declaration:

```
Klein_metric
symmetry_group "dodecahedron"
```

§4.9.8. CENTRAL SYMMETRY symmetry group

This is the order 2 symmetry group of inversion through the origin, X -¿ -X.

Datafile declaration:

```
symmetry_group "central_symmetry"
```

Group element encoding: 0 for identity, 1 for inversion.

§4.9.9. SCREW SYMMETRY symmetry group

This is the symmetry group of screw motions along the z axis.The global parameter `screw_height` is the translation distance (default 1), and the global parameter `screw_angle` is the rotation angle in degrees (default 0).

Sample datafile declaration:

```
parameter screw_height = 4.0
parameter screw_angle  = 180.0
symmetry_group "screw_symmetry"
```

Group element encoding: The integer value is the power of the group generator.

§4.10. Symmetric surfaces.

Symmetric surfaces can often be done more easily by evolving just one fundamental domain. This is easiest if the fundamental region is bounded by planes of mirror symmetry. These planes become constraint surfaces for the fundamental domain (see Constraints below). Multiple images of the fundamental domain may be displayed by using transformation matrices (see `view_transforms` in the Datafile chapter).

§4.11. Level set constraints.

A constraint is a restriction on the motion of vertices. It may be represented as a level set of a function or as a parametric manifold. The term "constraint" as used in the Evolver refers to the level set formulation, and "boundary" refers to the parametric formulation. The term "volume constraint" is something entirely else, refering to bodies.

A level set constraint may have several roles:

1. Vertices may be required to lie on a constraint (equality constraint) or on one side (inequality constraint). A constraint may be declared `GLOBAL`, in which case it applies to all vertices. See `mound.fe` for an example. The formula defining the function may contain attributes of vertices, so in particular vertex extra attributes may be used to customize one formula to individual vertices.

2. A constraint may have a vectorfield associated with it that is integrated over edges lying in the constraint to give an energy. This is useful for specifying wall contact angles and for calculating gravitational energy. Integrals are not evaluated over edges that are FIXED. See `mound.fe` for an example.

3. A constraint may be declared `CONVEX`, in which case edges in the constraint have an energy associated with them that is proportional to the area between the straight edge and the curved wall. This energy (referred to as "gap energy") is meant to compensate for the tendency for flat facets meeting a curved wall to minimize their area by lengthening some edges on the wall and shortening others, with the net effect of increasing the net gap between the edges and the wall. See `tankex.fe` for an example.

4. A constraint may have a vectorfield associated with it that is integrated over edges lying in the constraint to give a volume contribution to a body whose boundary facets contain the edges. This is useful for getting correct volumes for bodies without completely surrounding them with otherwise useless facets. It is important to understand how the content is added to the body in order to get the signs right. The integral is evaluated along the positive direction of the edge. If the edge is positively oriented on a facet, and the facet is positively oriented on a body, then the integral is added to the body. This may wind up giving the opposite sign to the integrand from what you think may be natural. Integrals are not evaluated over edges that are FIXED. See `tankex.fe` for an example.

5. A constraint may have a vectorfield associated with it that is integrated as a surface integral over facets lying in the constraint to give a contribution to the total energy. This is useful for having gravity in an arbitrary direction. Integrals are not evaluated over facets that are FIXED. See `tankex.fe` for an example.

Edge integrands are evaluated by Gaussian quadrature. The number of points used is controlled by the `INTEGRAL_ORDER` option in the datafile.

§4.12. Boundaries.

Boundaries are parameterized submanifolds. Vertices, edges, and facets may be deemed to lie in a boundary. For a vertex, this means that the fundamental parameters of the vertex are the parameters of the boundary, and its coordinates are calculated from these. Vertices on boundaries may move during iteration, unless declared fixed. See `cat.fe` for an example. Edges on boundaries have energy and content integrals in the same manner as constraint edges, but they are internally implemented as named quantities.

A delicate point is how to handle wrap-arounds on a boundary such as a circle or cylinder. Subdividing a boundary edge requires a midpoint, but taking the average parameters of the endpoints can give nonsense. Therefore the average coordinates are calculated, and that point projected on the boundary as continued from one endpoint.

A general guideline is to use constraints for two-dimensional walls and boundaries for one-dimensional wires. If you are using a boundary wire, you can probably declare the vertices and edges on the boundary to be `FIXED`. Then the boundary becomes just a guide for refining the boundary edges.

NOTE: A vertex on a boundary cannot also have constraints.

§4.13. Energy.

The Evolver works by minimizing the total energy of the surface. This energy can have several components:

1. Surface tension. Soap films and interfaces between different fluids have an energy content proportional to their area. Hence they shrink to minimize energy. The energy per unit area can also be regarded as a surface tension, or force per unit length. Each facet has a surface tension, which is 1 unless the datafile specifies otherwise (see `TENSION` option for faces). Different facets may have different surface tensions. Facet tensions may be changed interactively with the '`set facet tension ...`' command. The contribution to the total energy is the sum of all the facet areas times their respective surface tensions. The surface tension of a facet may also be specified as depending on the phases of the bodies it separates. See `PHASE`.

2. Gravitational potential energy. If a body has a density (see `DENSITY` option for bodies), then that body contributes its gravitational energy to the total. The acceleration of gravity G is under user control. Letting ρ be the body density, the energy is defined as

$$E = \int\int\int_{body} G\rho z dV \tag{1}$$

but is calculated using the Divergence Theorem as

$$E = \int\int_{body\ surface} G\rho\frac{z^2}{2}\vec{k}\cdot\vec{dS}. \tag{2}$$

This integral is done over each facet that bounds a body. If a facet bounds two bodies of different density, then the appropriate difference in density is used. Facets lying in the $z = 0$ plane make no contribution, and may be omitted if they are otherwise unneeded. Facets lying in constraints may be omitted if their contributions to the gravitational energy are contained in constraint energy integrals.

3. Named quantities. See the section on named quantities below.

4. Constraint edge integrals. An edge on a constraint may have an energy given by integrating a vectorfield $\vec{F}$ over the oriented edge:

$$E = \int_{edge} \vec{F}\cdot\vec{dl}. \tag{3}$$

The integral uses the innate orientation of the edge, but if the orientation attribute of the edge is negative, the value is negated. This is useful for prescribed contact angles on walls (in place of wall facets with equivalent tension) and for gravitational potential energy that would otherwise require facets in the constraint.

6. Convex constraints. Consider a soap film spanning a circular cylinder. The Evolver must approximate this surface with a collection of facets. The straight edges of these facets cannot conform to the curved wall, and hence the computed area of the surface leaves out the gaps between the outer edges and the wall. The Evolver will naturally try to minimize area by moving the outer vertices around so the gaps increase, ultimately resulting in a surface collapsed to a line. This is not good. Therefore there is provision for a "gap energy" to discourage this. A constraint may be declared `CONVEX` in the datafile. For an edge on such a constraint, an energy is calculated as

$$E = k \left\| \vec{S} \times \vec{Q} \right\| /6 \tag{4}$$

where $\vec{S}$ is the edge vector and $\vec{Q}$ is the projection of the edge on the tangent plane of the constraint at the tail vertex of the edge. The constant k is a global constant called the "gap constant". A gap constant of 1 gives the best approximation to the actual area of the gap. A larger value minimizes gaps and gets vertices nicely spread out along a constraint.

The gap energy falls off quadratically as the surface is refined. That is, refining once reduces the gap energy by a factor of four. You can see if this energy has a significant effect on the surface by changing the value of k with the `k` command.

7. Compressible bodies. If the ideal gas mode is in effect (see `PRESSURE` keyword for datafile), then each body contributes an energy

$$E = P_{amb} V_0 \ln(V/V_0) \tag{5}$$

where P_{amb} is the ambient pressure, V_0 is the volume prescribed in the datafile, and V is the actual volume of the body. To account for work done against the ambient pressure, each body also makes a negative contribution of

$$E = -P_{amb} V. \tag{6}$$

8. Prescribed pressure. Each body with a prescribed pressure P contributes energy

$$E = PV. \tag{7}$$

where V is the actual volume of the body. This can be used to generate surfaces of prescribed mean curvature, since mean curvature is proportional to pressure.

9. Squared mean curvature. An average mean curvature around each vertex can be calculated, and the integral of the square of this average counted as energy. This component is included by the `SQUARE_CURVATURE` phrase in the datafile. A curvature offset h_0 (so the energy is $(h - h_0)^2$) can be specified with `H_ZERO value` in the datafile. The weighting factor can be changed with the `A` command by changing the value of the adjustable constant `square curvature modulus`. For details of exactly how the integral of squared curvature is calculated, see the Technical Reference section. (This feature is not implemented for the quadratic model.) Several variants of squared mean curvature are available, toggled by `effective_area` and `normal_curvature` commands. See the technical reference chapter for details of the formulas. [HKS] reports some experimental results.

10. Mean curvature integral. The energy can include the integral of the signed mean curvature by including the `MEAN_CURVATURE_INTEGRAL` keyword with weighting factor in the datafile.

11. Squared Gaussian curvature. The Gauss map of a surface maps the surface to a unit sphere by mapping a point to its normal. The Gaussian curvature at a point is the Jacobian of this map. This definition can be extended to piecewise linear surfaces. The Evolver can include the integral of the squared Gaussian curvature in the energy by including the `SQUARE_GAUSSIAN_CURVATURE` keyword with weighting factor in the datafile.

12. Crystalline integrands. The Evolver can model energies of crystalline surfaces. These energies are proportional to the area of a facet, but they also depend on the direction of the normal. The energy is given by the largest dot product of the surface normal with a set of vectors known as the **Wulff vectors**. Surface area can be regarded as a crystalline integrand whose Wulff vectors are the unit sphere. See the datafile section on Wulff vectors for more. A surface has either crystalline energy or surface tension, not both. Use is not recommended since nonsmoothness makes Evolver work poorly.

§4.14. Named quantities and methods.

These are an effort to provide a more systematic way of adding new energies and constraints.

A "method" is a way of calculating a scalar value from some particular type of element (vertex, edge, facet, body). Each method is implemented internally as a set of functions for calculating the value and its gradient as a function of vertex positions. Methods are referred to by their names. See the Named Methods and Quantities chapter for the full list available, and the specifications each needs. Many hessians are available for named quantities that are not available for their old-fashioned counterparts. The command line option `-q` will automatically convert everything to named quantities, although some things cannot be converted yet. The Evolver command `convert_to_quantities` will accomplish the same effect interactively.

Adding a new method involves writing C routines to calculate the value and the gradient as a function of vertex coordinates, and adding a structure to the method name array in `quantity.c`. All the other syntax for invoking it from the datafile is already in place.

A "method instance" is a particular use of a method, with whatever particular parameters may be needed. A method instance may be created explicitly and given a name, or they may be created implicitly by defining quantities to use methods. See the Datafile chapter for details. Every method instance has a "modulus", which is multiplied times the basic method value to give the instance value. A modulus of 0 causes the entire instance calculation to be omitted.

A "quantity" is the sum total of various method instances although usually just one instance is involved. Any quantity may be declared to be one of three types: 1) "energy" quantities which are added to the overall energy of the surface; 2) "fixed" quantities that are constrained to a fixed target value (by Newton steps at each iteration); and 3) "info_only" quantities whose values are merely reported to the user. Each quantity has a "modulus", which is just a scalar multiplier of the sum of all instance values. A modulus of 0 will turn off calculation of all the instances.

The sample datafile `knotty.fe` contains some examples. Quantity values can be seen with the `v` or `A` command, and the `A` command can be used to change the target value of a fixed quantity, the modulus of a quantity, or some parameters associated to various methods.

It is planned that eventually all energies and global constraints will be converted to this system. However, existing syntax will remain valid wherever possible. Starting Evolver with the `-q` option will do this conversion now.

§4.15. Pressure.

Pressure is force per unit area perpendicular to a surface. In the Evolver, pressure can arise in three ways:

1. If a body has a volume constraint, then the boundary surface is unlikely to be a minimal surface. Hence pressure is needed to counteract the desire of the surface to shrink. When there are volume constraints, the Evolver automatically calculates the pressure needed. The value of the pressure can be seen with the `v` command.

2. A body may have a prescribed pressure. Then the appropriate force is added to the forces on the vertices when calculating the motion of the surface. This is the way to get a surface of prescribed mean curvature, since pressure = surface tension × mean curvature. NOTE: Prescribed volume and prescribed pressure on the same body are illegal. CAUTION: Prescribed pressure can make a surface expand to infinity if the pressure is too large.

3. The Evolver can treat bodies as being made up of an isothermal ideal gas, that is, bodies can be compressible. This happens if the `PRESSURE` keyword comes in the first part of the datafile. The pressure given there is the ambient pressure outside all bodies. Each body must have a volume specified, which is the volume of the body at the ambient pressure.

§4.16. Volume or content.

The terms "volume" and "content" are used for the highest dimensional measure of a region in n-space: area in R^2, volume in R^3, etc. Bodies may have a volume specified in the datafile, which then becomes a volume constraint. The volume of a body can be written as

$$V = \int\int\int_{body} 1dV, \tag{1}$$

which by the Divergence Theorem can be written a surface integral:

$$V = \int\int_{body\ surface} z\vec{k}\cdot d\vec{S}. \tag{2}$$

This integral is evaluated over all the boundary facets of a body.

The part of the boundary of a body lying on a constraint need not be given as facets. In that case, Stokes' Theorem can be used to convert the part of the surface integral on the constraint to a line integral over the edges where the body surface meets the constraint. The line integral integrands are given as constraint content integrands in the datafile.

An alternate surface integral is

$$V = \frac{1}{3}\int\int_{body\ surface} (x\vec{i} + y\vec{j} + z\vec{k})\cdot d\vec{S}. \tag{3}$$

This is used if `SYMMETRIC_CONTENT` is specified in the datafile. It is useful if the constraint content integral (which is evaluated by an approximation) leads to asymmetric results on what should be a symmetric surface.

As with surface area, the gap between flat facets and curved constraints causes problems. You can use constraint content integrals to overcome this. See the tank and sphere examples in the tutorial section.

§4.17. Diffusion.

The Evolver can simulate the real-life phenomenon of gas diffusion between neighboring bubbles. This diffusion is driven by the pressure difference across a surface. This is invoked by the keyword `DIFFUSION` in the first part of the datafile, followed by the value of the diffusion constant. The amount diffused across a facet during an iteration is calculated as

$$dm = (scale\ factor)(diffusion\ constant)(facet\ area)(pressure\ difference). \tag{1}$$

The scale factor is included as the time step of an iteration. The amount is added to or subtracted from the prescribed volumes of the bodies on either side of the facet.

§4.18. Motion.

The heart of the Evolver is the iteration step that reduces energy while obeying any constraints. The surface is changed by moving the vertices. No changes in topology or triangulation are made (unless certain options are toggled on). The idea is to calculate the velocity at each vertex and move the vertex in that direction. If the conjugate gradient option is in effect, the direction of motion is adjusted accordingly.

The force is the negative gradient of the energy. At each vertex, the gradient of the total energy (see the **Energy** section above) as a function of the position of that vertex is calculated. By default, the velocity is taken to be equal to the force. If the area normalization option is in effect, then the force is divided by 1/3 the area of the neighboring vertices to get an approximation to the mean curvature vector for velocity. If relevant, the gradients of quantity constraints (such as body volumes), and level set constraints are calculated at each vertex. Forces and quantity constraint gradients are projected onto the tangent space of the relevant level set constraints at each vertex. To enforce quantity constraints, a restoring motion along the volume gradients is calculated (there is found a constant for each quantity such that when each vertex is moved by the sum of $(constant)(quantity\ gradient)$ for all quantities including that vertex, then the quantity come out right, at least in a linear approximation). Then multiples of the quantity gradients are added to the forces to keep quantities constant (for volumes, the multiplying factors are the pressures of the bodies, which are actually found as the Lagrange multipliers for the volume constraints).

The actual motion of a vertex is the quantity restoring motion plus a universal scale factor times the velocity. The scale factor can be fixed or it can be optimized (see command `m`). If optimized, then the Evolver calculates the energy for several values of the scale (doubling or halving each time) until the minimum energy is passed. The optimum scale is then calculated by quadratic interpolation. One can also move by a multiple of the optimum scale by assigning a value to the variable `scale_scale`. A Runge-Kutta step is also available.

The unfixed vertices are then moved. If jiggling is in effect, each non-fixed vertex is moved in a random direction from a Gaussian distribution with standard deviation = (temperature)(characteristic length), where the characteristic length starts out as 1 and is halved at each refinement.

Finally, all level set constraints are enforced, including those on fixed vertices. Vertices violating an inequality or equality constraint are projected to the constraint (Newton's method). Several projection steps may be needed, until the violation is less that a certain tolerance or a certain number of steps are done (which generates a warning message). The default constraint tolerance is 1e-12, but it can be set with the `CONSTRAINT_TOLERANCE` option in the datafile, or setting the `constraint_tolerance` variable.

§4.19. Hessian.

There is another motion available in restricted circumstances that tries to get to the minimum in one step by calculating the second derivative matrix (Hessian) for energy, and then solving for the minimum energy. The command to do one iteration is "`hessian`". There are some restrictions on its use. The energies it applies to are area, constraint edge integrals, simplex model facet areas, and the named quantity methods so labelled earlier. Note particularly it does not apply to the string model or to ordinary gravity. One can fake strings in the soapfilm model with edge_length quantity, and use gravity_method instead of ordinary gravity. The constraints that are handled are body volumes, level set constraints, and fixed named quantities involving the afore mentioned methods. Unfixed vertices on parameterized boundaries are not handled. Some of these restrictions will be removed in future versions. If Evolver complains that it cannot do Hessian on your surface, try restarting Evolver with the `-q` option, which converts everything to named quantities if it can.

The Hessian iteration should be tried only when the surface is extremely close to a minimum (or some critical point). I advise evolving with other methods (like conjugate gradient) until the energy has converged to 8 places or so. If you do try the Hessian from too far away, it is likely to explode your surface. But if you do "`check_increase ON`", the surface will be restored to its previous state if the Hessian iteration would increase its energy.

The needed closeness to a minimum is largely due to motions tangential to the surface as the triangulation rearranges itself. There is a toggle `hessian_normal` that constrains motion to be along the surface normal (volume gradient, to be precise). Points without a well-defined normal (such as along triple junctions or at tetrahedral points) are not constrained.

Running the Hessian method will produce warning messages when the matrix is not positive definite. If the constraints don't make the net Hessian positive semidefinite, then you will get a message like

```
WARNING: Constrained Hessian not positive definite.  Index 3
```

where the index is the number of negative eigenvalues. This may show up in the first few iterations. Don't worry about it unless it shows up when the Hessian method has converged all the way, or causes the surface to blow up. If you are worried about blowing up, give the command `check_increase`. You can check positive definiteness without moving with commands mentioned in the next subsection.

The criterion for treating a value as zero in solving the Hessian is set by the variable "hessian_epsilon". Its default value is 1e-8.

To get a feel for the Hessian method, try it out with cat.fe.

Some experimental stuff using the Hessian matrix can be accessed with the command `hessian_menu`. This brings up a menu of various things. Some of the more interesting: choice E will find the lowest eigenvalue and corresponding eigenvector. Useful at a saddle point. After E, use choice S to move along eigenvector direction to minimum energy. Choice P finds the number of eigenvalues above and below a value.

For more details, see the Hessian section of the Technical Reference chapter.

§4.20. Eigenvalues and eigenvectors.

If the surface has reached a saddle point, it is nice to be able to analyze the Hessian. A positive definite Hessian proves the current surface to be a stable local minimum. Negative eigenvalues indicate directions of energy decrease. Zero eigenvalues may indicate symmetries such as translation or rotation. For background, see any decent linear algebra text, such as Strang [SG], especially chapter 6. For more on stability and bifurcations, see Arnol'd [AV] or Hale and Kocak [HK].

Several commands are available to analyze eigenvalues. The inertia of the Hessian with respect to some probe value may be found with "`eigenprobe` *value*", which reports the number of eigenvalues less than, equal to, or greater than the given probe value. Eigenvalues near the probe value may be found with the "`lanczos` *value*" command, which will print out 15 approximate eigenvalues near the probe value. The nearest is very accurate, but others may not be. Also, the multiplicities are often spurious. Since `lanczos` starts with a random vector, it can be run multiple times to get an idea of the error. For more accurate eigenvalues and multiplicities, there is the `ritz(`*value*`,` n`)` command, which takes a random n-dimensional subspace and applies shifted inverse iteration to it. It reports eigenvalues as they converge to machine precision. You can interrupt it by hitting the interrupt key (usually CTRL-C), and it will report the current values of the rest. The `saddle` command will find the lowest eigenvalue, and, if negative, will seek in the corresponding direction for the lowest energy.

The default Hessian usually has many small eigenvalues due to the approximate translational freedom of vertices tangentially to the surface. This is an especially troublesome problem in quadratic mode, where there tend to be lots of tiny negative eigenvalues on the order of -0.00001 that are extremely hard to get rid of by iteration. The cure for this is to restrict the motion of the vertices to be perpendicular to the surface. The `hessian_normal` command toggles this mode. The normal direction is defined to be the volume gradient. Points where this is not well-defined, such as points on triple lines, are not restricted. This mode is highly recommended to speed the convergence of regular Hessian iteration also.

You may want to approximate the eigenvalues and eigenvectors of a smooth surface. This requires some careful thought. First of all, `hessian_normal` mode should be used, since smooth surfaces are considered to move only in normal directions. Second, you may notice that eigenvalues change when you refine. This is due to the fact that an eigenvector is a critical value for the Rayleigh quotient

$$\frac{X^T H X}{X^T X}.$$

Now the presence of $X^T X$ is a clue that a metric is implicitly involved, and it is necessary to include the metric explicitly. Eigenvectors should really be regarded as critical points of the Rayleigh quotient

$$\frac{X^T H X}{X^T M X}.$$

The default metric is $M = I$, which gives unit weight to each vertex. However, for smooth surfaces, eigenfunctions are calculated by using the L_2 metric, which is integration on the surface:

$$< f, g >= \int_S f(x) g(x)\, dx.$$

There are a couple of ways to approximate the L_2 metric on an Evolver surface. One is to take a diagonal M where the weight of each vertex is just the area associated to the vertex, 1/3 of the area of the adjacent facets, to be precise. I refer to this as the "star metric". Another way is to take the true L_2 metric of the linear interpolations of functions defined by their values at the vertices. I call this the "linear metric". The `linear_metric` command toggles the use of metrics in the calculation of eigenvalues and eigenvectors. By default, the metric is a 50-50 mix of the star metric and the linear metric, since that gave the most accurate approximations to the smooth surface eigenvalues in a couple of test cases. But you may control the mix by setting the variable `linear_metric_mix` to the proportion of linear metric that you want.

In quadratic mode, `linear_metric` actually uses quadratic interpolation. No star metric is used, because eigenvalues with star metric converge like h^2 while eigenvalues with pure quadratic interpolation converge like h^4.

If you want to actually see the eigenvectors, you need to go to `hessian_menu`. Do choice 1 to initialize, then choice V with the approximate value of your choice. This will do shifted inverse iteration, and you will probably need less than 50 iterations to get a good eigenvector. Then use choice 4 to move by some multiple of the eigenvector. Choice 7 undoes moves, so don't be afraid.

§4.21. Mobility.

There is a choice to be made in the conversion of the forces on vertices into velocities of vertices. Technically, force is the gradient of energy, hence a covector on the manifold of all possible configurations. In the Evolver, that global covector can be represented as a covector at each vertex. The velocity is a global vector, which is represented as a vector at each vertex. Conversion from the global covector to the global vector requires multiplication by a metric tensor, i.e. singling out a particular inner product on global vectors and covectors. The tensor converting from force to velocity is the *mobility tensor*, represented as the *mobility matrix* M in some coordinate system. Its inverse, converting from velocity to force, is the *resistance tensor* $S = M^{-1}$. The same inner product has to be used in projecting the velocity tangent to the constraints, whether they be level set constraints on vertices or constraints on body volumes or quantity integrals. There are several choices implemented in the Evolver, corresponding to several different physical pictures of how the medium resists the motion of the surface through it.

§4.21.1. Vertex mobility.

This is the default mobility, in which the velocity is equal to the force. Hence M and S are the identity matrices in standard coordinates. The physical interpretation of this is that there is a resistance to motion of each vertex through the medium proportional to its velocity, but not for the edges. This does not approximate motion by mean curvature, but it is very easy to calculate.

§4.21.2. Area normalization.

In motion by mean curvature, the resistance to motion is really due to the surfaces, not the vertices. One way to approximate this is to say the resistance to motion of a vertex is proportional to the area associated with the vertex. So this scheme makes the resistance of a vertex equal to 1/3 of the area of the star of facets around it (or 1/2 the area of the star of edges in the string model). This is easy to calculate, since it is a local calculation for each vertex. S and M are diagonal matrices.

§4.21.3. Area normalization with effective area.

Simple area normalization as described in the previous paragraph isn't what's really wanted in certain circumstances. It has equal resistance for motion in all directions, both parallel and normal to the surface. If a vertex is a triple junction and migrating along the direction of one of the edges, it shouldn't matter how long that edge is. Therefore, if the effective area mode is in effect, the area associated with a vertex is the area of its star projected normal to the force at the vertex. This is a little more complicated calculation, but it is still local. S and M are block diagonal matrices, with one block for each vertex. At a free edge not on any constraint, the force is tangent to the surface, the resistance is zero, and the mobility is infinite. But this accurately describes a popping soapfilm.

§4.21.4. Approximate polyhedral curvature.

Following a suggestion of Gerhard Dzuik and Alfred Schmidt, the inner product of global vectors is taken to be the integral of the scalar product of their linear interpolations over the facets (or edges in the string model). This has the advantage that the rate of area decrease of the surface is equal to the rate volume is swept out by the surface, which is a characteristic of motion by mean curvature. A big disadvantage is that the matrices M and S are no longer local. See chapter 7 for the details. S is a sparse matrix with entries corresponding to each pair of vertices joined by an edge, and M is its dense inverse.

§4.21.5. Approximate polyhedral curvature with effective area.

The previous section did not make any distinction between motion parallel and perpendicular to the surface. A better approximation is to count only motion perpendicular to the surface. This can be done by projecting the interpolated vectorfields normal to the facets before integrating their scalar product. Now the rate of area decrease is equal to the rate geometric volume is swept out, as opposed to the slightly flaky way one had to calculate volume sweeping in the previous paragraph. Again S is a sparse matrix with entries corresponding to each pair of vertices joined by an edge, and M is its dense inverse.

§4.21.6. User-defined mobility.

The user may define a mobility tensor in the datafile. There is a scalar form with the keyword `MOBILITY` and a tensor form with `MOBILITY_TENSOR`. When in effect, this mobility is multiplied times the velocity to give a new velocity. This happens after any of the previous mobilities of this section have been applied and before projection to constraints. The formulas defining the mobility may include adjustable parameters, permitting the mobility to be adjusted during runtime.

§4.22. Stability.

The timestep of an iteration should not be so large as to amplify perturbations of the surface. Short wavelength perturbations are most prone to amplification. This section contains a sketch of the stability characteristics of the various mobility modes, enough to let the user relate the maximum timestep to the minimum facet or edge size. Two examples are discussed: a zigzag string and a nearly flat surface with equilateral triangulation. Effective area is not included, as it is an insignificant correction for nearly flat surfaces. The general moral of this section is that the maximum time step in iteration is limited by the length of the shortest edge or the area of the smallest facet, except in one case.

§4.22.1. Zigzag string.

Let the amplitude of the perturbation about the midline be Y and the edge length L. Then the force on a vertex is $F = -4Y/L$ for small Y. Let the timestep (the Evolver scale factor) be Δt. Let V be the vertex velocity. Then the critical timestep for amplification of the perturbation is given by $V\Delta t = -2Y$, or $\Delta t = -2Y/V$.

Vertex mobility. Here $V = F$, so $\Delta t = L/2$.

Area normalization. Here the vertex star has length $2L$, so $V = F/L$ and $\Delta t = L^2/2$.

Approximate curvature. It turns out that the zigzag is an eigenvector of the mobility matrix M for the largest eigenvalue 3/L, so $V = 3F/L$ and $\Delta t = L^2/6$. This is a major disadvantage of approximate curvature. If perturbation instability is the limitation on the timestep, it will take three times as many iterations as with area normalization to do the same evolution.

§4.22.2. Perturbed sheet with equilateral triangulation.

Consider a plane surface triangulated with equilateral triangles of area A. The perturbation consists of a tiling of hexagonal dimples with their centers of height Y above their peripheries. The force at a central vertex turns out to be $-\sqrt{3}Y$ and the force at a peripheral vertex $\sqrt{3}Y/2$.

Vertex mobility. The critical time step is given by

$$(\sqrt{3}Y + \sqrt{3}Y/2)\Delta t = 2Y,$$

so $\Delta t = 4/3\sqrt{3}$. Note that this is independent of the triangle size. This is consistent with experience in evolving with optimizing scale factor, where the optimum time step is in the range 0.2 - 0.3 indenpendent of triangle size. This is a definite advantage of this version of mobility, since different parts of the surface can have different size triangulations and one size time step can work for all.

Area normalization. The star area of each vertex is $6A$, so the velocities become $-\sqrt{3}Y/2A$ and $\sqrt{3}Y/4A$, and the critical time step $\Delta t = 4/6\sqrt{3}A$. Hence on a surface with varying size triangles, the timestep is limited by the area of the smallest triangles.

Approximate area. This force turns out to be an eigenvector of the mobility matrix with eigenvalue $2/A$. Hence the velocity is four times that of the area normalization, and the critical time step four times shorter.

§4.23. Topology changes.

The term “topology of the surface” refers to the topology of the union of the facets. The term “triangulation” refers to the specfic way facets subdivide the surface. Some operations, such as iteration, change neither. Some, such as refinement, change the triangulation but not the topology. Some, such as short edge elimination, can change both. Some operations, such as vertex popping, are meant to change the topology.

§4.24. Refinement.

“Refinement” of a triangulation refers to creating a new triangulation by subdividing each triangle of the original triangulation. The scheme used in the Evolver is to create new vertices at the midpoints of all edges and use these to subdivide each facet into four new facets each similar to the original. (See the Technical Reference chapter for simplex refinement.)

Certain attributes of new elements are inherited from the old elements in which they were created. Fixedness, constraints, and boundaries are always inherited. Torus wrapping on edges is inherited by some but not all new edges. Surface tension and displayability are inherited by the new facets. ‘Extra’ attributes are inherited by the same type of element.

Refinement can change surface area and energy if there are curved constraints or boundaries. Likewise for body volumes.

§4.25. Adjustable parameters and variables.

The user may define named variables or “parameters” and assign them values. They may be used in expressions whenever the expression does not have to be a constant, such as constraint formulas or boundary parameterizations. Their values may be changed with the `A` command, or by assignment, e.g. “`foo := 3.4`” from the command prompt. Variables may be created by defining them in the top section of the datafile as “`parameter foo = 3.4`”, or by assigning a value from the command prompt. All parameters are real-valued. Changing the value of a parameter declared in the top of the datafile will cause automatic recalculation of the surface, on the assumption that such a parameter is used in a constraint or boundary, unless `AUTORECALC` has been toggled off.

It is possible to make a parameter one of the optimization variables (along with the vertex coordinates) by declaring it with `optimizing_parameter` instead of `parameter` in the datafile. Energy gradients and hessiansi with respect to optimizing parameters are calculated numerically rather than symbolically, so there is a loss of speed and accuracy.

At runtime, a parameter may be toggled to be optimizing or not with the FIX and UNFIX commands. That is, `fix radius` would make the radius variable non-optimizing (fixed value).

§4.26. The String Model.

This section lists the differences you need to be aware of when using the string model.

In general, a face takes on the role of a body, and an edge takes on the role of a facet. Coordinates are still in 3D. If you want cells of prescribed volume, always put z = 0 for the vertices and for each face make a body with just that one face as boundary.

The face section of the datafile is optional.

Faces are not subdivided on input.

Face area is calculated solely using x,y coordinates, even though vertices are still 3D.

Content refers to area.

Constraint energy and content integrands are scalars (have only one component, `E1` or `C1` as the case may be, and are evaluated at vertices on constraints. Tails of edges count negative, and heads positive.

Chapter 5. The Datafile.

§5.1. Datafile organization.

The initial configuration of the surface is read from an ASCII datafile. The datafile is organized into six parts:

1. Definitions and options
2. Vertices
3. Edges
4. Faces
5. Bodies
6. Commands

In the syntax descriptions below, keywords will be in upper case. *const_expr* means a constant expression, and *expr* means any expression. *n* or *k* means an integer, which may be signed if it is being used as an oriented element label. Square brackets denote optional items. '|' means 'or'.

§5.2. Lexical format.

For those who know about such things, the datafile is read with a lexical analyzer generated by the `lex` program. The specification is in `datafile.lex`.

§5.2.1. Comments.

Comments may be enclosed in /* */ pairs (as in C) and may span lines. // indicates the rest of the line is a comment, as in C++.

§5.2.2. Lines and line splicing.

The file is made up of lines. Line breaks are significant. The next physical line can be spliced onto the current line by having '\' be the last character of the current line. Line splicing is not effective in // comments. Blank lines and comment lines may be placed freely anywhere in the datafile. The various combinations of CR and NL that various computer systems use are all recognized.

§5.2.3. Including files.

The standard C language method of including other files is available. The file name must be in quotes. If the file is not in the current directory, EVOLVERPATH will be searched. Includes may be nested to something like 10 deep. Example:

`#include` "common.stuff"

§5.2.4. Macros.

Simple macros (no parameters) may be defined as in C:

`#DEFINE` *identifier string*

identifier must be an identifier without other special meaning to the parser (see the keyword list below). *string* is the rest of the logical line, not including comments. It will be substituted for *identifier* whenever *identifier* occurs as a token subsequently. Substitutions are re-scanned. No checks for recursiveness are made. There is a maximum length (currently 500 characters) on a macro definition. Note: macro identifiers are separate tokens, so if "-M" translates into "-2", this will be read as two tokens, not a signed number.

The keyword `keep_macros` in the datafile will keep macro definitions active during runtime, until the next datafile is loaded.

§5.2.5. Case.

Case is not significant. All letters are converted to lower case on input.

§5.2.6. Whitespace.

Whitespace consists of spaces, tabs, commas, colons, and semicolons. So it's fine if you want to use commas to separate coordinate values. CTRL-Z is also whitespace, for benefit of files imported from DOS.

§5.2.7. Identifiers.

Identifiers follow standard C rules (composed of alphanumeric characters and '_' with the leading character not a digit) and must not be keywords. Identifiers are used for macros and adjustable constants. Use at least two characters, since single characters can be confused with commands. To find out if a name is already in use as a keyword or user-defined name, use the `is_defined` function, which has the syntax

is_defined(*stringexpr*)

The *stringexpr* must be a quoted string or other string expression. The return value is 0 if the name is undefined, 1 if defined.

§5.2.8. Real numbers.

Values may be in any form acceptable to `atof()`. This includes integers, fixed point, and scientific notation such as

```
2    -3    .5    23.        5e-10    +0.7D2
```

§5.2.9. Keywords.

All words mentioned in this manual as having special meaning to the Evolver should be regarded as reserved words and are not available for use by the user as identifiers for variables, commands, quantity names, etc.

§5.2.10. Colors.

The colors of edges and facets are recorded as integers. How these integers translate to colors on the screen is determined by how the graphics drivers are written. The following synonyms are supplied for the integers 0 through 15, and it is hoped that the graphics drivers will be written to display these correctly: `BLACK`, `BLUE`, `GREEN`, `CYAN`, `RED`, `MAGENTA`, `BROWN`, `LIGHTGRAY`, `DARKGRAY`, `LIGHTBLUE`, `LIGHTGREEN`, `LIGHTCYAN`, `LIGHTRED`, `LIGHTMAGENTA`, `YELLOW`, and `WHITE`. The special color value `CLEAR` (-1) makes a facet transparent. These tokens are simply translated to integer values wherever they occur, so these are reserved words.

§5.2.11. Expressions.

Variable expressions are used in constraint and boundary formulas and integrands. Constant expressions can be used wherever a real value is needed. Expressions are given in algebraic notation with the following tokens:

`x1,x2,...`	coordinates
`x,y,z,w`	coordinates
`p1,p2`	parameters for boundaries
constant	any integer or real number
`G`	current gravitational constant
identifier	adjustable constant
`E, PI`	special constants
`+,-,*,/,%,mod`	real arithmetic
`imod,idiv`	integer arithmetic
`=`	treated as low-precedence `-`
`(,)`	grouping and functional notation

`^`	raise to real power
`**`	raise to real power
`? :`	conditional expression, as in C language
functions:	
`abs`	absolute value
`sqr`	square
`sin,cos,tan`	trig functions
`acos,asin,atan`	inverse trig functions (acos, asin arguments clamped to [-1,1])
`atan2`	inverse tangent, atan2(y,x)
`sqrt`	square root (leeway of -1e-10 on sqrt(0))
`log,exp`	natural log, exponentiation base e
`sinh,cosh,tanh`	hyperbolic functions
`asinh,acosh,atanh`	inverse hyperbolic functions
`pow`	pow(x,y): raise x to real power y
`ceil,floor`	round up or down to integer
`minimum,maximum`	of two arguments, i.e. minimum(a,b)
`usr`*n*	user-defined functions

User-defined functions can be defined in C in `userfunc.c`. They are meant for situations where expression interpretation is too slow, or functions such as elliptic integrals are wanted. Currently, they are automatically functions of the coordinates. Do not give any arguments in the expression; for example "`(usr1 + usr3)/usr10`".

Expressions are parsed into evaluation trees, which are interpreted when needed. Constant subtrees are folded into constant nodes at parse time. For using compiled functions in certain places, see the section on dynamic load libraries at the end of the Install chapter.

Constant expressions must evaluate to values when they are read, i.e. they have no parameters or coordinates.

In parsing an expression, the longest legal expression is used. This permits coordinates to be specified by several consecutive expressions with no special separators.

NOTES: A '+' or '-' preceded by whitespace and followed by a number is taken to be a signed number. Thus "3 - 5" and "3-5" are single expressions, but "3 -5" is not. This is for convenience in separating multiple expressions listed on the same line for vertex coordinates, metric components, etc. If in doubt about how a '-' will be interpreted, or if you get error messages about "not enough values", check out the minus signs.

The mod operator '%' or `mod` does a real modulus operation:

$$x\%y = x - floor(x/y) * y.$$

The integer operator `idiv` rounds its operands toward zero before doing integer division (as implemented in C). `imod` rounds its operands down:

$$x \text{ imod } y = floor(x) - floor(floor(x)/floor(y)) * floor(y).$$

§5.3. Datafile top section: definitions and options.

Each line starts with a keyword. The order is immaterial, except that identifiers must be defined before use and quantities must be defined before referring to them in constraints. None of these are required, but those marked as default will be assumed unless an overriding option is present.

§5.3.1. Macros.

`#DEFINE` *identifier string*

See Macros section above.

§5.3.2. Version check.

```
evolver_version "2.10"
```

If a datafile contains features present only after a certain version of the Evolver, the datafile can contain a line of the above form. This will generate a version error message if the current version is earlier, or just a syntax error if run on an Evolver version earlier than 2.10.

§5.3.3. Variables.

`PARAMETER` *identifier* = *const_expr*

This declares *identifier* to be a variable with the given initial value. The value may be changed with the `A` command, or by assignment. Variables may be used in any subsequent expression or constant expression. Changing variables defined heres results in automatic recalculation of the surface, unless `AUTORECALC` has been toggled off.

`OPTIMIZING_PARAMETER` *identifier* = *const_expr* `PDELTA =` *const_expr* `SCALE =` *const_expr*

This declares a variable as above with the additional property that it is subject to optimization. That is, it joins the vertex coordinates in the set of independent variables. It is different from coordinates in the sense that its gradient is calculated by finite differences rather than analytically. Hence it may be used in any kind of expression where a variable is permitted. Hessians of optimizing parameters are implemented. The optional pdelta value is the parameter difference to use in finite differences; the default value is 0.0001. The optional scale value is a multiplier for the parameter's motion, to do "impedance matching" of the parameter to the surface energy. These attributes may be set on any parameter, for potential use as an optimizing parameter. At runtime, a parameter may be toggled to be optimizing or not with the FIX and UNFIX commands. That is, `fix radius` would make the radius variable non-optimizing (fixed value). "Optimising_parameter" is a synonym.

§5.3.4. Dimensionality.

`STRING`

The Evolver is to use the string model (see the **String Model** section)

`SOAPFILM` (default)

The Evolver is to use the standard two-dimensional surface model.

`SURFACE_DIMENSION` *const_expr*

Surface is of given dimension. Dimension over 2 is valid only in the simplex model.

§5.3.5. Domain.

`EUCLIDEAN` (default)

The surface lives in Euclidean space.

`SPACE_DIMENSION` *const_expr*

The surface lives in Euclidean space of the given dimension. Default is 3. The dimension must be at most the value of `MAXCOORD` in `model.h`, which is 4 in the distributed version.

`TORUS`

The surface lives in a 3 dimensional flat torus. Surface is equivalent to a periodic surface.

`TORUS_FILLED`

Indicates the entire torus is filled by bodies to enable the program to avoid degenerate matrices when adjusting constrained volumes.

`PERIODS`
expr *expr* *expr*
expr *expr* *expr*
expr *expr* *expr*

Used with `TORUS` domain. Specifies that the side vectors (basis vectors) of the unit cell parallelpiped follow on the next lines. Each vector is given by its components. The size of this matrix depends on the space dimension. Default is unit cube. Adjustable parameters may be used in the expressions, so the fundamental domain may be changed interactively with either the 'A' command or by assigning new values to the parameters. Be sure to do a '`recalc`' to get the period matrix re-evaluated.

`SYMMETRY_GROUP "`*name*`"`

The domain is the quotient of R^n by a user-defined symmetry group. The user must link in C functions that define the group operations. See `quotient.c` for an example. "*name*" is a double-quoted name that is checked against a name in `quotient.c` to be sure the right symmetry group is linked in.

`SYMMETRIC_CONTENT`

For body volume integrals, use an alternate surface integral

$$V = \frac{1}{3}\int\int_{body\ surface}(x\vec{i} + y\vec{j} + z\vec{k})\cdot d\vec{S}. \tag{1}$$

It is useful if unmodelled sides of a body are radial from the origin, or if constraint content integrals (which is evaluated by an approximation) lead to asymmetric results on what should be a symmetric surface.

§5.3.6. Representation.

`LINEAR` (default)

Facets are flat triangles.

`QUADRATIC`

Facets are quadratic patches. See the **Quadratic Representation** section.

`LAGRANGE $n$`

The surface is in the Lagrange order n representation. Best not to try to create a Lagrange representation input file by hand. This phrase is in here so dump files of the Lagrange representation may be reloaded.

`SIMPLEX_REPRESENTATION`

Facets are defined by oriented vertex list rather than edge list. See the section above on **Simplex Model**.

§5.3.7. Dynamic load libraries.

To load a ¡a href="install.htm"¿dynamic library¡/a¿ of compiled functions, the syntax is

`LOAD_LIBRARY "`*filename*`"`

where the double-quoted filename is the library. The current directory and the `EVOLVERPATH` will be searched for the library. For details on how to set up and use a dynamic load library, see the Installation chapter.

§5.3.8. Extra attributes.

It is possible to dynamically define extra attributes for elements, which may be single values or vectors. The definition syntax is

DEFINE *elementtype* ATTRIBUTE *name type* [[*dim*]]

where *elementtype* is `vertex, edge, facet,` or `body`, *name* is an identifier of your choice, and *dim* is an optional expression for the vector dimension. *Type* is `REAL` or `INTEGER` (internally there is also a `ULONG` unsigned long type also). There is no practical distinction between real and integer types at the moment, since everything is stored internally as reals. But there may be more datatypes added in the future. Extra attributes are inherited by elements of the same type generated by subdivision. Examples:

```
define edge attribute charlie real
define vertex attribute oldx real[3]
```

The dimension of an extra attribute may be retrieved at runtime with the `sizeof` function, with the syntax

SIZEOF(*name*)

§5.3.9. Surface tension energy.

The surface tension energy is always included in the total energy by default. It can be turned off only by giving the facets (or edges in the string model) the "`density 0`" attribute.

AREA (default)

Surface energy of a facet is its area times its surface tension.

WULFF "*filename*"

Specifies that a crystalline integrand is to be used. The next token should be a double-quoted filename (with path) of a file giving the Wulff vectors of the integrand. The format of the file is one Wulff vector per line with its three components in ASCII decimal format separated by spaces. The first blank line ends the specification. Some special integrands can be used by giving a special name in place of the file name. Currently, these are `"hemisphere"` for a Wulff shape that is an upper unit hemisphere, and `"lens"` for two unit spherical caps of thickness 1/2 glued together on a horizontal plane. These two don't need separate files.

PHASEFILE "*filename*"

This permits the surface tension of a grain boundary to depend on the phases or types of the adjacent grains. The information is read from an ASCII file. The first line of the file has the number of different phases. Each line after consists of two phase numbers and the surface tension between them. Lines not starting with a pair of numbers are taken to be comments.If a pair of phases is not mentioned, the surface tension between them is taken to be 1.0. Facets in the string model or bodies in the soapfilm model can be labelled with phases with the `PHASE` phrase.

§5.3.10. Squared mean curvature.

SQUARE_CURVATURE *const_expr*

This phrase indicates that the integral of squared mean curvature will be included in the energy with a weight given by *const_expr*. The weight can be changed with the `A` command by changing the value of the adjustable constant `square curvature modulus`.

§5.3.11. Integrated mean curvature.

MEAN_CURVATURE_INTEGRAL *const_expr*

This phrase indicates that the integral of mean curvature will be included in the energy with a weight given by *const_expr*. The weight can be changed with the `A` command by changing the value of the adjustable constant `mean curvature modulus`.

§5.3.12. Squared Gaussian curvature.

`SQUARE_GAUSSIAN_CURVATURE` *const_expr*

This phrase indicates that the integral of squared Gaussian curvature will be included in the energy with a weight given by *const_expr*. The weight can be changed with the `A` command by changing the value of the adjustable constant `square Gaussian modulus`. Synonyms: `squared_gaussian_curvature`, `sqgauss`.

§5.3.13. Ideal gas model.

`PRESSURE` *const_expr*

This specifies that bodies are compressible and the ambient pressure is the given value. The default is that bodies with given volume are not compressible.

§5.3.14. Gravity.

`GRAVITY_CONSTANT` *const_expr*

Specifies gravitational constant G. Default 1.0.

§5.3.15. Gap energy.

`GAP_CONSTANT` *const_expr*

Multiplier for convex constraint gap energy. Default 1.0. Synonym: `spring_constant`

§5.3.16. Knot energy.

There are a bunch of named quantity methods for knot energies, and that syntax should be used. But there are a couple of synonyms floating around.

`INSULATING_KNOT_ENERGY` *const_expr*

The total energy will include the knot energy method knot_energy with multiplier *const_expr*. Abbreviation for a named quantity.

`CONDUCTING_KNOT_ENERGY` *const_expr*

The total energy will include the knot energy method edge_knot_energy with multiplier *const_expr*. Abbreviation for a named quantity.

§5.3.17. Mobility and motion by mean curvature.

`GRADIENT_MODE` (default)

Velocity is equal to the force.

`AREA_NORMALIZATION`

The velocity of a vertex is the force divided by 1/3 area of neighboring facets (or 1/2 length of neighboring edges in string model) to approximate motion by mean curvature.

`APPROXIMATE_CURVATURE`

Calculates vertex velocity from force by means of polyhedral linear interpolation inner product. Do not use together with `AREA_NORMALIZATION`.

`EFFECTIVE_AREA`

For both area normalization and approximate curvature modes, the resistance to motion is only on the component of velocity normal to the surface.

§5.3.18. Annealing.

`JIGGLE`

Continuous jiggling. Default none.

`TEMPERATURE` *const_expr*

Gives temperature for jiggling. Default 0.05.

§5.3.19. Diffusion.

`DIFFUSION` *const_expr*

Specifies that diffusion between bodies is in effect with diffusion constant as given. Default 0.

§5.3.20. Named method instances.

These are methods of calculating scalar values for geometric elements that are referred to by name. They are used by named quantities (see next subsection). For which models each is valid in (linear, quadratic, Lagrange, simplex, etc.), see Chapter 4.

`METHOD_INSTANCE` *name* `METHOD` *methodname* [`MODULUS` *constexpr*] [`GLOBAL`] *parameters*

This is an explicit definition of a named method instance for a named quantity. Such an explicit definition is useful if your quantity has several method instances, and you want to see their individual values or apply the instances to different elements. The modulus multiplies the method value to give the instance value. The default modulus is 1. `GLOBAL` makes the method apply to all elements of the appropriate type. Non-global instances may be applied to elements individually.

Each method may have various parameters to specialize it to an instance. Currently the only parameters specified here are scalar integrands, with syntax

`SCALAR_INTEGRAND:` *expr*

vector integrands, with syntax

`VECTOR_INTEGRAND:`
`Q1:` *expr*
`Q2:` *expr*
`Q3:` *expr*

2-form integrands, with syntax

`FORM_INTEGRAND:`
`Q1:` *expr*
`Q2:` *expr*
`Q3:` *expr*
...

where the form components are listed in lexicographic order, i.e. in 4D the six components 12,13,14,23,24,34 would be listed as Q1 through Q6.

The expressions may use attributes for individual elements (density, length, extra attributes, etc.)

Some methods use global parameters (a holdover that will be done away with eventually.) The instance definition does not have to be on one line.

See the Named Methods and Quantities chapter for a list of the methods available and the specifications each needs.

§5.3.21. Named quantities.

`QUANTITY` *name* `ENERGY|` `FIXED =` *value*| `INFO_ONLY` `MODULUS` *constexpr* `LAGRANGE_MULTIPLIER` *constexpr*

`TOLERANCE` *constexpr* *methodlist* | `FUNCTION` *methodexpr*

These are an effort to provide a more systematic way of adding new energies and constraints. A "method" is a way of calculating a scalar value from some particular type of element (vertex, edge, facet, body). A "quantity" is the sum total of various methods applied to various elements, although usually just one method is involved. The name is an identifier. Any quantities may be declared to be one of three types: 1) energy quantities are added to the overall energy of the surface; 2) fixed quantities that are constrained to a fixed target value (by a single Newton step at each iteration); and 3) information quantities whose values are merely reported to the user.

Each quantity has a "modulus", which is just a scalar multiplier of the whole quantity. A modulus of 0 will turn off an energy quantity. The default modulus is 1. Adding a new method involves writing C routines to calculate the value and the gradient as a function of vertex coordinates, and adding a structure to the method name array in `quantity.c`.

For fixed quantities, the optional Lagrange multiplier value supplies the initial value of the Lagrange multiplier (the "pressure" attribute of the quantity). It is meant for dump files, so on reloading no iteration need be done to have a valid Lagrange multiplier.

For fixed quantities, the tolerance attribute is used to judge convergence. A surface is deemed converged when the sum of all ratios of quantity discrepancies to tolerances is less than 1. This sum also includes bodies of fixed volume. If the tolerance is not set or is negative, the value of the variable target_tolerance is used, which has a default value of 0.0001.

The *methodlist* version of the quantity definition may contain one or more method instances. To incorporate a previously explicitly defined instance, include

`METHOD` *instancename*

To instantiate a method in the quantity definition, you essentially incorporate the instance definition, but without an instance name:

`METHOD` *methodname* [`MODULUS` *constexpr*] [`GLOBAL`] *parameters*

See the previous subsection for method details. Usually the second, implicit definition will be more convenient, as there is usually only one method per quantity. If `GLOBAL_METHOD` is used instead of `GLOBAL`, then the method is applied to all elements of the proper type. It is equivalent to using the `GLOBAL` keyword in the method specification. Nonglobal instances must be applied individually to elements. That is done by simply adding the quantity or instance name to the line defining the element. If a quantity name is used, then all method instances of that quantity of the appropriate type are applied to the element. Original attachments of quantities are remembered, soIf an edge method is applied to a facet, then edges created from refining that facet will inherit the edge method. Orientable methods can be applied with negative orientation to elements in the datafile by following the name with a dash. The orientation in a set command follows the orientation the element is generated with.

The quantity may also be defined by an arbitrary function of method instances. The keyword `FUNCTION` indicates this, and is followed by the defining function expression. The method instances involved must all have been previously defined as named method instances.

Quantity values can be seen with the `Q` or `A` command, or may be referred to as "*quantityname*`.value`" in commands. Do not use just *quantityname* since *quantityname* alone is interpreted as an element attribute. The `A` command can be used to change the target value of a fixed quantity, the modulus of a quantity, or some parameters associated to various methods. Or you can assign values to *quantityname*`.target` or *quantityname*`.modulus`.

Examples:

Hooke energy:

`quantity hooke ENERGY modulus 10 global_method hooke_energy`

A little sample datafile:

```
// test of quantity integrands on constraints
```

```
method_instance len method edge_length
quantity lenny info_only method len

quantity sam info_only method facet_scalar_integral
scalar_integrand z

vertices
1   0 0 1 fixed
2   1 1 2 fixed
3   0 1 3 fixed

edges
1   1 2 lenny
2   2 3
3   3 1 len

faces
1   1 2 3 sam
```

The sample datafile `knotty.fe` contains more examples.

The keyword `EVERYTHING_QUANTITIES` in the top section of the datafile causes all areas, volumes, etc. to be converted to named quantities and methods. It is equivalent to the command line option -q, or the `convert_to_quantities` command.

§5.3.22. Level set constraints.

`CONSTRAINT` *n* `[GLOBAL] [CONVEX] [NONNEGATIVE | NONPOSITIVE]`
`EQUATION | FORMULA | FUNCTION` *expr*
[`ENERGY`
`E1` *expr*
`E2` *expr*
`E3` *expr*]
[`CONTENT`
`C1` *expr*
`C2` *expr*
`C3` *expr*]

This defines constraint number *n*, where *n* is an integer between 0 and `MAXCONSTR` (currently 127). `GLOBAL` means the constraint automatically applies to all vertices (but not automatically to edges or faces). `GLOBAL` constraints count in the number limit. If `CONVEX` is given, then an additional gap energy is attributed to edges on the constraint to prevent them from trying to short-circuit a convex boundary; see the Constraints and Energy sections above and the `k` command. If `NONNEGATIVE` or `NONPOSITIVE` is given, then all vertices will be forced to conform appropriately to the constraint at each iteration. The `EQUATION` expression defines the zero level set which is the actual constraint. It may be written as an equation, since '=' is parsed as a low-precedence minus sign. Do not use '>' or '<' to indicate inequalities; use `NONNEGATIVE` or `NONPOSITIVE` and an expresson. Conditional expressions, as in C language, are useful for defining constraints composed of several surfaces joined smoothly, such as a cylinder with hemispherical caps.

The formula may include any expressions whose values are known to the Evolver, given the particular vertex. Most commonly one just uses the coordinates (x,y,z) of the vertex, but one can use variables, quantity values, or vertex extra attributes. Using a vertex extra attribute is a good way to customize one formula to individual vertices. For example, if there were a vertex extra attribute called zfix, one could force vertices to individual z values with one constraint with the formula z = zfix, after of course assigning proper values to zfix for each vertex. Be sure to fix up the extra attribute after refining or otherwise creating new vertices, since new vertices will have a default value of 0 for the extra attribute.

NOTE: One-sided constraints can cause the optimal scale factor algorithm to misbehave. It may be necessary to use a fixed scale factor. See the `m` command below.

IMPORTANT NOTE: Do not let two constraints with the same formula apply to a vertex; that leads to a singular matrix inversion when trying to project the vertex onto the constraints. For example, do not have a vertex subject to X1 = 0 and also have a global X1 `NONNEGATIVE`.

`ENERGY` signifies that vertices or edges on the constraint are deemed to have an energy. In the `SOAPFILM` model, the next three lines give components of a vectorfield that will be integrated along each edge on the constraint. In the `STRING` model, just one component is needed, which is evaluated at each vertex on the constraint. The main purpose of this is to permit facets entirely on the constraint to be omitted. Any energy they would have had should be included here. One use is to get prescribed contact angles at a constraint. This energy should also include gravitational potential energy due to omitted facets. Integrals are not evaluated on FIXED edges.

`CONTENT` signifies that vertices (`STRING` model) or edges (`SOAPFILM` model) on the constraint contribute to the area or volume of bodies. If the boundary of a body that is on a constraint is not given as facets, then the body volume must get a contribution from a content integral. It is important to understand how the content is added to the body in order to get the signs right. The integral is evaluated along the positive direction of the edge. If the edge is positively oriented on a facet, and the facet is positively oriented on a body, then the integral is added to the body. This may wind up giving the opposite sign to the integrand from what you think may be natural. Always check a new datafile when you load it to be sure the integrals come out right.

Warning: These integrals are evaluated only for edges which are on the constraints and both of whose endpoints are on the constraints. It is a bad idea to put any of these integrals on one-sided constraints, as both endpoints must actually hit the constraint to count.

§5.3.23. Constraint tolerance.

`CONSTRAINT_TOLERANCE` *const_expr*

This is the tolerance within which a vertex is deemed to satisfy a constraint. Default 1e-12.

§5.3.24. Boundaries.

`BOUNDARY` *n* `PARAMETERS` *k* `[CONVEX]`
`X1` *expr*
`X2` *expr*
`X3` *expr*
[`ENERGY`
`E1` *expr*
`E2` *expr*
`E3` *expr*]
[`CONTENT`
`C1` *expr*
`C2` *expr*
`C3` *expr*]

This defines boundary number *n*, where *n* is an integer between 1 and `MAXBDRY` (currently 20) and *k* is the number of parameters (1 or 2). If `CONVEX` is given, then an additional energy is attributed to edges on the boundary to prevent them from trying to short-circuit a convex boundary; see the `k` menu option below. The following three lines have the functions for the three coordinates in terms of the parameters `P1` and maybe `P2`. Energy and content integrals for boundaries are implemented with the same syntax as for constraints.

§5.3.25. Numerical integration precision.

INTEGRAL_ORDER_1D *n*
INTEGRAL_ORDER_2D *n*

Sets the degree of polynomial done exactly by numerical integration. Edge integrals are done by k-point Gaussian quadrature. This give exact values for polynomials of degree $n = 2k - 1$. Default is $k = 2$, which will do cubic polynomials exactly. No limit on order, as weights and abscissas are calculated by Evolver.

Facet integrals are done with 1, 3, 7, 12, or 28 point integration, corresponding to n =1, 2, 5, 6, or 11.

§5.3.26. Scale factor.

SCALE *const_expr* [FIXED]

Sets the initial scale factor. If FIXED is present, sets fixed scale factor mode. Default is scale = 0.1 and optimizing mode.

SCALE_LIMIT *const_expr*

Sets upper bound on scale factor to prevent runaway motions. Default value is 1. If you use surface tensions and densities not near unity, you may have to set this value.

§5.3.27. Mobility.

MOBILITY_TENSOR
expr *expr* *expr*
expr *expr* *expr*
expr *expr* *expr*

or

MOBILITY *expr*

The force vector is multiplied by the mobility scalar or tensor to get the velocity. Good for, say, having grain boundary mobility depend on temperature.

§5.3.28. Metric.

METRIC
expr *expr* *expr*
expr *expr* *expr*
expr *expr* *expr*

or

CONFORMAL_METRIC
expr

or

KLEIN_METRIC

The user may define a background metric for the string model only. The keyword METRIC is follwed by the N^2 components of the metric tensor, where N is the dimension of space. The components do not have to obey any particular line layout; they may be all on one line, or each on its own line, or any combination. It is up to the user to maintain symmetry. A conformal metric is a scalar multiple of the identity matrix, and only the multiple need be given. A conformal metric will run about twice as fast. The Klein metric is a built-in metric for hyperbolic n-space modelled on the unit disk or ball.

§5.3.29. Autochopping.

AUTOCHOP *const_expr*

This turns on autochopping of long edges. The constant is the maximum edge length.

§5.3.30. Autopopping.

AUTOPOP

This turns on autopopping of short edges and improper vertices.

§5.3.31. Total time.

TOTAL_TIME *const_expr*

This permits setting the initial time of a surface evolving with a fixed scale. Used primarily when resuming from a dump file.

§5.3.32. Runge-Kutta.

RUNGE_KUTTA *const_expr*

This turns on doing iteration with the Runge-Kutta method.

§5.3.33. Homothety scaling.

HOMOTHETY *const_expr*

This turns on doing homothety scaling each iteration. The scaling is a uniform scaling from the origin to keep the total volume of all bodies at the given value.

§5.3.34. Viewing matrix.

VIEW_MATRIX
const_expr const_expr const_expr const_expr
const_expr const_expr const_expr const_expr
const_expr const_expr const_expr const_expr
const_expr const_expr const_expr const_expr

For the specification of the initial viewing transformation matrix of the surface. The matrix is in homogeneous coordinates with translations in the last column. The size of the matrix is one more than the space dimension. This matrix will be part of all dump files, so the view can be saved between sessions. This matrix only applies to internal graphics (Postscript, Xwindows, etc.) and not external graphics (geomview). The elements may be read at runtime by `view_matrix[i][j]`, where the indices start at 1.

§5.3.35. View transforms.

VIEW_TRANSFORMS *integer*
[color *color*]
[swap_colors *swap_colors*]
const_expr const_expr const_expr const_expr
const_expr const_expr const_expr const_expr
const_expr const_expr const_expr const_expr
const_expr const_expr const_expr const_expr
...

For the display of several transformations of the surface simultaneously, a number of viewing transformation matrices may be given. The transforms apply to all graphics, internal and external, and are prior to the viewing transformation matrix for internal graphics. The identity transform is always done, so it does not need to be specified. The number of matrices follows the keyword `VIEW_TRANSFORMS`. Each matrix is in homogeneous coordinates. The size of each matrix is one more than the space dimension. Individual matrices need no special separation; Evolver just goes on an expression reading frenzy until it has all the numbers it wants. Each matrix may be preceded by a color specification that applies to facets transformed by that matrix. The color applies to one transform only; it does not continue until the next color specification. If SWAP_COLORS is present instead, facet frontcolor and backcolor will be swapped when this matrix is applied. Transforms may be activated or deactivated interactively with the **transforms on** or **transforms off**. See the next paragraph for a more sophisticated way to control view transforms.

§5.3.36. View transform generators.

`VIEW_TRANSFORM_GENERATORS` *integer*
`SWAP_COLORS`
const_expr const_expr const_expr const_expr
const_expr const_expr const_expr const_expr
const_expr const_expr const_expr const_expr
const_expr const_expr const_expr const_expr
...

Listing all the view transforms as in the previous paragraph is tedious and inflexible. An alternative is to list just a few matrices that can generate transforms. See the `transform_expr` command for instructions on entering the expression that generates the actual transforms. Special Note: in torus mode, the period translations are automatically added to the end of the list. So in torus mode, these are always available, even if you don't have `view_transform_generators` in the datafile. If SWAP_COLORS is present, facet frontcolor and backcolor will be swapped when this matrix is applied.

§5.3.37. Zoom parameters

`ZOOM_RADIUS` *constexpr*

`ZOOM_VERTEX` *constexpr*

Sets the current parameters for the zoom command. Used in dump files, rather than user's original datafiles.

§5.3.38. Fixed area constraint.

`FIXED_AREA` *expr* or AREA_FIXED *expr*

Obsolete method of constraining the total area to a given value. Do not use anymore. Use the facet_area method in a fixed named quantity.

§5.3.39. Merit factor.

`MERIT_FACTOR` *expr*

If the keyword `MERIT_FACTOR` is present, then the `i` command will print the ratio $total\_area^3/total\_volume^2$, which measures the efficiency of area enclosing volume. This is a holdover from the Evolver's early days of trying to beat Kelvin's partition of space.

§5.3.40. Parameter files.

`PARAMETER` *name* `PARAMETER_FILE` *string*

A parameter can be initialized with a set of values from a file, but I forget at the moment how it is all supposed to work.

§5.3.41. Surface energies (obsolete).

Prior to named quantities, a surface integral of a vectorfield could be incorporated into the energy by defining a `surface_energy`. This is now obsolete, retained only for backward compatibility. Use a named quantity with the `facet_vector_integral` method instead.

§5.4. Element lists.

The lists of geometric elements follow a general format. Each element is defined on one line. The first entry on a line is the element number. Numbering need not be consecutive, and may omit numbers, but be aware that internally elements will be renumbered in order. The original number in the datafile is accessible as the "original" attribute of an element. After the element number comes the basic defining data, followed by optional attributes in arbitrary order. Besides the particular attributes for each element type listed below, one may specify values for any extra attributes defined earlier. The syntax is attribute name followed by the appropriate number of values. Also an arbitrary number of named quantities or method instances may be listed. These add method values for this element to the named quantity. The named quantity or instance must have been declared in the top section of the datafile. See the **Named quantity** section above.

§5.5. Vertex list.

The vertex list is started by the keyword `VERTICES` at the start of a line. It is followed by lines with one vertex specification per line in one of these formats:

> *k x y z* [`FIXED`] [`CONSTRAINT` *c1* [*c2*]] [`BARE`] [*quantityname* ...] [*methodname* ...]
> *k p1* [*p2*] `BOUNDARY` *b* [`FIXED`] [`BARE`] [*quantityname* ...] [*methodname* ...]

Here *k* is the vertex number, a positive integer. Vertices do not need to be listed in order, and there may be gaps in the numbering. However, if they are not in consecutive order, then the numbering in dump files will be different. *x*, *y*, and *z* are constant expressions for coordinates in the domain; *p1* and *p2* are constant expressions for parameter values. If `FIXED` is given, then the vertex never moves, except possibly for an initial projection to constraints. If `CONSTRAINT` is given, then one or two constraint numbers must follow. (Actually, you can list as many constraints as you want, as long as those that apply exactly at any time are consistent and independent.) The given coordinates need not lie exactly on the constraints; they will be projected onto them.

If `BOUNDARY` is given, then the boundary parameter values are given instead of the coordinates. The vertex coordinates will be defined by the coordinate formulas of boundary number *b*. A vertex may be on only one boundary.

The `BARE` attribute is just an instruction to the checking routines that this vertex is not supposed to have an adjacent facet in the soapfilm model, so spurious warnings will not be generated. This is useful when you want to show bare wires or outline fundamental domains.

§5.6. Edge list.

The edge list is started by the keyword `EDGES` at the start of a line. It is followed by lines with one edge specification per line in this format (linespliced here):

> *k v1 v2* [*midv*] [*s1 s2 s3*] [`FIXED`] [`BOUNDARY` *b*] [`CONSTRAINTS` *c1 c2* . . .]\
> [`TENSION` | `DENSITY` *const_expr*] [`COLOR` *n*] [`BARE`] [`NO_REFINE`] \
> [*quantityname* ...] [*methodname* ...]

Here *k* is the edge number, with numbering following the same rules as for vertices. *v1* and *v2* are the numbers of the tail and head vertices of the edge. In the quadratic model, the edge midpoint may be listed as a third vertex *midv* (by default, a midpoint will be created). In a `TORUS` model, there follow three signs *s1 s2 s3* indicating how the edge wraps around each unit cell direction: + for once positive, * for none, and - for once negative. `FIXED` means that all vertices and edges resulting from subdividing this edge will have the `FIXED` attribute. It does not mean that the endpoints of the edge will be fixed. (Note: `EFIXED` is an obsolete version of `FIXED` left over from when `FIXED` did fix the endpoints.) Likewise the `BOUNDARY` and `CONSTRAINT` attributes will be inherited by all edges and vertices derived from this edge. If a constraint has energy or content integrands, these will be done for this edge. IMPORTANT: If a constraint number is given as negative, the edge energy and content integrals will be done in the opposite orientation. In the string model, the default tension is 1, and in the soapfilm model, the default tension is 0. However, edges may be given nonzero tension in the soapfilm model, and they will contribute to the energy. `NO_REFINE` means this edge will not be subdivided by the `r` command.

If the simplex model is in effect, edges are one less dimension than facets and given by an ordered list of vertices. Only edges on constraints with integrals need be listed.

The `BARE` attribute is just an instruction to the checking routines that this ede is not supposed to have an adjacent facet in the soapfilm model, so spurious warnings will not be generated. This is useful when you want to show bare wires or outline fundamental domains.

§5.7. Face list.

The face list is started by the keyword `FACES` at the start of a line. It is followed by lines with one facets specification per line in this format:

> *k e1 e2 ...* [`FIXED`] [`TENSION` | `DENSITY` *const_expr*] [`BOUNDARY` *b*] \
> [`CONSTRAINTS` *c1* [*c2*]] [`NODISPLAY`] [`NO_REFINE`] \
> [`COLOR` *n*] [`FRONTCOLOR` *n*] [`BACKCOLOR` *n*] [`PHASE` *n*] \
> [*quantityname ...*] [*methodname ...*]

Here *k* is the face number, with numbering following the same rules as for vertices. There follows a list of oriented edge numbers in counterclockwise order around the face. A negative edge number means the opposite orientation of the edge from that defined in the edge list. The head of the last edge must be the tail of the first edge (except if you're being tricky in the `STRING` model). There is no limit on the number of edges. The face will be automatically subdivided into triangles if it has more than three edges in the soapfilm model. The `TENSION` or `DENSITY` value is the energy per unit area (the surface tension) of the facet; the default is 1. Density 0 facets exert no force, and can be useful to define volumes or in displays. Fractional density is useful for prescribed contact angles. `NODISPLAY` prevents the facet from being displayed. The `COLOR` attribute applies to both sides of a facet; `FRONTCOLOR` applies to the positive side (edges going counterclockwise) and `BACKCOLOR` to the negative side. The `PHASE` number is used in the string model to determine the surface tension of edges between facets of different phases, if phases are used. The `FIXED`, `BOUNDARY`, `CONSTRAINT`, `DENSITY` and `NODISPLAY` attributes will be inherited by all facets, edges, and vertices resulting from the subdivision of the interior of the face. `NO_REFINE` has no effect on refining the facet itself, but does get inherited by edges created in the interior of the facet.

If the simplex model is in effect, the edge list should be replaced by an oriented list of vertex numbers.

The faces section is optional in the STRING model.

§5.8. Bodies.

The body list is started by the keyword `BODIES` at the start of a line. It is followed by lines with one body specification per line in this format:

> *k f1 f2 f3 ....* [`VOLUME` *v*] [`VOLCONST` *v*] [`PRESSURE` *p*] [`DENSITY` *d*] \
> [`PHASE` *n*] [`ACTUAL_VOLUME` *v*] [*quantityname ...*] [*methodname ...*]

Here *k* is the body number, and *f1 f2 f3 ...* is an unordered list of signed facet numbers. Positive sign indicates that the facet normal (as given by the right-hand rule from the edge order in the facet list) is outward from the body and negative means the normal is inward. Giving a `VOLUME` value *v* means the body has a volume constraint, unless the ideal gas model is in effect, in which case *v* is the volume at the ambient pressure. `VOLCONST` is a value added to the volume; it is useful when the volume calculation from facet and edge integrals differs from the true volume by a constant amount, as may happen in the torus model. Beware, though, when changing things that affect body volume; you may have to reset volconst. `ACTUAL_VOLUME` is a number that can be specified in the rare circumstances where the torus volume volconst calculation gives the wrong answer; volconst will be adjusted to give this volume of the body. Giving a `PRESSURE` value p means that the body is deemed to have a constant internal pressure p; this is useful for prescribed mean curvature problems. It is incompatible with prescribed volume. Giving a `DENSITY` value d means that gravitational potential energy (gravitational constant G) will be included. v, p, and d are constant expressions.

To endow a facet with `VOLUME`, `PRESSURE`, or `DENSITY` attributes in the `STRING` model, define a body with just the one facet.

The `PHASE` number is used in the soapfilm model to determine the surface tension of facets between bodies of different phases, if phases are used.

The `BODIES` section is optional.

§5.9. Commands.

Encountering the keyword `READ` in the datafile causes the Evolver to switch from datafile mode to command mode and read the rest of the datafile as command input. This feature is useful for automatic initialization of the surface with refining, iteration, defining your own commands, etc.

Chapter 6. Operation.

§6.1. System command line.

Syntax:

```
evolver [-ffilename] [-a-] [-d] [-e] [-i] [-m] [-pn] [-q] [-x] [-w] [datafile]
```

The current directory and `EVOLVERPATH` will be searched for the datafile. If the datafile is not found, then a new search with extension "`.fe`" is done. If it is not found, or no datafile is given on the command line, the user will be prompted to supply a datafile name.

Options:

`-a-` Do not enable automatic conversion to named methods and quantities mode when a situation requiring it arises.

`-d` Prints YACC debugging trace as datafile is parsed. May be helpful if you can't figure out why your datafile doesn't get read in properly AND you know YACC.

`-e` Echo input. Meant for echoing commands of piped input to screen so the user can follow what is going on in case Evolver is being controlled by another process.

`-f` Specifies the name of a file to be used as command input. At the end of this file, input reverts to stdin. The effect is the same as redirecting input from the file, except that -f will echo commands to the screen and revert to stdin at the end. Also note that errors will cause input to revert to stdin.

`-i` Keeps elements numbers as listed in the datafile, instead of renumbering them consecutively. Leaves no good way to compact or order element ids in list, so it is not default for now.

`-m` Turn memory debugging on at start of program. Same effect as `memdebug` command.

`-p`*n* Forces use of n processes for an Evolver compiled in multi-processor mode. n may be larger or smaller than the physically available number of processors. The default is 1 processor.

`-q` Convert everything to named quantities internally. There are some things for which no quantities exist yet, producing error messages.

`-x` Causes Evolver to exit whenever an error occurs. Meant to be used when Evolver is run in a shell script.

`-w` Causes Evolver to exit whenever a warning occurs. Meant to be used when Evolver is run in a shell script.

§6.2. Initialization.

The following steps occur when a new datafile is read in:

1. Any previous surface has all memory deallocated.
2. Defaults are initialized.
3. The datafile is read in.
4. Any non-triangular faces are divided into triangles in the soapfilm model.
5. The order of facets around each edge is determined geometrically.
6. Vertices are projected to their constraints.
7. Checks as described under the `C` command are run.
8. Initial areas, energies, and volumes are calculated.
9. The viewing transformation matrix is reset to center the surface on the screen.
10. The main command interface is started.

§6.3. Error handling.

There are several categories of errors:

1. `WARNING`. Something has happened that you should know about, but it is possible to proceed.
2. `EXPRESSION ERROR`. There is an error in parsing an expression.
3. `PARSING ERROR`. Error in datafile syntax, but parsing will continue as best it can.
4. `DATAFILE ERROR`. Error in datafile semantics, but parsing will continue as best it can.
5. `ERROR`. The current operation cannot continue. It is abandoned and you return to the command prompt.
6. `FATAL ERROR`. The program cannot continue. Exits immediately.

All error messages go to stderr. Errors in the datafile report line numbers and print the current line so far. Note that the real error may have been at the end of the previous line. If command input is being taken from a file at the time an error occurs, the line number of the offending command will be printed and command input will revert to stdin. If the `-x` option was given when Evolver was started, then Evolver will exit immediately with a nonzero error code.

§6.4. Commands.

The Evolver command language continues to grow by accretion, and it looks like it's headed towards a full programming language. It has variables, expressions, subroutines, conditionals, and iteration constructs. But not subroutine arguments, local variables, structures, or arrays. Variables are either floating point, string, or subroutine names. Some longer examples of command scripts follow the language description.

Commands are of two main types. The first type is one letter (case is significant). These perform simple and common actions. Single-letter commands are listed below in functional groups. The second type is composed of spelled-out words, and is a sort of combination SQL type query language and a programming language.

Commands may be read from the end of the datafile, from a file given on the system command line, from stdin (the terminal), or from a file given in a `READ` command. The interactive command prompt is "`Enter command: `".

§6.5. General language syntax.

Commands are entered one line at a time, parsed, and executed. Multi-line commands may be entered by enclosing them in braces. If a line ends while nested in braces or parenthesis, Evolver will ask for more input. It will also ask for more if the line ends with certain tokens (such as '+') that cannot legally end a command. Unclosed quotes will also ask for more, and embedded newlines will be omitted. Explicit continuation to the next line may be indicated by ending a line with a backslash (linesplicing). You may want to use the `read` command to read long commands from a file.

Successfully parsed commands are saved in a history list, up to 100 commands. They may be accessed with `!!` for the last command or `!`*string* for the latest command with matching initial string. `!`*number* will repeat a command by number. The command will be echoed. The saved history list may be printed with the `history` command.

Some single-letter commands require interactive input. For those, there are equivalent commands listed below that take input information as part of the command. This is so commands may be read from a script without having to put input data on additional lines after the command, although that can still be done for the single-letter versions.

General note: Some commands will prompt you for a value. A null response (just `RETURN`) will leave the old value unchanged and return you to the command prompt. On options where a zero value is significant, the zero must be explicitly entered. Commands that need a real value will accept an arbitrary expression.

Many commands that change the surface or change the model will cause energies and volumes to be recalculated. If you suspect a command has not done this, the `recalc` command will recalculate everything. It will also update any automatic display.

In the following command syntax description, keywords are shown in upper case, although case is irrelevant in actual commands, except for single-letter commands. Square brackets enclose optional parts of commands.

§6.6. General control structures.

§6.6.1. Command separator.

command ; *command* ...

Several commands on the same line may be separated by a semicolon. Semicolons also are needed to separate commands inside compound commands. A semicolon is not needed after the last command. Example:

```
g 10; r; g 10; u
```

§6.6.2. Compound commands.

{*command* ; ... }

Curly braces group a list of commands into one command. This is useful in the various control structures. A semicolon is necessary after a } if there is a following command (note this is different from the C language). Do not use a semicolon after the first command in an IF THEN ELSE command. An empty compound command {} is legal.

§6.6.3. Command repetition.

command expr

For execution of a command a number of times. Be sure to leave a space between a single-letter command and the expression lest your command be interpreted as one identifier. To avoid several types of confusion, only certain types of commands are repeatable:

1. single letter commands that don't have optional arguments (l,t,j,m,n,w,P,M,G have optional arguments)
2. command list in braces
3. user-defined procedure names
4. redefined single letter commands

§6.6.4. Piping command output.

command | *stringexpr*

The output of the command is piped to a system command. The *stringexpr* needs to be a quoted string or a string variable. It is interpreted as a system command. On systems (such as DOS, Windows, and Macs) without true piping, the string is interpreted as a filename, so output goes directly to a file instead of a command. In such a case, if the filename begins with '+', the output will be appended to the file, rather than replacing it.

Examples:

```
list facets | "more"
list vertices | "tee vlist" ; g 10
list edges | "cat >edgefile"
```

§6.6.5. Redirecting command output.

command >> *stringexpr*

command >>> *stringexpr*

The output of the command is redirected to a file, appending with ¿¿ and overwriting with ¿¿¿. The *stringexpr* needs to be a quoted string or a string variable.

Redirection with '>' to replace current contents is not available due to the use of '>' as an comparison operator.

Examples:

```
{{g 10; u}  15 } >> "logfile"
list vertices >>> "vlist.txt"
```

§6.6.6. Flow of control.

`IF` *expr* `THEN` *command* [`ELSE` *command*]

For conditional execution of commands. *expr* is true if nonzero. Do not use a semicolon to end the first command. Example:

```
if max(edges,length) > 0.02 then {r; g 100 } else g 4
```

`WHILE` *expr* `DO` *command*

`DO` *command* `WHILE` *expr*

For iterated execution of command controlled by a logical expression. Expression is true if nonzero. Example:

```
while max(edges,length) > 0.02 do {r; {g 40; u} 5 }
```

`BREAK`

Exits the innermost current loop. Note: Commands with repetition counts do not qualify as loops.

`BREAK` *n*

Exits the innermost *n* loops. Note: Commands with repetition counts do not qualify as loops.

`CONTINUE`

Skips the rest of the body of the current loop, and goes to the next iteration. Note: Commands with repetition counts do not qualify as loops.

`CONTINUE` *n*

Exits the innermost *n-1* loops, and skips to the generator of the *n*th innermost loop. Note: Commands with repetition counts do not qualify as loops.

`RETURN`

Exits the current command. If the current command is a user-defined command called by another command, the parent command continues. This is essentially a return from a subroutine.

§6.7. Expressions.

Expressions are of two types: numeric and string. String expressions are either quoted strings or are created by the SPRINTF command. Numeric expressions are always floating-point, not integer. Boolean values for conditions are also floating point, 0 for false, nonzero for true (1 for true result of booleanoperation). Numeric expressions are given in algebraic notation with the following terms and operators (in precedence order):

Values:

constant	an explicit number: integer, fixed point, or scientific notation
identifier	variable (listed by `A` command)
`G`	current gravitational constant
`E, PI`	special constants

Element attributes:

`X1,X2,...`	coordinates of vertices, components of edge vector or facet normal
`X,Y,Z,W`	same as `X1,X2,X3,X4`
`P1,P2`	parameters for boundaries
`ID`	element identifying number
`ORIGINAL`	number of parent datafile element
`TAG`	facet tag number from datafile
`COLOR`	integer representing color of facet (color of front if back different) or edge
`FRONTCOLOR`	color of front of facet
`BACKCOLOR`	color of back of facet
`VALENCE`	number of edges on a vertex, facets on edge, edges on a facet, or facets on a body
`AREA`	area of facet
`LENGTH`	length of edge
`VOLUME`	actual volume of body
`TARGET`	target fixed volume of body
`VOLFIXED`	read-only, 1 if body volume fixed, 0 if not.
`VOLCONST`	constant added to calculated volume of body
`DENSITY`	density of edge, facet, or body
`DIHEDRAL`	dihedral angle of facets on an edge (soapfilm model) or edges at a vertex (string model)
`ORIENTATION`	sign for oriented integrals of edges or facets
`ON_CONSTRAINT` *n*	test if element on given constraint
`HIT_CONSTRAINT` *n*	test if a vertex on a one-sided constraint has hit the constraint
`ON_BOUNDARY` *n*	test if element on given boundary
`WRAP`	numerical edge wrap in torus model or other quotient space
`MIDV`	in the quadratic model, the midpoint of an edge.
quantity_name	contribution to a named quantity of an element
extra_attribute[expr]	name of user-defined extra attribute, with subscript if needed.
`TETRA_POINT`	For telling Evolver six films meet at this vertex.
`TRIPLE_POINT`	For telling Evolver three films meet at this vertex.
`vertexnormal[`*n*`]`	components of a normal vector at a vertex.

operators:

`()`	grouping and functional notation
`^`	raise to real power
`**`	raise to real power
`*,/,%,mod,imod,idiv`	arithmetic
`+,-`	arithmetic
`==,>,<,<=,>=,!=`	comparison
`NOT, !`	logical NOT
`AND, &&`	logical AND
`OR, ||`	logical OR
`? :`	conditional expression, as in C language

functions: `SQR, SQRT, SIN, COS, TAN, ACOS, SINH, COSH, ASIN, ATAN, ATAN2(y,x), LOG, EXP, ABS, FLOOR, CEIL TANH, ASINH, ACOSH, ATANH, POW, MAXIMUM(a,b), MINIMUM(a,b)`

some internal read-only values:

`clock`	elapsed time in seconds since starting Evolver
`datafilename`	string containing name of current datafile
`vertex_count`	number of vertices
`edge_count`	number of edges
`facet_count`	number of facets

`body_count`	number of bodies
`facetedge_count`	number of facetedges
`total_time`	elapsed time in the form of total scale factors
`total_energy`	total energy of the surface
`total_area`	total area of the surface (film model)
`total_length`	total length of the surface (string model)
`space_dimension`	dimension of ambient space
`surface_dimension`	dimension of surface
`torus`	whether torus domain (Boolean)
`torus_filled`	whether `torus_filled` specified in effect (Boolean)
`torus_periods`[*expr*] [*expr*]	torus period vectors, as in datafile top section; 1-based indexes
`inverse_periods`[*expr*] [*expr*]	inverse of the `torus_periods` matrix; 1-based indexes
`symmetry_group`	whether any symmetry active (Boolean)
`simplex_representation`	whether facets are represented as simplices (Boolean)
`iteration_counter`	value of index of current iteration loop
`equi_count`	number of edges flipped in last equiangulation
`delete_count`	number of deletions in last delete command
`notch_count`	number of edges notched in last notch command
`refine_count`	number of edges or facets refined in last 'refine edges' or 'refine facets' command
`dissolve_count`	number of items dissolved in last dissolve command
`pop_count`	number of items popped in last '`o`' or '`O`' command
`where_count`	number of items satisfying last `where` clause
`transform_count`	number of image transformations active
`random`	random number between 0 and 1
`last_eigenvalue`	eigenvalue from last `saddle`, `ritz`, or `eigenprobe` command.
`eigenpos`	number of positive eigenvalues in last Hessian factoring.
`eigenneg`	number of negative eigenvalues in last Hessian factoring.
`eigenzero`	number of zero eigenvalues in last Hessian factoring.
`last_hessian_scale`	stepsize from last `saddle` or `hessian_seek` command.
`total` *quantity_name*	value of a named quantity
quantity_name`.value`	value of a named quantity
quantity_name`.pressure`	Lagrange multiplier for a constrained named quantity.
`memory_arena`	Total memory allocated to the program's heap. SGI and Win32 versions only.
`memory_used`	Total memory used in the program's heap. SGI and Win32 versions only.

some internal read-write values:

`ambient_pressure_value`	Value of the external pressure in the ideal gas model.
`random_seed`	seed for random number generator. Defaults to 1 at start of datafile.
`constraint_tolerance`	constraint value regarded as equivalent to zero.
`target_tolerance`	Default value of fixed quantity error tolerance, defaults to 0.0001.
`gravity_constant`	value of the gravitational constant; also set by the `G` command.
`hessian_epsilon`	magnitude regarded as zero by `hessian` command.
`hessian_slant_cutoff`	Makes hessian commands treat vertices whose normal vector is nearly perpendicular to constraints as fixed. hessian_slant_cutoff is the cosine of angle. Works on vertices with one degree of freedom in hessian_normal mode.
`integral_order`	order of polynomial done by 1D and 2D Gaussian integration Much better to set 1D and 2D separately.
`integral_order_1d`	order of polynomial done by 1D Gaussian integration
`integral_order_2d`	order of polynomial done by 2D Gaussian integration
`jiggle_temperature`	current temperature for jiggling.
`lagrange_order`	Order of Lagrange model. Set with the `lagrange` command.
`last_error`	Number of last error message.

`scale`	current scale factor
`thickness`	thickness to separate facet sides of different colors when doing 3D graphics, to prevent weird stippling effects
quantity_name`.target`	constrained value of a named quantity
quantity_name`.modulus`	modulus of a named quantity
`random_seed`	random number generator seed
`pickvnum`	number of last vertex picked in geomview
`pickenum`	number of last vertex picked in geomview
`pickfnum`	number of last vertex picked in geomview
`brightness`	median gray level used in PostScript output and screen graphics.
`background`	background color used on certain screen graphics (xgraph, at the moment).
`linear_metric_mix`	fraction of linear interpolation in Hessian metric, as opposed to vertex weighting.
`quadratic_metric_mix`	fraction of quadratice interpolation in Hessian metric in the quadratic model. (Rather useless to change from the default value of one.)

and some miscellaneous functions:

`sizeof(`*name* `)`	Dimension of the extra attribute *name* (which is not in quotes).
`is_defined` *string*	Returns 1 if the identifier in *string* is known to the Evolver, as keyword or variable name or quantity name or whatever, 0 if not.

Also any toggle command may be used as a Boolean variable in an expression (full word toggles, not single letters). But beware the ambiguity in interpreting a toggle as a command or a value. You may have to force the toggle to be interpreted as a value. `ad := autodisplay` sets `ad` as a command synonym for `autodisplay`, while `ad := (autodisplay)` records the current boolean value.

NOTES: A '+' or '-' preceded by whitespace and followed by a number is taken to be a signed number. Thus "3 - 5" and "3-5" are single expressions, but "3 -5" is not. This is for convenience in separating multiple expressions listed on the same line in the datafile.

The boolean AND and OR operators use short-circuit evaluation; i.e. the second operand is evaluated only if necessary.

The mod operator '%' does a real modulus operation:

$$x\%y = x - floor(x/y) * y,$$

but it works fine on integer values. The integer operator `idiv` rounds its operands toward zero before doing integer division (as implemented in C). `imod` rounds its operands down:

$$x \text{ imod } y = floor(x) - floor(floor(x)/floor(y)) * floor(y).$$

Elements inherit the original number of their parent element from the datafile; this can be referred to as `original`. New vertices and edges that subdivide facets have an `original` value of -1.

Coordinates for edges are components of the edge vector. Coordinates for a facet are the components of its normal vector, whose length is the facet area (valid in 3D only). Note that these edge and normal components are valid only in commands. X, Y, Z appearing in the datafile, say in constraint or quantity integrals, refer to space coordinates.

The wrap number for an edge in a quotient space such as a torus is the internal numerical representation of the wrap, as defined by the writer of the quotient group. For the torus, the current encoding is four bits per dimension, with 1 for '+' wrap and 7 for '-' wrap, low dimension in low bits.

¡a name="stringexpr"¿¡h2¿String expressions¡/h2¿¡/a¿ A string expression evaluates to a string of characters. At present, the only ways to produce strings are:

double-quoted string literals, e.g. `"this is a string"`. The following standard C escape sequences are recognized:

\ `n` newline

\ `r` carriage return

\ `t` tab

\ `b` backspace

\ `q` double-quote mark

\ *c* the character 'c' elsewise In DOS, MS-Windows, or Windows NT paths, use / as the directory separator, since backslash is an escape character. DOS and Windows have always accepted / as a directory separator.

string variables, either internal like `datafilename`, or user-defined.

output from `sprintf`.

§6.8. Element generators.

One feature different from ordinary C is the presence of generators of geometric elements. These occur wherever an element type (vertices, edges, facets, bodies, singular or plural) appears in a command. Attributes of the generated element may be used later in the command. The general form of a generator is

elementtype [*name*] [`WHERE` *condition*]

The generated element may be named (useful in nested iterations), in which case the name must be used when the element is referenced. Element types used as attributes will just generate the elements associated with the parent element, so if ff is a facet, then ff.vertex will generate its three vertices. Currently implemented subelements are vv.edge, vv.facet, ee.vertex, ee.facet, ff.vertex, ff.edge, ff.body, and bb.facet, where vv, ee, ff, and bb are the are names of vertices, edges, facets, and bodies respectively. But be sure to remember that in a nested iteration, an unqualified element type generates all elements of that type, not just those associated with the parent element. Also, each generator can take a "where" clause to limit the elements included. Example:

```
list facet where color == red
```

```
foreach edge ee where ee.length < .3 do list ee.vertex
```

The internal facet-edge structures can also be generated with the element type facetedge.

Indexed element generators. An element generator may carry an index to refer to just one of the elements. The index is in square brackets. Examples:

```
list vertex[3]
```

```
print edge[2].vertex[1].id
```

Indexing an element type directly generates just the one element, without a linear search through all the elements of that type. However, indexing a subelement does result in a linear search through the subelements. Indexing starts at 1.

§6.9. Aggregate expressions.

The maximum, minimum, sum, and average of an expression over a set of elements may be done with aggregate expressions. The `histogram` and `loghistogram` commands have the same form. The general form is

aggregate(*generator*, *expr*)

where *aggregate* may be `max`, `min`, `sum`, or `avg`. The generator generates elements as discussed above. Max and min over logical expressions may be used for logical any and all. The max of an empty set is large negative, and the min of an empty set is large positive, so beware when using WHERE clauses..

Example: This displays facet 4 and all its neighboring facets:

```
show facets ff where max(ff.edge ee,max(ee.facet fff, fff.id == 4))
```

§6.10. Single-letter commands.

The oldest and most commonly used commands are just single letters. Case is significant for these. Single letters are always interpreted as commands, so you may not use single letters for variable names.

It is possible to reassign a single letter to your own command by the syntax

letter `:::=` *command*

but this should only be used in special circumstances, such as redefining `r` to do additional actions along with refinement. The old meaning can be restored with a null assignment, "*letter* :::=". Use single quotes around the letter to get the old meaning, i.e. `'r'` will do a standard refine when `r` has been redefined. Redefinitions are cleared when a new surface is loaded. Be careful when using redefined commands in defining other commands. Redefinition is effective on execution of the redefinition command, not on parsing. Redefinition is not retroactive to uses in previously defined commands.

§6.10.1. Single-letter command summary.

There are five groups of commands:

Reporting:

`C` Run consistency checks.

`c` Report count of elements.

`e` Extrapolate.

`i` Information on status.

`v` Report volumes.

`X` List element extra attributes.

`z` Do curvature test.

Model characteristics:

`A` Set adjustable constants.

`a` Toggle area normalization

`b` Set body pressures.

`f` Set diffusion constant.

`G` Set gravity.

`J` Toggle jiggling on every move.

`k` Set boundary gap constant.

`M` Toggle linear/quadratic model.

`m` Toggle fixed motion scale.

`p` Set ambient pressure.

`Q` Report or set quantities.

`U` Toggle conjugate gradient method.

`W` Homothety toggle.

Surface modification:

`g` Go one iteration step. Often followed by a repetition count.

`j` Jiggle once.

`K` Skinny triangle long edge divide.

`l` Subdivide long edges.

`N` Set target volumes to actual.

`n` Notch ridges and valleys.

`O` Pop non-minimal edges.

`o` Pop non-minimal vertices.

`r` Refine triangulation.

`t` Remove tiny edges.

`u` Equiangulate.

`V` Vertex averaging.

`w` Weed out small triangles.

`y` Torus duplication.

`Z` Zoom in on vertex.

Output:

`D` Toggle display every iteration.

`d` Dump surface to datafile.

`P` Create graphics display file.

`s` Show triangulation graphically.

Miscellaneous:

`F` Toggle command logging.

`H,h,?` Help screen.

`q,x` Exit.

The `G`, `j`, `l`,`m`,`n`,`t`, and `w` commands require real values, which may be entered on the same line, or given in response to a prompt if not.

§6.10.2. Alphabetical single-letter command reference.

`A` Adjustable values. Displays current values and allows you to enter new values. New value is entered as the number of the constant (from the list) and the new value. Values of named quantities, their moduli, and the target values of fixed named quantities also appear here. The modulus and target value may be changed. Only explicitly user-defined named quantities appear here, unless show_all_quantities is toggled on.

`a` Toggles area normalization of vertex forces and other gradient. Be sure you have a small enough scale factor, or else things tend to blow up. Reduce scale factor temporarily after refinement, since triangle areas are cut by a factor of 4 but the old creases remain. When this option is ON, there is an optional check that can be made for facets that move too much. This is done by computing the ratio of the length of the normal change to the length of the old normal. If this exceeds the user-specified value, then all vertices are restored to their previous position. The user should reduce the motion scale factor and iterate again.

`b` Permits user to change body prescribed volumes or pressures. Prints old value for each body and prompts for new.

`C` Runs various internal consistency checks. If no problems, just prints "Checks completed." The checks are:

Element list integrity - checks that data structures are intact.

Facet-edge check - that if a facet adjoins an edge, then the edge adjoins the facet, and that the three edges around a facet link up.

Facet-body check - whether adjacent facets have the same body on the same side.

Collapsed elements - check if endpoints of an edge are the same, and whether neighboring facets share more than one edge and two vertices.

`c` Prints count of elements and memory used (just for element structures, not everything) and prints various model parameters. Synonym: counts.

`D` Toggles displaying every iteration. Default is ON.

`d` Dumps data to ASCII file in same format as initial data file. You will be prompted for a filename. An empty reponse will use the default dump name, which is the datafile name with a ".dmp" extension. Useful for checking your input is being read correctly, for saving current configuration, and for debugging.

`e` Extrapolates total energy to infinite refinement if at least two refinements have been done. Uses last energy values at three successive levels of refinement. Synonym: extrapolate.

`F` Toggle logging of commands in file. If starting logging, you will be prompted for the name of a log file. Any existing file of that name will be overwritten. Logging stops automatically when the surface is exited. Only commands that change the surface are logged.

`f` Set diffusion constant. Prints old and prompts for new.

`G` Toggles gravity on or off. Gravity starts ON if any body has `DENSITY`; otherwise OFF. If followed by a value, sets gravity to that value. Otherwise prints old value of gravitational constant and prompts for new. Optionally takes new gravity value on command line.

`g` Do iteration step. Each iteration calculates the force on each vertex due to its contribution to the total energy and moves the vertex by a multiple of the force. There is a global scale factor that multiplies the force to give the displacement. If area normaliztion is turned on, the force at each vertex is also divided by the total area of facets adjacent to the vertex to better approximate motion by mean curvature (but this seems to be often numerically badly behaved due to long skinny facets). If any bodies have prescribed volumes, the vertices are also displaced to bring the volumes back to near the prescribed values.

If scale optimizing (see command `m`) is ON, then different global scale values will be tried until a quadratic interpolation can be done to find the optimal value. (This can blow up under certain circumstances.) The scale factor used for motion is then multipied by the `scale_scale` variable (default value 1).

The output consists of the number of iterations left (for people who wonder how close their 1000 iterations are to ending), the area and energy, and the scale factor. The user can abort repeated iterations by sending an interrupt to the process (SIGINT, to be precise; CTRL-C or whatever on your keyboard).

`h,H,?` Prints help screen listing these commands.

`i` Prints miscellaneous information:

Total energy

Total area of facets

Count of elements and memory usage

Area normalization, if on

LINEAR or QUADRATIC model

Whether conjugate gradient on

Order of numerical integration

Scale factor value and option (fixed or optimizing)

Diffusion option and diffusion constant value

Gravity option and gravitational constant value

Jiggling status and temperature

Gap constant (for gap energy, if active)

Ambient pressure (if ideal gas model in effect)

`J` Toggles jiggling on every iteration. If jiggling gets turned on, prompts for temperature value.

`j` Jiggles all vertices once. Useful for shaking up surfaces that get in a rut, especially crystalline integrands. You will be prompted for a "temperature" which is used as a scaling factor, if you don't give a temperature with the command. The actual jiggle is a random displacement of each vertex independently with a Gaussian distribution. See the `longj` command below for a user-definable perturbation.

`K` Finds skinny triangles whose smallest angle is less than a specified cutoff. You will be prompted for a value. Such triangles will have their longest edge subdivided. Should be followed with tiny edge removal (`t`) and equiangulation (`u`).

`k` Sets "gap constant" for gap energy for `CONVEX` boundaries. Adds energy roughly proportional to area between edge and boundary. You will be prompted for a value. Normal values are on the order of magnitude of unity. Value k = 1 is closest to true area. Use 0 to eliminate the energy.

`l` Subdivides long edges, creating new facets as necessary. You will be prompted for a cutoff edge length, if you don't give a value with the command. Existing edges longer than the cutoff will be divided once only. Newly created edges will not be divided. Hence there may be some long edges left afterward. If you enter `h`, you will get a histogram of edge lengths. If you hit RETURN with no value, nothing will be done. It is much better to refine than to subdivide all edges. A synonym for "`l` *value*" is "`edge_divide` *value*".

`M` Sets model type to `LINEAR`, `QUADRATIC`, or `LAGRANGE`. Changing from `LINEAR` to `QUADRATIC` adds vertices at the midpoints of each edge. Changing from `QUADRATIC` to `LINEAR` deletes the midpoints. Likewise for Lagrange. Optionally takes new model type (1 (linear), 2 (quadratic), or ¿ 2 (Lagrange of given order)) on command line.

`m` Toggles quadratic search for optimal global motion scale factor. If search is toggled OFF, you will be prompted for a scale factor. If you give a value with the command, then you are setting a fixed scale factor. Values around 0.2 work well when the triangulation is well-behaved and area normalization (command `a`) is off. In optimizing mode, a scale factor getting small, say below 0.01, indicates triangulation problems. Too large a scale factor will show up as total energy increasing. If you have motion by area normalization ON (command `a`), use a small scale factor, like 0.001, until you get a feel for what works.

`N` Normalize body prescribed volumes to current actual volumes.

`n` Notching ridges and valleys. Finds edges that have two adjacent facets, and those facets' normals make an angle greater than some cutoff angle. You will be prompted for the cutoff angle (radians) if you don't give a value with the command. Qualifying edges will have the adjacent facets subdivided by putting a new vertex in the center. Should follow with equiangulation. In the string model, it will refine edges next to vertices with angle between edges (parallel orientation) exceeding the given value. Optionally takes cutoff angle on command line.

`O` Pop non-minimal edges. Scans for edges with more than three facets attached. Splits such edges into triple-facet edges. Splits propagate along a multiple edge until they run into some obstacle. Try `octa.fe` for an example.

o Pop non-minimal vertices. This command scans the surface for vertices that don't have the topologies of one of the three minimal tangent cones. These are "popped" to proper local topologies. Only vertices in the interior of a surface, not fixed or on constraints or boundaries, are tested. This command tends to create a lot of small edges and skinny triangles. Try `popstr.fe` and `octa.fe` for examples.

P Produce graphics output files. The view is the same as seen with the `s` command. Several formats are currently available, and a menu will appear. These are now Pixar, geomview, PostScript, SoftImage, and a personal format called Triangle. You may optionally give the menu item number on the command line. If you are doing a torus surface, you will be asked for a display option, for which see the **Torus** section. You will be prompted for a filename. For Pixar format, you will be asked whether vertices should have normal vectors for normal interpolation (calculated as the average normal of all facets around a vertex); whether inner facets (adjacent to two bodies) outer facets (adjacent to zero or one body), or all facets are to be plotted; and whether different colors should be used for different bodies. If so, you are asked for the name of a file with a colormap in the format of RGB values, one set per line, values between 0 and 1. (This map may not be effective on all devices.)

You may also be asked if you want thickening. If you do, each facet will be recorded twice, with opposite orientations, with vertices moved from their original positions by the thickening distance (which the option lets you enter) in the normal direction. The normal used at each vertex is the same as used for normal interpolation, so all the facets around a planar vertex will have that vertex moved the same amount. Triple junctions will be separated. Thickening is good for rendering programs that insist on consistently oriented surfaces, or that can't see the backside of a surface. The default thickening distance is one one-thousandth of the diameter of the object.

For file formats, see the section on Graphics file formats.

For those of you that have `geomview`, the relevant commands are here. `Geomview` uses a pipe interface at the moment. Besides options for starting and stopping simultaneous `geomview`, there are options for starting a named pipe without invoking `geomview`. You will be told the name of the pipe, and it is up to you to start a pipe reader. Evolver blocks until a pipe reader is started. This is useful for having a second instance of Evolver feed a second surface to `geomview` by having `geomview` load the pipe. Commands in the geomview command language may be sent with the command `GEOMVIEW` *string*.

This command can be followed by a number to pick a menu option without displaying the menu. Thus `P 8` starts geomview.

See also the GEOMVIEW and POSTSCRIPT commands.

p Sets ambient pressure in ideal gas model. A large value gives more incompressible bodies.

Q Report current values and optionally change target values of user-defined quantities.

q Exit program. You will be given a chance to have second thoughts. You may also load a new datafile. Automatically closes graphics if you really quit. Does not save anything.

r Refines the triangulation. Edges are divided in two, and `SOAPFILM` facets are divided into four facets with inherited attributes. Reports the number of structures and amount of memory used.

s Shows the surface on the screen. The proper display routines must have been linked in for the machine the display is on. Goes into the graphics command mode (see below). Torus surfaces have display options you will be asked for the first time, for which see the **Torus** section. The graphics window may be closed with the `CLOSE_SHOW` command.

`t` Eliminates tiny edges and their adjacent facets. You will be prompted for a cutoff edge length if you don't give a value with the command. If you enter `h`, you will get an edge length histogram. If you hit `RETURN` without a value, nothing will happen. Some edges may not be eliminable due to being `FIXED` or endpoints having different attrtibutes from the edge.

`U` This toggles conjugate gradient mode, which will usually give faster convergence to the minimum energy than the default gradient descent mode. The only difference is that motion is along the conjugate gradient direction. The scale factor should be in optimizing mode. The history vector is reset after every surface modification, such as refinement or equiangulation. After large changes (say, to volume), run without conjugate gradient a few steps to restore sanity.

`u` This command, called "equiangulation", tries to polish up the triangulation. In the soapfilm model, each edge that has two neighboring facets (and hence is the diagonal of a quadrilateral) is tested to see if switching the quadrilateral diagonal would make the triangles more equiangular. For a plane triangulation, a fully equiangulated triangulation is a Delaunay triangulation, but the test makes sense for skew quadrilaterals in 3-space also. It may be necessary to repeat the command several times to get complete equiangulation. The tt edgeswap command can force flipping of prescribed edges. In the simplex model, equiangulation works only for surface dimension 3. There, two types of move are available when a face of a tetrahedron violates the Delaunay void condition: replacing two tetrahedra with a common face by three, or the reverse operation of replacing three tetrahedra around a common edge by two, depending on how the condition is violated. This command does not work in the string model.

`V` Vertex averaging. For each vertex, computes new position as area-weighted average of centroids of adjacent facets. Only adjacent facets with the same constraints and boundaries are used. Preserves volumes, at least to first order. See the `rawv` command below for vertex averaging without volume preservation, and `rawestv` for ignoring likeness of constraints. Does not move vertices on triple edges or fixed vertices. Vertices on constraints are projected back onto their constraints. All new positions are calculated before moving. For sequential calculation and motion, see the `vertex_average` command.

`v` Shows prescribed volume, actual volume, and pressure of each body. Also shows named quantities. Only explicitly user-defined named quantities are shown, unless show_all_quantities has been toggled on. Pressures are really the Lagrange multipliers. Pressures are computed before an iteration, so the reported values are essentially are one iteration behind. Synonym: `show_vol`.

`W` Toggles homothety. If homothety ON, then after every iteration, the surface is scaled up so that the total volume of all bodies is 1. Meant to be used on surfaces without any constraints of any kind, to see the limiting shape as surface collapses to a point.

`w` Tries to weed out small triangles. You will be prompted for the cutoff area value if you don't give a value with the command. If you enter `h`, you will get a histogram of areas to guide you. If you hit `RETURN` with no value, nothing will be done. Some small triangles may not be eliminable due to constraints or other such obstacles. The action is to eliminate an edge on the triangle, eliminating several facets in the process. Edges will be tried for elimination in shortest to longest order. WARNING: Although checks are made to see if it is reasonable to eliminate an edge, it is predicated on facets being relatively small. If you tell it to eliminate all below area 5, Evolver may eliminate your entire surface without compunction.

`X` List the current element extra attributes, including name, size, and type. Some internal attributes are also listed, beginning with double underscore.

x Exit program. You will be given a chance to have second thoughts. You may also load a new datafile. Automatically closes graphics if you really quit. Does not save anything.

y Torus duplication. In torus model, prompts for a period number (1,2, or 3) and then doubles the torus unit cell in that direction. Useful for extending simple configurations into more extensive ones.

Z Zooms in on a vertex. Asks for vertex number and radius. Number is as given in vertex list in datafile. Beware that vertex numbers change in a dump (but correct current zoom vertex number will be recorded in dump). Eliminates all elements outside radius distance from vertex 1. New edges at the radius are made `FIXED`. Meant to investigate tangent cones and intricate behavior, for example, where wire goes through surface in the overhand knot surface. Zooming is only implemented for surfaces without bodies.

z Do curvature test on `QUADRATIC` model. Supposed to be useful if you're seeking a surface with monotone mean curvature. Currently checks angle of creases along edges and samples curvature on facet interiors. Orientation is with respect the way facets were originally defined.

§6.11. General commands.

§6.11.1. SQL-type queries on sets of elements.

These commands generally operate on a set of elements defined by an element generator.

`FOREACH` *generator* `DO` *command*

Repeat a command for each element generated by the generator. Examples:

```
foreach vertex do print x^2 + y^2 + z^2
foreach edge ee where ee.dihedral > .4 do \
{  printf "id %g\n",id; foreach ee.vertex do printf " %g %g %g\n",x,y,z; }
```

`LIST` *generator*

List elements on the screen in the same format as in the datafile. Piping to `more` can be used for long displays. Examples:

```
list vertices | "more"
list vertices where x < 1 and y > 2 and z >= 3 | "tee vfile"
list edges where id == 12
```

`REFINE` *generator*

Subdivides edges by putting a vertex in the middle of each edge and splitting neighboring facets in two. It is the same action as the long edge subdivide command (command `l`). Facets will be subdivided by putting a vertex in the center and creating edges out to the old vertices. It is strongly suggested that you follow this with equiangulation to nicen up the triangulation. Edge refinement is better than facet refinement as facet refinement can leave long edges even after equiangulation. Example:

```
refine edges where not fixed and length > .1
```

`DELETE` *generator*

Deletes edges by shrinking the edge to zero length (as in the tiny edge weed command `t`) and facets by eliminating one edge of the facet. Facet edges will be tried for elimination in shortest to longest order. Edges will not be deleted if both endpoints are fixed, or both endpoints have different constraints or boundaries from the edge. Example:

```
delete facets where area < 0.0001
```

`DISSOLVE` *generator*

Removes elements from the surface without closing the gap left. The effect is the same as if the line for the element were erased from a datafile. Hence no element can be dissolved that is used by a higher dimensional element. (There are two exceptions: dissolving an edge on a facet in the string model, and dissolving a facet on a body in the soapfilm model.) Thus "`dissolve edges; dissolve vertices`" is safe because only unused edges and vertices will be dissolved. No error messages are generated by doing this. Good for poking holes in a surface.

```
dissolve facets where original == 2
```

`EDGESWAP` *edge_generator*

If any of the qualifying edges are diagonals of quadrilaterals, they are flipped in the same way as in equiangulation, regardless of whether equiangularity is improved. "`edgeswap edge`" will try to swap all edges, and is not recommended, unless you like weird things.

```
edgeswap edge[22]
edgeswap edge where color == red
```

`FIX` *generator* [*name*] [`WHERE` *expr*]

Sets the FIXED attribute for qualifying elements. Example:

```
fix vertices where on_constraint 2
```

`FIX` *variable*

Converts the variable to a non-optimizing parameter.

`FIX` *named quantity*

Converts an info_only named quantity to a fixed quantity constraint.

`SET` *generator* [*name*] *attrib expr* [`WHERE` *expr*]

`SET` *generator.attrib expr* [`WHERE` *expr*]

Sets an attribute to the indicated value for qualifying elements. The new value expression may refer to the element in question. The second form permits the use of '.' for people who use '.' instinctively in field names. `SET` can change the following attributes: constraint, coordinates, density, orientation, non-global named quantity or named method, user-defined extra attributes, body target volume, body volconst, fixed, pressure, color, frontcolor, backcolor, and opacity (for the appropriate type elements). Fixed, named quantity, and named method attributes are just toggled on; they do not need the first *expr*. For constraint, the *expr* is the constraint number. For color, the color value is an integer. The integers from 0 to 15 can be referred to by the predefined names `BLACK`, `BLUE`, `GREEN`, `CYAN`, `RED`, `MAGENTA`, `BROWN`, `LIGHTGRAY`, `DARKGRAY`, `LIGHTBLUE`, `LIGHTGREEN`, `LIGHTCYAN`, `LIGHTRED`, `LIGHTMAGENTA`, `YELLOW`, and `WHITE`. The special color value `CLEAR` (-1) makes a facet transparent. Edge colors may not show up in `geomview` unless you do '`show edges where 1`' to have Evolver explicitly feed edges to `geomview` instead of letting `geomview` just show facet edges on its own. On certain systems (Silicon Graphics and Pixar at present) the opacity value sets the transparency of ALL facets. A value of 0.0 corresponds to transparent, and a value of 1.0 to opaque. Examples:

```
set facets density 0.3 where original == 2
set vertices x 3*x where id < 5 // multiplies x coordinate by 3
set body target 5 where id == 1 // sets body 1 target volume to 5
set vertices constraint 1 where id == 4
set facet color clear where original < 5
```

`SET` *name attrib expr*

Inside a FOREACH loop with a named element, one can set an attribute for the named element. A '.' may be used as in sl name.attrib by people who like that sort of thing.

`SET` *quantityname attrib expr*

`SET` *instancename attrib expr*

Set the value of a named quantity or named method instance attribute. For a named quantity, the settable attributes are target, modulus, volconst, and tolerance. For a named method instance, only modulus. There is no implicit reference to the quantity in the expression, so say

```
set myquant target myquant.value
```

rather than `set myquant target value`.

`SET BACKGROUND` *color*

Sets the backgound color for native graphics (not geomview). The color is an integer 0-15 or a color name listed in the general `SET` command below. Example:

```
set background black
```

`UNFIX` *elements* [*name*] [`WHERE` *expr*]

Removes the FIXED attribute for qualifying elements. Example:

```
unfix vertices where on_constraint 2
```

`UNFIX` *variable*

Converts the variable to an optimizing parameter.

`UNFIX` *named quantity*

Converts a fixed named quantity to an info_only quantity.

`UNSET` *elements* [*name*] *attrib* [`WHERE` *expr*]

Removes the attribute for qualifying elements. Attributes here are `FIXED`, `VOLUME`, `PRESSURE`, `DENSITY`, non-global named quantities or named methods, `FRONTBODY`, `BACKBODY`, `CONSTRAINT` n, or `BOUNDARY` n. A use for the last is to use a boundary or constraint to define an initial curve or surface, refine to get a decent triangulation, then use "unset vertices boundary 1" and "unset edges boundary 1" to free the curve or surface to evolve. The form "`unset facet bodies ...`" is also available to disassociate given facets from their bodies.

`UNSET` *elements* [*name*] `CONSTRAINT` *expr* [`WHERE` *expr*]

Removes constraint of given number from qualifying elements.

`VERTEX_AVERAGE` *vertexgenerator*

The action is the same as the `V` command, except that each new vertex position is calculated sequentially, instead of simultaneously, and an arbitrary subset of vertices may be specified. Fixed vertices do not move. Examples:

vertex_average vertex[2]
vertex_average vertex where id ¡ 10
vertex_average vertex vv where max(vv.facet,color==red) == 1

`WRAP_VERTEX(`v,w`)`

In a symmetry group model, transforms the coordinates of vertex number n by symmetry group element w and adjusts wraps of adjacent edges accordingly.

Single elements:

The verbs set, unset, list, delete, refine, fix, unfix, dissolve, and vertex_average can be used on single named elements inside an iteration construct. Examples:

```
foreach edge ee where ee.length < .01 do {
printf "Refining edge %g.",ee.id; refine ee; }
```

§6.11.2. Variable assignment.

Values can be assigned to variables. The variable names must be two or more letters, in order they not be confused with single-letter commands. Note that ':=' is used for assignment, not '='. Numeric, string, and command values may be assigned.

The C-style arithmetic assignments `+=`, `-=`, `*=`, and `/=` work. For example, `val += 2` is equivalent to `val := val + 2`. These also work in other assignment situations where I thought they made sense, such as attribute assignment.

identifier `:=` *expr*

identifier `::=` *expr*

Evaluates expression and assigns it to the named variable. If the variable does not exist, it will be created. These are the same class of variables as the adjustable parameters in the datafile, hence are all of global scope and may also be inspected and changed with the 'A' command. If "::=" is used instead of ":=", then the variable becomes permanent and will not be forgotten when a new surface is begun. (Well, that was the intention. But the semantics got tricky, so for now "::=" means the same as ":=".)

Assignment may also be done for single element attributes, named quantity attributes, and named method attributes. Examples:

`counter := 0`

`body[1].density := 8.5`

`vertex[2].x[2] += pi`

`myquant.modulus := 2`

Assignments to variables may be made at the start of expressions, mainly for the purpose of evaluating common subexpressions only once in constraint and quantity integrands. The syntax for such a compound expression is

variable `:=` *expr* , *expr*

The value of the expression is the value of the second expression.

identifier `:=` *command*

identifier `::=` *command*

Makes *identifier* an abbreviation for *command*. Subsequently, *identifier* may be used as a command itself. You are strongly urged to enclose the command on the right side in braces so the parser can tell it's a command and not an expression. Also multiline commands then don't need linesplicing. Examples:

```
gg := {g 10; u }
gg 12
```

`AREA_FIXED :=` *expr*

Sets the target value for the fixed area constraint. Does not activate constraint.

`AUTOCHOP :=` *expr*

Sets autochop length, so edges longer than this will automatically be subdivided each iteration when autochopping is on. Does not toggle autochopping on.

`COLORMAP := "`*filename*`"`

Sets the colormap file name.

`DIFFUSION :=` *expr*

Sets diffusion constant. Does not toggle diffusion on.

`GAP_CONSTANT :=` *expr*

Sets the gap constant for assigning energy to gaps between facets and curved walls. Use 1 for best approximation to area.

`GRAVITY :=` *expr*

Sets gravitational constant. Does not toggle gravity ON if it is OFF.

`THICKEN :=` *expr*

Sets thickness for double layer display. Does not toggle it on.

§6.11.3. Information commands.

`EPRINT`

Function that prints an expression and returns the value. Syntax:

`eprint` *expr*

Meant for debugging; probably an archaic leftover from when the command language was not as developed. Example:

`print sum(facet, eprint area)`

will print out all the facet areas and then the sum.

`HELP`

Prints extract from the HTML documention for keywords, or prints some information on user-defined names. Syntax:

`help` *string*

The string is used as a look-up into the index of all anchor names. If an exact match is found, the extract is printed, up to the next horizontal rule. No formatting is done beyond line breaks and paragraphs, so the output may look a little rough. In any case, the command also prints all anchor names containing the given string, so you should be able to find related references. Note that *string* is not a *stringexpr*, but a literal string. If the anchor name is just one word, the quotes may be omitted from around the string. The HTML files must be on EVOLVERPATH.

`IS_DEFINED`

To find out if a name is already in use as a keyword or user-defined name, use the `is_defined` function, which has the syntax

`is_defined(`*stringexpr*`)`

The *stringexpr* must be a quoted string or other string expression. The return value is 0 if the name is undefined, 1 if defined. This function is evaluated at parse-time, not run-time, so a command like `if is_defined(newvar) then newvar := 1 else newvar := 0` will give `newvar` the value 0 if this is its first appearance.

`LIST ATTRIBUTES`

Prints a list of the "extra attributes" of each type of element. Besides user-defined extra attributes, this list also contains the predefined attributes that make use of the extra attribute mechanism (being of variable size), such as coordinates, parameters, forces, and velocities. It does not list permanent, fixed-size attributes such as color or fixedness, or possible attributes that are not used at all.

`LIST BOTTOMINFO`

Prints what would be dumped in the "`read`" section at the end of a dumpfile: command definitions and various toggle states.

`LIST PROCEDURES`

`LIST TOPINFO`

Prints the first section of the datafile on the screen. This is everything before the vertices section.

`LOGFILE` *stringexpr*

Starts recording all input and output to the file specified by *stringexpr*, which must be a quoted string or a string variable or expression. Appends to an existing file. To end logging, use `logfile off`.

`PRINT` *expr*

Prints the value of the expression. `PRINT` *expr* can also be used inside an expression, where it prints the expression and evaluates to the value of its expression. Will do string expressions. Examples:

```
print datafilename
print max(edge,length)
print max(vertex, print (x^2+y^2+z^2) )
```

`PRINT` *procedure_name*

Prints the command assigned to the named procedure. These are reconstructed from the parse tree, so may not be identical in appearance to the original commands.

`PRINTF` *string*, *expr*, *expr*, . . .

Prints to standard output using the standard C sprintf function. Only string and real-valued numeric expressions are valid, so the format string should use only `%s`, `%f` and `%g` formats. The format string can be a string variable or a quoted string. (Note: former quirks with string arguments have been removed, so they can be used normally.) Example:

```
printf "Datafile:  %s area:  %21.15g\n",datafilename,total_area
```

`SPRINTF` *string*, *expr*, . . .

Prints to a string using the standard C printf function. Otherwise same as `PRINTF`. Can be used wherever a string expression is called for. Examples:

```
dump sprintf "file%g.dmp",count

message := sprintf "Total area is %g\n",total_area
```

`HISTOGRAM(`*element* [`WHERE` *expr1*], *expr2*)

`LOGHISTOGRAM(`*element* [`WHERE` *expr1*], *expr2*)

Prints a histogram of the values of *expr2* for the qualifying elements. The syntax is the same as the aggregate expressions. It uses 20 bins, and finds its own maximum and minimum values, so the user does not have to specify binsize. The log version will lump all zero and negative values into one bin. Example:

```
histogram(edge,dihedral*180/pi)
```

§6.11.4. Action commands.

`ALICE` *expr*

A special private command.

`AREAWEED` *expr*

Weeds out facets smaller than given area. Same as 'w' command, except does not need interactive response.

`BURCHARD` *expr*

A special private command.

`CHDIR` *stringexpr*

Changes the current directory, used for searching for files before EVOLVERPATH is used. In MS-Windows, use a front slash '/' or a double backslash '\\' instead of a single backslash as the path character. Example: `chdir "/usr/smith/project"`

`CLOSE_SHOW`

Closes the native graphics window started by the 's' or `SHOW` commands. Does not affect geomview or OpenGL version. Synonym: show_off.

`DEFINE` It is possible to dynamically define extra attributes for elements, which may be single values or vectors. The definition syntax is

`DEFINE` *elementtype* `ATTRIBUTE` *name* `REAL|INTEGER` [[*dim*]]

where *elementtype* is `vertex, edge, facet,` or `body`, *name* is an identifier of your choice, and [*dim*] is an optional expression for the vector dimension (with the brackets). There is no practical distinction between real and integer types at the moment, since everything is stored internally as reals. But there may be more datatypes added in the future. It is not an error to redefine an attribute that already exists, as long as the definition is the same. Extra attributes are inherited by elements of the same type generated by subdivision. Examples:

```
define edge attribute charlie real
define vertex attribute oldx real[3]
```

The value of an extra attribute can also be calculated by user-supplied code. The attribute definition is followed by the keyword "function" and then the code in brackets. In the code, the keyword "self" is used to refer to the element the attribute is being calculated for. Example: To implement the lowest z value of a facet as an attribute:

```
define facet attribute minz real function
{self.minz := min(self.vertex,z);}
```

These attributes can also be indexed. Due to current parser limitations on parsing executable code, this type of extra attribute definition cannot occur in the top section of the datafile, although the non-function version can to declare the attribute name, and the function part added in a re-definition in the READ section of the datafile.

It is also possible to use DEFINE to declare a variable without giving it an initial value. This is primarily a mechanism to be sure a variable is defined before use, without overwriting an existing value. If the variable is new, the initial value is zero or the null string. Syntax:

`DEFINE` *name* `REAL|INTEGER|STRING`

`DIRICHLET`

Does one iteration of minimizing the Dirichlet integral of the surface. The current surface is the domain, and the Dirichlet integral of the map from the current surface to the next. This is according to a scheme of Konrad Polthier and Ulrich Pinkall [PP]. At minimum Dirichlet integral, the area is minimized also. Works only on area with fixed boundary; no volume constraints or anything else. Seems to converge very slowly near minimum, so not a substitute for other iteration methods. But if you have just a simple soap film far, far from the minimum, then this method can make a big first step. `DIRICHLET_SEEK` will do an energy-minimizing search in the direction. See section in Technical Reference chapter.

`DUMP` *filename*

Dumps current surface to named file in datafile format. The filename can be a quoted string or a string variable or expression. With no filename, dumps to the default dump file, which is the datafile name with ".dmp" extension.

`EDGEWEED` *expr*

Weeds out edges shorter than given value. Same as 't' command, except does not need interactive response.

`EIGENPROBE` *expr*

`EIGENPROBE(`*expr,expr*`)`

Prints the number of eigenvalues of the energy Hessian that are less than, equal to, and greater than the given value. It is OK to use an exact eigenvalue (like 0, often) for the value. Useful for probing stability. Second form will further do inverse power iteration to find an eigenvector. The second argument is the limit on the number of iterations. The eigenvalue will be stored in the `last_eigenvalue` variable, and the eigenvector can be used by the `move` command. The direction of the eigenvector is chosen to be downhill in energy, if the energy gradient is nonzero.

`HESSIAN`

Attempt to minimize energy in one step using Newton's method with the Hessian matrix of the energy. See the Hessian section of the Model chapter, and the Hessian section of the Technical Reference chapter.

`HESSIAN_MENU`

Brings up a menu of experimental stuff involving the Hessian matrix. Not all of it works well, and may disappear in future versions. A one-line prompt with options appears. Use option "?" to get a fuller description of the choices. A quick summary of the current options:

`1. Fill in hessian matrix.`

Allocation and calculation of Hessian matrix.

`2. Fill in right side. (Do 1 first)`

Calculates gradient and constraint values.

`3. Solve. (Do 2 first)`

Solves system for a motion direction.

`4. Move. (Do 3, A, B, C, E, K, or L first)`

Having a motion direction, this will move some stepsize in that direction. Will prompt for stepsize.

`7. Restore original coordinates.`

Will undo any moves. So you can move without fear.

`9. Toggle debugging. (Don't do this!)`

Prints Hessian matrix and right side as they are calculated in other options. Produces copious output, and is meant for development only. Do NOT try this option.

`B. Chebyshev (For Hessian solution ).`

Chebyshev iteration to solve system. This option takes care of its own initialization, so you don't have to do steps 1 and 2 first. Not too useful.

`C. Chebyshev (For most negative eigenvalue eigenvector).`

Chebyshev iteration to find most negative eigenvalue and eigenvector. Will ask for number of iterations, and will prompt for further iterations. End by just saying 0 iterations. Prints Rayleigh quotient every 50 iterations. After finding an eigenpair, gives you the chance to find next lowest. Last eigenvector found becomes motion for step 4. Last eigenvalue is stored in the `last_eigenvalue` variable. Self initializing. Not too useful.

`E. Lowest eigenvalue. (By factoring. Do 1 first)`

Uses factoring to probe the inertia of $H - \lambda I$ until it has the lowest eigenvalue located within .01. Then uses inverse iteration to find eigenpair. The eigenvalue is stored in the `last_eigenvalue` variable.

`F. Lowest eigenvalue. (By conjugate gradient. Do 1 first)`

Uses conjugate gradient to minimize the Rayleigh quotient. The eigenvalue is stored in the `last_eigenvalue` variable.

`L. Lanczos. (Finds eigenvalues near probe value. )`

Uses Lanczos method to solve for 15 eigenvalues near the probe value left over from menu choices 'P' or 'V'. These are approximate eigenvalues, but the first one is usually very accurate. Do not trust apparent multiplicities. From the main command prompt, you can use the `lanczos` *expr* command. The first eigenvalue printed is stored in the `last_eigenvalue` variable.

`R. Lanczos with selective reorthogonalization.`

Same as 'L', but a little more elaborate to cut down on spurious multiplicities by saving some vectors to reorthogonalize the Lanczos vectors. Not quite the same as the official "selective reorthogonalization" found in textbooks. Last eigenvalue is stored in the `last_eigenvalue` variable.

`Z. Ritz subspace iteration for eigenvalues. (Do 1 first)`

Same as "`ritz`" main command. Will prompt for parameters. The first eigenvalue printed is stored in the `last_eigenvalue` variable.

`X. Pick Ritz vector for motion. (Do Z first)`

For moving by the various eigenvectors found in option `Z`. Particularly useful for multiple eigenvalues.

`P. Eigenvalue probe. (By factoring. Do 1 first)`

Reports inertia of $H - \lambda I$ for user-supplied values of λ. The Hessian H includes the effects of constraints. Will prompt repeatedly for λ. Null response exits. From the main command prompt, you can use the `eigenprobe` *expr* command.

`S. Seek along direction. (Do 3, A, B, E, C, K, or L first)`

Can do this instead of option 4 if you want Evolver to seek to lowest energy in an already found direction of motion. Uses the same line search algorithm as the optimizing '`g`' command.

`Y. Toggle YSMP/alternate minimal degree factoring.`

Default Hessian factoring is by Yale Sparse Matrix Package. The alternate is a minimal degree factoring routine of my own devising that is a little more aware of the surface structure, and maybe more efficient. If YSMP gives problems, like running out of storage, try the alternate. This option is available at the main prompt as the `ysmp` toggle.

`U. Toggle Bunch-Kaufman version of min deg.`

YSMP is designed for positive definite matrices, since it doesn't do any pivoting or anything. The alternate minimal degree factoring method, though, has the option of handling negative diagonal elements in a special way. This option is available at the main prompt as the `bunch_kaufman` toggle.

`M. Toggle projecting to global constraints in move.`

Toggles projecting to global constraints, such as volume constraints. Default is ON. Don't mess with this. Actually, I don't remember why I put it in.

`G. Toggle minimizing square gradient in seek.`

For converging to unstable critical points. When this is on, option "`S`" will minimize the square of the energy gradient rather than minimizing the energy. Also the regular `saddle` and `hessian_seek` commands will minimize square gradient instead of energy.

`0. Exit hessian.`

Exits the menu. '`q`' also works.

`HESSIAN_SEEK` *maxscale*

Seeks to minimize energy along the direction found by Newton's method using the Hessian. *maxscale* is an optional upper bound for the distance to seek. The default maxscale is 1, which corresponds to a plain hessian step. The seek will look both ways along the direction, and will test down to 1e-6 of the maxscale before giving up and returning a scale of 0. This command is meant to be used when the surface is far enough away from equilibrium that the plain hessian command is unreliable, as hessian_seek guarantees an energy decrease, if it moves at all.

`HISTORY`

Print the saved history list of commands.

`LAGRANGE` *expr*

Converts to the Lagrange model of the given order. Note that `lagrange 1` gives the Lagrange model of order 1, which has a different internal representation than the linear model. Likewise, `lagrange 2` does not give the quadratic model.

`LANCZOS` *expr*

`LANCZOS` (*expr*,*expr*)

Does a little Lanczos algorithm and reports the nearest approximate eigenvalues to the given probe value. In the first form, *expr* is the probe value, and 15 eigenvalues are found. In the second form, the first argument is the probe value, the second is the number of eigenvalues desired. The output begins with the number of eigenvalues less than, equal to, and greater than the probe value. Then come the eigenvalues in distance order from the probe. Not real polished yet. Beware that multiplicities reported can be inaccurate. The eigenvalue nearest the probe value is usually very accurate, but others can be misleading due to incomplete convergence. Since the algorithm starts with a random vector, running it twice can give an idea of its accuracy. The first eigenvalue printed is stored in the `last_eigenvalue` variable.

`LINEAR`

Converts to the LINEAR model. Removes midpoints from quadratic model, and interior vertices from the Lagrange model.

`LOAD` *filename*

Terminates the current surface and loads a new datafile. The filename is the datafile name, and can be either a quoted string or a string variable. Since the automatic re-initialization makes Evolver forget all commands (including whatever command is doing the load), this command is not useful except as the last action of a command.

`LONGJ`

Long jiggle once. This provides long wavelength perturbations that can test a surface for stability. The parameters are a wavevector, a phase, and a vector amplitude. The user will be prompted for values. Numbers for vectors should be entered separated by blanks, not commas. An empty reply will accept the defaults. A reply of `r` will generate random values. Any other will exit the command without doing a jiggle. In the random cases, a random amplitude $\vec{A}$ and a random wavelength $\vec{L}$ are chosen from a sphere whose radius is the size of the object. The wavelength is inverted to a wavevector $\vec{w}$. A random phase ψ is picked. Then each vertex $\vec{v}$ is moved by $\vec{A}\sin(\vec{v}\cdot\vec{w}+\psi)$. More control over perturbations may be had with the "`set vertex` *coord* ..." type of command.

`METIS` *expr*

`KMETIS` *expr*

Partitions facets (edges in string model) into n parts using the METIS library of Karypis and Kumar, (`http://www.cs.umn.edu/ karypis/metis/metis.html`) if this library has been compiled into the Evolver (which it isn't in the public distribution, yet). Meant for experiments in partitioning the surface for multi-processors. Partition number of facet left in extra attribute fpart (epart for string model). METIS uses the PMETIS algorithm, KMETIS uses the KMETIS algorithm.

`MOVE` *expr*

Moves the surface along the previous direction of motion by the stepsize given by *expr*. The previous direction can be either from a gradient step or a hessian step (hessian, saddle, hessian_seek, etc.). The stepsize does not affect the current scale factor. A negative step is not a perfect undo, since it cannot undo projections to constraints. `Move` sometimes does not work well with optimizing parameters and hessian together.

`NEW_VERTEX (` *expr*, *expr*, ... `)`

For creating a new vertex. The syntax is that of a function instead of a verb, since it returns the id number of the new vertex. The arguments are the coordinates of the vertex. The new vertex is not connected to anything else; use the `new_edge` command to connect it. Usage:

`newid := NEW_VERTEX(` *expr*, *expr*, ... `)`

Examples:

```
newid := new_vertex(0,0,1)
newid := new_vertex(pi/2,0,max(vertex,x))
```

`NEW_EDGE (` *expr*, *expr* `)`

For creating a new edge. The syntax is that of a function instead of a verb, since it returns the id number of the new edge. The arguments are the id's of the tail and head vertices. Usage:

`newid := NEW_EDGE(` *expr*, *expr* `)`

The new edge has the same default properties as if it had been created in the datafile with no attributes, so you will need to explicitly add any attributes you want. Example to create a set of coordinate axes in 3D:

```
newv1 := new_vertex(0,0,0); fix vertex[newv1];
newv2 := new_vertex(1,0,0); fix vertex[newv2];
newv3 := new_vertex(0,1,0); fix vertex[newv3];
newv4 := new_vertex(0,0,1); fix vertex[newv4];
newe1 := new_edge(newv1,newv2); fix edge[newe1];
newe2 := new_edge(newv1,newv3); fix edge[newe2];
newe3 := new_edge(newv1,newv4); fix edge[newe3];
set edge[newe1] no_refine; set edge[newe1] bare;
set edge[newe2] no_refine; set edge[newe2] bare;
set edge[newe3] no_refine; set edge[newe3] bare;
```

`NEW_FACET (` *expr*, *expr*, ... `)`

For creating a new facet. The syntax is that of a function instead of a verb, since it returns the id number of the new edge. The arguments are the oriented id's of the edges around the boundary of the facet, in the same manner that a face is defined in the datafile. The number of edges is arbitrary, and they need not form a closed loop in the string model. In the soapfilm model, if more than three edges are given, the new face will be triangulated by insertion of a central vertex. In that case, the returned value will be the original attribute of the new facets. In the simplex model, the arguments are the id's of the facet vertices. Usage:

`newid := NEW_FACET(`*expr*, *expr*, ... `)`

The new facet has the same default properties as if it had been created in the datafile with no attributes, so you will need to explicitly add any attributes you want. Example:

```
newf := new_facet(1,2,-3,-4); fix facet where original == newf;
```

`NEW_BODY`

For creating a new body. The syntax is that of a function instead of a verb, since it returns the id number of the new body. There are no arguments. Usage:

```
newid := NEW_BODY
```

The body is created with no facets. Use the `set facet frontbody` and `set facet backbody` commands to install the body's facets. The new body has the same default properties as if it had been created in the datafile with no attributes, so you will need to explicitly add any attributes you want, such as density or target volume. Example:

```
newb := new_body
set facet frontbody newb where color == red
```

`NOTCH` *expr*

Notches ridges and valleys with dihedral angle greater than given value. Same as 'n' command.

`OMETIS` *expr*

Computes an ordering for Hessian factoring using the METIS library of Karypis and Kumar, (`http://www.cs.umn.edu/ karypis/metis/metis.html`) if this library has been compiled into the Evolver (which it isn't in the public distribution, yet). The optional *expr* is the smallest partition size, which defaults to 100. To actually use METIS ordering during factoring, use the toggle metis_factor.

`OPTIMIZE` *expr*

Set 'g' iteration mode to optimizing with upper bound of *expr* on the scale factor. Synonym: optimise.

`PAUSE`

Pauses execution until the user hits RETURN. Useful in scripts to give the user a chance to look at some output before proceeding.

`QUADRATIC`

Changes to the quadratic model by inserting midpoints in edges.

`QUIT, BYE`

Exits Evolver or starts new datafile. Same as '`q`' single-letter command.

`RAWESTV`

Does vertex averaging on all vertices without regard for conserving volume or whether averaged vertices have like constraints. But doesn't move vertices on boundaries. Use rawest_vertex_average for doing one vertex at a time.

`RAWEST_VERTEX_AVERAGE` *generator* Vertex average a set of vertices. Each vertex is moved to the average position of its edge-connected neighbors without regard for volume preservation or any constraints. Vertices are projected back to level-set constraints after motion. Example:

```
rawest_vertex_average vertex[23]
```

`READ` *filename*

Reads commands from the named file. The filename can be either a quoted string or a string variable.

`RAWV`

Does vertex averaging on all vertices without conserving volume on each side of surface. Will only average vertices with those of like type of constraints. Doesn't move vertices on boundaries. Use raw_vertex_average for doing one vertex at a time.

`RAW_VERTEX_AVERAGE` *generator*

Vertex average a set of vertices. Each vertex is moved to the average position of its edge-connected neighbors without regard for volume preservation. Only averages with vertices on the same level-set constraints. Vertices are projected back to level-set constraints after motion. Example:

```
raw_vertex_average vertex[23]
```

`REBODY`

Recalculates connected bodies. Useful after a body has been disconnected by a neck pinching off. Facets of an old body are divided into edge-connected sets, and each set defines a new body (one of which gets the old body number). The new bodies inherit the attributes of the old body. If the original body volume was fixed, then the new bdoies' target volumes become the new actual volumes. If the original body had a volconst, then the new bodies will inherit the same value; this will probably lead to incorrect volumes, so you will need to adjust it by hand. In commands, you may specify the new bodies descended from an original body by using the 'original' atttribute.

`RECALC`

Recalculates everything. Useful after changing some variable or something and recalculation is not automatically done.

`RENUMBER_ALL`

Reassigns element id numbers in accordance with order in storage, i.e. as printed with the LIST commands. Besides renumbering after massive topology changes, this can be used with the `reorder_storage` command to number elements as you desire. Do NOT use this command inside an element generator loop!

`REORDER_STORAGE`

Reorders the storage of element data structures, sorted by the extra attributes `vertex_order_key`, `edge_order_key`, `facet_order_key`, `body_order_key`, and `facetedge_order_key`. Originally written for testing dependence of execution speed on storage ordering, but could be useful for other purposes, particularly when `renumber_all` is used afterwards. Example:

```
define vertex attribute vertex_order_key real
define edge attribute edge_order_key real
define facet attribute facet_order_key real
define body attribute body_order_key real
define facetedge attribute facetedge_order_key real
reorder :=
set vertex vertex_order_key x+y+z;
set edge ee edge_order_key min(ee.vertex,vertex_order_key);
set facetedge fe facetedge_order_key fe.edge[1].edge_order_key;
set facet ff facet_order_key min(ff.vertex,vertex_order_key);
set body bb body_order_key min(bb.facet,facet_order_key);
reorder_storage;
```

`RITZ(`*expr*`,`*expr*`)`

Applies powers of inverse shifted Hessian to a random subspace to calculate eigenvalues near the shift value. First argument is the shift. Second argument is the dimension of the subspace. Prints out eigenvalues as they converge to machine accuracy. This may happen slowly, so you can interrupt it by hitting whatever your interrupt key is, such as CTRL-C, and the current values of the remaining eigenvalues will be printed out. Good for examining multiplicities of eigenvalues.

`SADDLE`

Seek to minimum energy along the eigenvector of the lowest eigenvalue of the Hessian. The eigenvalue is stored in the `last_eigenvalue` variable.

`SHELL`

Invokes a system subshell for the user.

`SOBOLEV`

Uses a positive definite approximation to the area Hessian to do one Newton iteration, following a scheme due to Renka and Neuberger [RN]. See section in Technical Reference chapter for more. Works only on area with fixed boundary; no volume constraints or anything else. Seems to converge very slowly near minimum, so not a substitute for other iteration methods. But if you have just a simple soap film far, far from the minimum, then this method can make a big first step. `SOBOLEV_SEEK` will do an energy-minimizing search in the direction.

`STABILITY_TEST`

Calculates the largest eigenvalue of the mobility matrix. Generates a random vector and applies mobility matrix 20 times. Prints out ratios of successive vector lengths. Useful in investigating stability when using approximate curvature.

`SYSTEM` *stringexpr*

Invokes a subshell to execute the given command. Command must be a quoted string or a string variable. Will wait for command to finish before resuming.

`UTEST`

Runs a test to see if the triangulation is Delaunay. Meant for higher dimensions and simplex model.

`ZOOM` [*integer expr*]

Zooms in on vertex whose id is the given integer, with radius the given expression. Same as 'Z' command, but not interactive. If id and radius omitted, then uses current values.

§6.11.5. Toggles.

Various features can be toggled on or off by giving the toggle name with `ON` or `OFF`. The default is `ON`. This is different from the single-letter command toggles, which always change the state. The toggle names below have brief descriptions of their actions in the ON state. Toggles will usually print their previous state. The current value of a toggle may be found by `print` *togglename*.

`AMBIENT_PRESSURE`

Toggles ideal gas mode, where there is a fixed external pressure. The external pressure can be set with the `pressure` phrase in the topwith the top of the datafile, or at runtime with the 'p' command, e.g. `p 10`.

`APPROXIMATE_CURVATURE`

Use polyhedral curvature (linear interpolation over facets for metric) for mean curvature vector. Actually establishes the inner product for forms or vectors to be integration over facets of euclidean inner products of linear interpolation of values at vertices. synonyms: `approx_curv, approx_curvature`.

`AREA_NORMALIZATION`

Convert force on vertex to mean curvature vector by dividing force by area of star around vertex.

`ASSUME_ORIENTED`

Tells squared mean curvature routines that they can assume the surface is locally consistently oriented. Significant only for extreme shapes.

`AUTOCHOP`

Do autochopping of long edges each iteration. Use `AUTOCHOP :=` *expr* to set autochop length. Each iteration, any edge that is projected to become longer than the cutoff is bisected. If any bisections are done, the motion calculation is redone.

`AUTODISPLAY` Toggle automatic display every time the surface changes. Same effect as 'D' command. Default is ON.

`AUTOPOP`

Toggles automatic deletion of short edges and popping of improper vertices each iteration. Before each iteration, any edge projected to shorten to under the critical length is deleted by identifying its endpoints. The critical length is calculated as $L_c = \sqrt{2\Delta t}$, where Δt is the time step or scale factor. Hence this should be used only with a fixed scale, not optimizing scale factor. The critical length is chosen so that instabilities do not arise in motion by mean curvature. If any edges are deleted, then vertices are examined for improper vertices as in the '`o`' command. Useful in string model. Not fully implemented in soapfilm model yet, since it doesn't check facet areas.

`AUTORECALC`

Toggles automatic recalculation of the surface whenever adjustable parameters or energy quantity moduli are changed. Default is ON.

`BACKCULL`

Prevents display of facets with normal away from viewer. May have different effects in different graphics displays. For example, to see the inside back of a body only, "set frontcolor clear" alone works in 2D displays, but needs backcull also for direct 3D.

`BOUNDARY_CURVATURE`

When doing integrals of mean curvature or squared curvature, the curvature of a boundary vertex cannot be defined by its neighbor vertices, so the area of the boundary vertex star instead is counted with an adjacent interior vertex.

`BUNCH_KAUFMAN`

Toggles Bunch-Kaufman factoring in the alternative minimal degree factoring method (ysmp off). This factors the Hessian as LBL^T where L is lower triangular with ones on the diagonal, and B is block diagonal, with 1x1 or 2x2 blocks. Supposed to be more stable when factoring indefinite hessians.

`CHECK_INCREASE`

Toggles checking for increase of energy in an iteration loop. If energy increases, then the iteration loop is halted. Meant for early detection of instabilities and other problems causing the surface to misbehave. Useful in doing a multiple iteration with a fixed scale. Caution: there are circumstances where an energy increase is appropriate, for example when there are volume or quantity constraints and conforming to the constraints means an energy increase initially.

`CLIPPED, CLIPPED_CELLS`

Sets torus quotient space display to clip to fundamental region. Not an on-off toggle. 3-way toggle with `RAW_CELLS` and `CONNECTED`.

`COLORMAP`

Use colormap from file. Use `COLORMAP := "filename"` to set file.

`CONF_EDGE`

Calculation of squared curvature by fitting sphere to edge and adjacent vertices (conformal curvature).

`CONJ_GRAD`

Use conjugate gradient method. See also `RIBIERE`.

`CONNECTED`

Sets quotient space display to do each body as a connected, wrapped surface. Not an on-off toggle. 3-way toggle with `CLIPPED` and `RAW_CELLS`.

`CONVERT_TO_QUANTITIES`

This will do an automatic conversion of old-style energies to new-style named quantities. This has the same effect as the `-q` command line option, but can be done from the Evolver command prompt. Useful when "`hessian`" complains about not being able to do a type of energy. A few energies don't convert yet. It is my intention that this will be the default sometime in the near future, if it can be made sufficiently fast and reliable.

`CROSSINGFLAG`

Makes the POSTSCRIPT command put breaks in background edges in the string model.

`DEBUG`

Print YACC debug trace of parsing of commands.

`DETURCK`

Motion by unit velocity along normal, instead of by curvature vector.

`DIFFUSION`

Activates diffusion step each iteration.

`DIV_NORMAL_CURVATURE`

Toggle to make `sq_mean_curvature` energy calculate the mean curvature by the divergence of the normal vectors at the vertices of a facet.

`EFFECTIVE_AREA`

In area normalization, the resistance factor to motion is taken to be only the projection of the vertex star area perpendicular to the motion. If squared mean curvature is being calculated, this projected area is used in calculating the curvature.

`ESTIMATE`

Activates estimation of energy decrease in each iteration step. Useful only for a fixed scale factor much less than optimizing, so linear approximation is good.

`FACET_COLORS`

Enables coloring of facets in certain graphics interfaces (e.g. xgraph). If off, facet color is white. Default on.

`FORCE_POS_DEF`

If this is on during YSMP factoring of Hessian and the Hessian turns up indefinite, something will be added to the diagonal element to make it positive definite. Left over from some experiment probably.

`GRAVITY`

Includes gravitational energy in total energy.

`GRIDFLAG`

Makes the POSTSCRIPT command draw all edges of displayed facets, not just those satisfying the current edge-show condition.

`GV_BINARY`

Toggles sending data to geomview in binary format, which is faster than ascii. Default is binary on SGI, ascii on other systems, since there have been problems with binary format on some systems. Ascii is also useful for debugging.

`H_INVERSE_METRIC`

Replaces force by Laplacian of force. For doing motion by Laplacian of mean curvature.

`HESSIAN_DOUBLE_NORMAL`

When hessian_normal is also on and the space dimension is even, then the normal vector components in the last half of the dimensions are copies of those in the first half. Sounds weird, huh? But it is useful when using a string model in 4D to calculate the stability of cylindrically symmetric surfaces.

`HESSIAN_DIFF`

Calculates Hessian by finite differences. Very slow. For debugging Hessian routines.

`HESSIAN_NORMAL`

Constrains hessian iteration to move each vertex perpendicular to the surface. This eliminates all the fiddly sideways movement of vertices that makes convergence difficult. Perpendicular is defined as the volume gradient, except at triple junctions and such, which are left with full degrees of freedom.

`HESSIAN_NORMAL_ONE`

If this and hessian_normal are on, then the normal at any point will be one-dimensional. This is meant for soap films with Plateau borders, where there are triple junctions with tangent films. Ordinary hessian_normal permits lateral movement of such triple junctions, but hessian_normal_one does not. Valid only for the string model in 2D and the soapfilm model in 3D. The normal vector is computed as the eigenvector of the largest eigenvalue of the sum of the normal projection matrices of all the edges or facets adjoining the vertex.

`HESSIAN_NORMAL_PERP`

If this is on, then the Hessian linear metric uses only the component of the normal perpendicular to the facet or edge. This raises eigenvalues slightly.

`HESSIAN_QUIET`

Suppresses status messages during Hessian operations. Default on. For debugging.

`HESSIAN_SPECIAL_NORMAL`

Toggles special private version of hessian_ normal.

`HOMOTHETY`

Adjust total volume of all bodies to fixed value after each iteration by uniformly scaling entire surface.

`INTERP_BDRY_PARAM`

For edges on parameterized boundaries, calculate the parameter values of new vertices (introduced by refining) by interpolating parameter values, rather than extrapolating from one end. Useful only when parameters are not periodic.

`INTERP_NORMALS`

Display using interpolated vertex normals for shading for those graphics interfaces that support it.

`ITDEBUG`

Prints some debugging information during a 'g' step. For gurus only.

`JIGGLE`

Toggles jiggling on every iteration.

`KUSNER`

Calculation of squared curvature by edge formula rather than vertex formula.

`LABELFLAG`

Makes the POSTSCRIPT command print labels on elements.

`LINEAR_METRIC`

Eigenvalues and eigenvectors of the Hessian are defined with respect to a metric. This command toggles a metric that imitates the smooth surface natural metric of L_2 integration on the surface. Use with `HESSIAN_NORMAL` to get eigenvalues and eigenvectors similar to those on smooth surfaces.

`MEMDEBUG`

Prints memory allocation/deallocation messages, and has 'c' command print memory usage. For debugging.

`METIS_FACTOR`

Toggles experimental Hessian matrix factoring using the METIS library of Karypis and Kumar. Not in the public distribution.

`METRIC_CONVERT`

If a Riemannian metric is defined, whether to use the metric to do gradient form to vector conversions. Synonym: metric_conversion.

`NORMAL_CURVATURE`

Calculation of squared curvature by taking area of vertex to be the component of the volume gradient parallel to the mean curvature vector.

`NORMAL_MOTION`

Projects motion to surface normal, defined as the volume gradient. May be useful with squared curvature if vertices tend to slither sideways into ugly patterns.

`OLD_AREA`

In the string model with area normalization, at a triple vertex Evolver normally tries to calculate the motion so that Von Neumann's law will be obeyed, that is, the rate of area change is proportional to the number of sides of a cell. If old_area is ON, then motion is calculated simply by dividing force by star area.

`PINNING`

Check for vertices that can't move because adjacent vertices are not on same constraint when they could be. Obscure.

`PSCOLORFLAG`

Makes the POSTSCRIPT command do color.

`QUANTITIES_ONLY`

Inactivates all energies except named quantities. Meant for programmer's debugging use.

`QUIET`

Suppresses all normal output messages. Good while running scripts. Interrupting a command will turn QUIET off, for sanity.

`QUIETGO`

Suppresses only iteration step output.

`POST_PROJECT`

Introduces extra projections to volume and fixed quantity constraints each iteration. If convergence fails after 10 iterations, you will get a warning message, repeated iterations will stop, and the variable `iteration_counter` will be negative.

`RAW_CELLS`

Sets quotient space display for plain, unwrapped facets. Not an on-off toggle. 3-way toggle with `CLIPPED` and `CONNECTED`.

`RIBIERE`

Makes the conjugate gradient method use the Polak-Ribiere version instead of Fletcher-Reeves. (The toggle doesn't turn on conjugate gradient.) Polak-Ribiere seems to recover much better from stalling. Ribiere is the default mode.

`RUNGE_KUTTA`

Use Runge-Kutta method in iteration step (fixed scale factor only).

`SELF_SIMILAR`

If squared mean curvature energy is being used, this scales the velocity toward a self-similar motion.

`SHADING`

Toggles facet shading in certain graphics interfaces (xgraph, psgraph). Darkness of facet depends on angle of normal from vertical, simulating a light source above surface. Default is ON.

`SHOW_ALL_QUANTITIES`

By default, only explicitly user-defined named quantities are displayed by the 'A' or 'v' commands. If show_all_quantities is on, then all internal named quantities (created by the -q option or by convert_to_quantities) are also shown.

`SHOW_INNER`

Display interior facets, those on 2 bodies.

`SHOW_OUTER`

Display outer facets, those on 0 or 1 body.

`SQUARED_GRADIENT`

Causes the `hessian_seek` command to minimize the square of the gradient of the energy rather than minimize the energy itself. Useful for converging to unstable critical points.

`THICKEN`

Display thickened surface, for use when different sides of a facet are colored differently. Use `THICKEN := ` *expr* to set thickness. The default thickness is 0.001 times the maximum linear dimension of the surface.

`VERBOSE`

Prints action messages from pop edge, pop vertex, delete, notch, refine, dissolve, edgeswap.

`VOLGRADS_EVERY`

Toggles recalculating volume constraint gradients every projection step during constraint enforcement. Good for stiff problems.

`YSMP`

Toggles between Yale Sparse Matrix Package routines for factoring hessians, and my own minimal degree factoring.

`ZENER_DRAG`

Toggles Zener drag feature, in which the velocity of the surface is reduced by a magnitude given by the variable zener_coeff, and the velocity is set to zero if it is smaller than zener_coeff.

§6.12. Graphics commands.

Graphics commands control the display of the surface, both the viewing transformation and which elements are seen. Keep in mind that there are two ways to display graphics on many systems: a native or internal mode entirely under the control of the Evolver (such as the X-windows display), and an external mode where information is provided to an external program such as geomview. Remember that hardcopy output (such as Postscript) will show what the native mode shows, not what geomview shows. Some general commands that affect both modes can be made from the main prompt **Enter command:**. Others, that control the native graphics, are made in a special graphics command mode entered by the '**s**' or '**show**' commands.

Note: native mode graphics are assumed to be slow, and don't automatically redraw by default. In particular, there usually is no polling for window redraw events from the operating system, so after moving or resizing a graphics window you may have to give an explicit 's' command to redraw.

The display consists entirely of facets and edges. Special edges (fixed edges, boundary edges, constraint edges, triple edges) are always shown, unless you make their color CLEAR. The individual facet edges can be toggled with the graphics command '`e`' or the geomview command 'ae'.

Picking: If you use geomview or a Windows OpenGL version of the Evolver, you can right mouse click on a vertex, edge, or facet in the geomview window, and Evolver will print the number of the items picked. These numbers are saved in the internal variables `pickvnum, pickenum, pickfnum`.

This section lists the graphics mode commands first and then the general commands.

The graphics mode prompt is "`Graphics command: `". A graphics command is a string of letters followed by RETURN. Each letter causes an action. Some commands may be preceded by an integer count of how many repetitions to do. Example command: `rr15u2z`. Rotation commands may be preceded by a real number giving the degrees of rotation; an integer will give a repetition count with the default angle of 6 degrees. If you mean to specify the angle, be sure to include a decimal point.

Commands in graphics command mode:

Repeatable commands:

`u` Tip up. Rotates image six degrees about horizontal axis.

`d` Tip down. Rotates image six degrees other way.

`r` Rotate right. Rotates by six degrees about vertical axis.

`l` Rotate left. Rotates by six degrees the other way.

`c` Rotate clockwise by six degrees about center of screen.

`C` Rotate counterclockwise by six degrees about center of screen.

`z` Zoom. Expands image by factor of 1.2.

`s` Shrink. Contracts image by factor of 1.2.

`arrow keys` Move image half screen width in appropriate direction. May not work on all terminals. For MS-Windows versions, arrow keys work when the mouse is in the display window.

Non-repeatable commands:

`R` Reset viewing angles to original defaults and rescale the image to fit the viewing window.

`e` Toggle showing all the facet edges.

`h` Toggle hiding hidden surfaces. When ON, takes longer to display images, but looks better.

`b` Toggles display of bounding box. Useful for visualizing orientation.

`t` Reset mode of displaying torus surfaces. Choice of raw cell, connected bodies, or clipped unit cell. See the **Torus** section for more explanation.

`w` Toggles display of facets entirely on constraints. Meant to show traces where surfaces meet constraints. "w" stands for "wall". Applies to facets run up against inequality constraints.

`B` Toggles display of facets on boundaries or equality constraints attribute.

`v` Toggles showing of convex and concave edges in different colors. "v" stands for "valleys".

`+` Increments color number used for facet edges.

`-` Decrements color number used for facet edges.

`?` Prints help screen for graphics commands.

`q,x` Exit from graphics mode, and return to main command mode.

OpenGL graphics

There is a version of the Evolver for Windows 95/NT that uses OpenGL for its screen graphics. OpenGL is standard on Windows NT, but Windows 95 users may have to download the appropriate OpenGL libraries. Tbe graphics display is invoked with the 's' command, which leaves you at the graphics prompt, which you should quit 'q' right away since graphics commands are better given in the graphics window. Besides all the standard screen graphics commands, there are many geomview-like features. The left mouse button moves the surface continuously, and the right mouse button picks vertices, edges, and facets. With the graphics window in the foreground, these keyboard commands are active:

h	Print a help screen on the console window
r	Rotate mode for left mouse button
t	Translate mode for left mouse button
z	Zoom mode for left mouse button
c	Clockwise/counterclockwise spin mode for left mouse button
W	Widen edges
w	Narrow edges
R	Reset the view
p	Toggle orthogonal/perspective projection
s	Toggle cross-eyed stereo
e	Toggle showing all edges, regardless of "show edge" condition
f	Toggle showing all facets, regardless of "show facet" condition
F	Move the rotate/zoom origin to the last picked vertex
arrow keys	Translate a bit

Graphics commands in main command mode.

The following commands are entered at the main command prompt:

`GEOMVIEW [ON|OFF]`

Toggles geomview display. Same as `P` menu options `8` and `9`.

`GEOMVIEW` *stringexpr*

Sends the string to geomview. Useful for Evolver scripts using geomview to make movies.

`GEOMPIPE [ON|OFF]`

Toggles sending geomview output through named pipe. Same as `P` menu options `A` and `B`.

`GEOMPIPE` *stringexpr*

Redirects Evolver's geomview output through the command *stringexpr*, which is a quoted string or a string variable. Good for debugging, but be sure to have `gv_binary off` so the output is human readable.

`POSTSCRIPT` *stringexpr*

Creates a PostScript file of name *stringexpr* for the current surface. This is the same as the `P 3` command, except there are no interactive responses needed, so it is useful in scripts. The PostScript output options are controlled by the pscolorflag, gridflag, crossingflag, and labelflag toggles. The median gray level can be set with the variable brightness.

`SHOW` *elements* [*name*] [`WHERE` *expr*]

Shows surface with screen graphics and goes into graphics command mode. The condition will prescibe which facets or edges will be displayed with graphics commands. The prescription will remain in effect until the next `show` query. It will also be saved in any dump file. Example:

```
show facets where area < .01
```

The default for facets is to show all facets. This can be done with '`show facets`'. But since many graphics displays (such as `geomview`) like to handle showing edges themselves, the default for edges is just to show triple junctions, borders, and other special edges. To force showing of all edges, do '`show edges where 1`'. Such edges will be displayed by `geomview` despite the status of its `ae` command.

`SHOWQ`

Will show surface and return immediately to main command mode, without going into graphics command mode.

`SHOW_EXPR` *elements* [*name*] [`WHERE` *expr*]

This sets the display prescription without actually doing any graphics. Useful when graphics are not available, as when working remotely, and you want to generate graphical file output with a prescription, or in start-up commands at the end of a datafile.

`SHOW_TRANS` *string*

Applies string of graphics commands to the image transformation matrix without doing any graphic display. The string must be in double quotes or be a string variable, and is the same format as is accepted by the regular graphics command prompt. Example:

```
show_trans "rrdd5z"
```

`TRANSFORMS [ON|OFF]`

Toggles displaying of multiple viewing transforms. These are either those listed explicitly in the datafile or those generated with a previous `transform_expr` command.

`TRANSFORM_DEPTH` *n*

Quick way of generating all possible view transforms from view transform generators, to a given depth *n*. I.e. generates all possible products of *n* or less generators.

`TRANSFORM_EXPR` *string*

If view transform generators were included in the datafile, then a set of view transforms may be generated by a expression with a syntax much like a regular expression. An expression generates a set of transforms, and are compounded by the following rules. Here a lower-case letter stands for one of the generators, and an upper-case letter for an expression.

a	Generates set I, a.
AB	Generates all ordered products of pairs.
nA	Generates all n-fold ordered products.
$A\|B$	Generates union of sets A and B.
(A)	Grouping; generates same set as A.

The precedence order is that nA is higher than AB which is higher that $A|B$. Note that the expression string must be enclosed in double quotes or be a string variable. Examples:

`transform_expr "3(a|b|c)"` (All products of 3 or fewer generators.)

`transform_expr "abcd"` (Generates sixteen transforms)

All duplicate transforms are removed, so the growth of the sets do not get out of hand. Note that the identity transform is always included. The letter denoting a single generator may be upper or lower case. The order is the same order as in the datafile. If 26 generators are not enough for somebody, let me know.

The current `transform_expr` may be used as a string, and the number of transformations generated can be accessed as `transform_count`. For example,

```
print transform_expr
```

`VIEW_4D [ON|OFF]`

For `geomview` graphics, sends a full 4D file in 4OFF and 4VECT format. Default is OFF, so `geomview` gets just the 3D projection.

§6.13. Script examples.

Here are some longer examples using the command language. Since the language expects one command per line, long commands have line breaks only within braces so Evolver can tell the command continues. I tend to break long commands up into several little named commands. The best way to use these longer commands is to put their definitions at the end of your datafile in a "read" section, or have them in a separate file you can read with the "read" command.

1. A script to evolve and regularly save dumps in different files:

```
nn := 1
while nn < 100 do { g 100; ff := sprintf "stage%g.dmp",nn; dump ff; nn:=nn+1}
```

2. Your own iterate command to print out what you want, like energy change each step:

```
olde := total_energy;
gg := { g; printf "Change:  %10.5g\n",total_energy-olde; olde:=total_energy}
```

3. A command to print a file listing vertices and edges (happens to be the OFF format for geomview):

```
aa:=foreach vertex vv do { printf "%f %f %f \n",x,y,z }
bb:=foreach facet ff do { printf "3 ";
foreach ff.vertex do printf "%g ",id-1; printf "\n"}
do_off:= { printf "OFF\n%g %g %g\n",vertex_count,facet_count,edge_count;aa;bb}
```

This prints to stdout, so you would use it with redirection:

```
Enter command:  do_off | "cat >test.off"
```

4. A command to duplicate the fundamental region of a symmetric surface across a plane of mirror symmetry.

```
// Producing datafile surface duplicated across a mirror.
// This is set up to do constraint 1 only.
// Mirrored constraints given x numbers.
// Resulting datafile needs mirrored constraints added by hand.
// Usage: mm | "cat >filename.fe"

// Mirror plane aa*x + bb*y + cc*z = dd
// You have to set these to the proper coefficients for constraint 1
// before doing mm.
aa := 1; bb := 0; cc := 0; dd := 0

ma := foreach vertex do { printf "%g  %g %g %g ",id,x,y,z;
           if fixed then printf "fixed ";
   if on_constraint 1 then printf "constraint 1 " ;
   printf "\n";
   }
mb := { voff := vertex_count }
mc := foreach vertex do if not on_constraint 1 then {
     lam := 2*(dd-aa*x-bb*y-cc*z)/(aa*aa+bb*bb+cc*cc);
           printf "%g  %g %g %g ",id+voff,x+lam*aa,y+lam*bb,z+lam*cc;
           if fixed then printf "fixed ";
   if on_constraint 1 then printf "constraint 1x " ;
   printf "\n";
   }
md := foreach edge ee do {
   printf "%g   ",id;
   foreach ee.vertex do printf "%g ",oid;
           if ee.fixed then printf "fixed ";
   if ee.on_constraint 1 then printf "constraint 1 " ;
   printf "\n";
   }
me := { eoff := edge_count }
mf := foreach edge ee do if not on_constraint 1 then {
   printf "%g   ",id+eoff;
   foreach ee.vertex do {
   if on_constraint 1 then { printf "%g ",oid;}
   else { printf "%g ", id+voff;}  } ;
           if ee.fixed then printf "fixed ";
   if ee.on_constraint 1 then printf "constraint 1x " ;
   printf "\n";
   }
mg := foreach facet ff do {
   printf "%g   ",id;
   foreach ff.edge do printf "%g ",oid;
           if ff.fixed then printf "fixed ";
   if ff.on_constraint 1 then printf "constraint 1 " ;
   printf "\n";
```

```
   }
mh := { foff := facet_count}
mi := foreach facet ff do if not on_constraint 1 then  {
   printf "%g   ",id+foff;
   foreach ff.edge do
     { if on_constraint 1 then { printf "%g ",id;}
       else printf "%g ",(oid<0?-(id+eoff):id+eoff); } ;
           if ff.fixed then printf "fixed ";
   if ff.on_constraint 1 then printf "constraint 1x " ;
   printf "\n";
   }
mm := { list topinfo;
printf "\nVertices\n"; ma;mb;mc;
printf "\nEdges\n"; md;me;mf;
printf "\nFaces\n"; mg;mh;mi;
}
```

§6.14. Interrupts.

Evolver operation may be interrupted with the standard keyboard interrupt, CTRL-C usually (SIGINT for you unix gurus). During repeated iteration steps, this will set a flag which causes the repetition to cease after the current step. Otherwise, the current command is aborted and control returns to the main prompt. If Evolver receives SIGTERM (say from the unix kill command), it will dump to the default dump file and exit. This is useful for stopping background processes running on a script and saving the current state. The same thing will happen with SIGHUP, so losing a modem connection will save the current surface.

Note: In Windows NT/95/98, the second interrupt doesn't do anything much since Windows creates a separate thread to handle the interrupt, and I can't find any way to force the offending thread to stop and longjmp back to where it should. So if the Evolver is really, really stuck, you may just have to kill the whole program.

§6.15. Graphics output file formats.

This section explains the file formats produced by the `P` command.

§6.15.1. Pixar .

A file extension "`.quad`" is automatically added to the filename.

This format lists the facets as quadrilaterals, with the third vertex being repeated. Two options are available:

1. Interpolated normals. This requires recording the normal vector at each vertex.
2. Colors. This requires recording color values (red,green,blue,alpha) at each vertex.

The file starts with a keyword:

`CNPOLY`	colors and normals
`NPOLY`	normals
`CPOLY`	colors
`POLY`	vertices only.

There follows one facet per line. Each vertex is printed as three coordinates, three normal vector components (if used), and four color values (if used).

The coordinates are the true coordinates, not transformed for viewing. Facets are not ordered back to front, since Pixar handles viewing and hidden surfaces itself.

§6.15.2. Geomview.

The `P` command menu option 8 starts the geomview program and feeds it data through a pipe. The file fed into the pipe follows the geomview command file format. Parts of it may be in binary format for speed. A copy of the file may be created by using the P menu option A to create a named pipe, and then using cat on the pipe to read the pipe into your own file. End the file with P option B.

Some handy geomview keyboard commands (to be entered with mouse in the geomview image window):

r Set mouse to rotate mode.

t Set mouse to translate mode.

z Set mouse to zoom mode.

ab Toggle drawing bounding box.

ae Toggle facet edge drawing.

af Toggle drawing facets.

Remember that what you see in geomview is not what you will get with a Postscript output, since Evolver knows nothing about the commands you give geomview.

§6.15.3. PostScript.

A PostScript file can be created with the POSTSCRIPT command or the P command option 3. The output file is suitable for direct input to a PostScript device. The view is from the current viewing angle as established with the `s` command or the `show_trans` *string* command. Edges and facets are drawn back to front.

The P 3 command is interactive. You are given a choice whether to draw all the grid lines (all the facet edges). Even if you don't choose to, all special edges (fixed, boundary, triple edges, bare edges) will still be drawn. You can prevent an edge from being drawn by giving it the color CLEAR. Color data is optionally included; you will be prompted for your choice. For a string model in at least 3 dimensions, you will also be asked whether you want to show edge crossings with breaks in the background edge, which helps in depth perception. There is also a label option, which prints element numbers, for easy cross-referencing with datafiles. Edge orientation is indicated by the labels being slightly displaced toward the edge head, and face labels are signed according to which side you are seeing. By responding with 'i' or 'o' at the labels prompt, you can get either current id numbers or original id numbers.

The POSTSCRIPT command is not interactive. You give the filename with the command, and the output options are controlled by the pscolorflag, gridflag, crossingflag, and labelflag options.

§6.15.4. Triangle file.

This is a format I put in to get the data I wanted for a certain external application. This lists one facet per line. Coordinates are viewing transformed to the screen plane. Facets are listed back to front. The format for a line is

x1 y1 x2 y2 x3 y3 w1 w2 w3 d

Here *(x1,y1)*, *(x2,y2)*, and *(x3,y3)* are the vertex coordinates, and *w1*, *w2*, and *w3* are types of edges:

type 0: ordinary edge, no constraints or boundaries, two adjacent facets.

type 1: edge on constraint or boundary, but not fixed.

type 3: fixed edge.

type 4: edge adjacent to at least 3 facets.

These values can be used to weight different types of edges to illustrate the structure of the surface. *d* is the cosine of the angle of the facet normal to the screen normal. Useful for shading facets.

§6.15.5. SoftImage file.

This consists of two files, one with extension `.mdl` and one with `.def`. The extensions are added automatically.

Chapter 7. Technical Reference.

This chapter explains the mathematics and algorithms used by the Evolver. You don't need to know this stuff to use the Evolver, but it's here for anybody who want to know what exactly is going on and for a programmer's reference.

This chapter currently describes mostly the linear soapfilm model, although some of the named quantity methods are done for quadratic model.

§7.1. Notation.

The term "original orientation" refers to the orientation of an element as recorded in its structure. Original orientations begin with the orientations in the datafile, and are inherited thereafter during refinement and other operations.

Edges.

In referring to a edge e:

$\vec{t}$ and $\vec{h}$ are the tail and head of the side.

$\vec{s}$ is the edge vector, $\vec{s} = \vec{h} - \vec{t}$.

a_i and w_i are the abscissas and weights for Gaussian quadrature on the interval [0,1].

Facets.

In referring to a facet f:

T is the facet surface tension.

v_0, v_1, and v_2 are the vertices around a facet in counterclockwise order.

$\vec{s}_0$, $\vec{s}_1$, and $\vec{s}_2$ are the edges in counterclockwise order around a facet, and v_0 is the tail of $\vec{s}_0$.

a_{ij} and w_i are the barycentric coordinates (j is the vertex index) and weights for evaluation point i for Gaussian quadrature on a triangle.

§7.2. Surface representation.

The basic geometric elements are vertices, edges, three-sided facets, and bodies. There is a structure type for each (see `skeleton.h` for definitions). In addition, there is a facet-edge structure for each incident facet and edge pair. Edges, facets, and facet-edges are oriented. Vertices and bodies could in principle be oriented, but they are not. Elements can be referred to by identifiers which implicitly contain an orientation for the element. Currently these identifiers are 32-bit bitfields with one bit for an orientation bit, bits for element type, a bit for validity, and the rest for a list pointer. Thus one structure is used for both orientations, and the programmer doesn't have to worry about the orientation of the stored structure; all the orientation conversions are taken care of by the structure access functions and macros.

The topology of the surface is defined by the cross-references among the elements. Each facet-edge is in two doubly-linked lists: one of facet-edges around the edge, and another of facet-edges around the facet. Each edge points to its two endpoints (called *head* and *tail*) and to one facet-edge in the facet-edge loop around it. Each facet points to one facet-edge in the facet-edge loop around it, and to the two bodies on its two sides. Each body points to one facet-edge on its boundary. The upshot of all this is that it is easy to find all the facets around an edge or all the edges around a facet. There are also various other linked lists connecting elements; see the Iterators section of the Operation chapter.

§7.3. Energies and forces.

As described in the **Energies** section of the **Model** chapter, all energies are carried by facets or edges. (Volumes for ideal gas energy are calculated from facets and edges also.) Hence it is straightforward to run through all the edges and facets, adding up their energies and accumulating their energy gradients onto the total forces on their vertices. This section gives the formulas used for all the types of energies and forces.

§7.3.1. Surface tension.

Energy of facet.

$$E = \frac{T}{2}\|\vec{s}_0 \times \vec{s}_1\|. \tag{1}$$

Force on vertex.

$$\vec{F}(v_0) = \frac{T}{2}\frac{\vec{s}_1 \times (\vec{s}_0 \times \vec{s}_1)}{\|\vec{s}_0 \times \vec{s}_1\|}. \tag{2}$$

§7.3.2. Crystalline integrand.

$\vec{W}$ is the Wulff vector corresponding to the normal of the facet. The facet has its original orientation.

Energy of facet.

$$E = \frac{T}{2}\vec{W} \cdot \vec{s}_0 \times \vec{s}_1. \tag{3}$$

Force on vertex.

$$\vec{F}(v_0) = \frac{T}{2}\vec{s}_1 \times \vec{W}. \tag{4}$$

§7.3.3. Gravity.

Let G be the acceleration of gravity. Let ρ_+ be the density of the body from which the facet normal is outward, and let ρ_- be the density of the body with inward normal. Let z_i be the z coordinate of v_i. The following formulas are exact for flat facets.

Energy of facet.

$$\begin{aligned} E &= G(\rho_+ - \rho_-)\int\int_{facet} \frac{1}{2}z^2\vec{k} \cdot d\vec{A} \\ &= G(\rho_+ - \rho_-)\frac{1}{12}(z_0^2 + z_1^2 + z_2^2 + z_0z_1 + z_0z_2 + z_1z_2)\frac{1}{2}(\vec{k} \cdot \vec{s}_0 \times \vec{s}_1). \end{aligned} \tag{5}$$

Force on vertex.

$$\vec{F}(v_0) = -\frac{G(\rho_+ - \rho_-)}{24}((2z_0 + z_1 + z_2)(\vec{k} \cdot \vec{s}_0 \times \vec{s}_1)\vec{k} + (z_0^2 + z_1^2 + z_2^2 + z_0z_1 + z_0z_2 + z_1z_2)(\vec{k} \times \vec{s}_1)). \tag{6}$$

§7.3.4. Level set constraint integrals.

This occurs for each constraint with energy integrand that an edge is deemed to satisfy. The edge orientation is the original one. The points evaluated for Gaussian quadrature are

$$\vec{x}_i = a_i\vec{h} + (1 - a_i)\vec{t}. \tag{7}$$

Energy of edge.

$$E = \int_{edge} \vec{E} \cdot \vec{ds} \quad \approx \quad \sum_i w_i \vec{E}(\vec{x}_i) \cdot \vec{s}. \tag{8}$$

Force on vertex.

Component notation and the Einstein summation convention are used here, as vector notation gets confusing. Comma subscript denotes partial derivative.

$$\begin{aligned} F_j(head) &= -w_i(a_i E_{k,j}(\vec{x}_i)s_k + E_j(\vec{x}_i)), \\ F_j(tail) &= -w_i((1-a_i)E_{k,j}(\vec{x}_i)s_k - E_j(\vec{x}_i)). \end{aligned} \tag{9}$$

§7.3.5. Gap areas.

Every edge on a `CONVEX` constraint contributes here. Let $\vec{q}_t$ be the projection of $\vec{s}$ on the constraint tangent space at the tail, and $\vec{q}_h$ likewise at the head. k is the gap constant.

Energy of edge.

$$E = \frac{k}{12}\left(\sqrt{(s^2 - q_t^2)q_t^2} + \sqrt{(s^2 - q_h^2)q_h^2}\right). \tag{10}$$

Force on vertex.

This energy is not the direct derivative of the previous formula. Rather it comes directly from the geometric decrease in gap area due to vertex motion. The formula is derived for the case of a flat gap surface, which should be close enough to the truth to be useful.

$$\begin{aligned} \vec{F}(tail) &= \frac{k\sqrt{(s^2 - q_t^2)q_t^2}}{2q_t^2}\vec{q}_t, \\ \vec{F}(head) &= -\frac{k\sqrt{(s^2 - q_h^2)q_h^2}}{2q_h^2}\vec{q}_h. \end{aligned} \tag{11}$$

§7.3.6. Ideal gas compressibility.

The ideal gas mode is in effect when the keyword is listed in the first part of the datafile. Then prescribed volumes become in essence moles of gas. P_{amb} is the ambient pressure. A body has prescribed volume V_0, actual volume V, and pressure P. The temperature is assumed constant, so $PV = P_{amb}V_0$.

Energy of body.

$$\begin{aligned} E &= \int P - P_{amb} dV = \int P_{amb}V_0/V - P_{amb}dV \\ &= P_{amb}V_0 \ln(V/V_0) - P_{amb}(V - V_0) \end{aligned} \tag{12}$$

Note the energy starts at 0 for $V = V_0$.

Force on vertex.

Let $\vec{g}_m$ be the gradient of the volume body m as a function of the coordinates of the vertex. Let P_m be the pressure of body m. Then

$$\vec{F}(v) = -\sum_m (P_m - P_{amb})\vec{g}_m. \tag{13}$$

§7.3.7. Prescribed pressure.

Ambient pressure is taken to be zero, and body pressures are constant.

Energy of body.

$$E = \int P dV = PV. \tag{14}$$

Force on vertex.

Let $\vec{g}_m$ be the gradient of the volume body m as a function of the coordinates of the vertex. Let P_m be the pressure of body m. Then

$$\vec{F}(v) = -\sum_m P_m \vec{g}_m. \tag{15}$$

§7.3.8. Squared mean curvature.

The integral of squared mean curvature in the soapfilm model is calculated as follows: Each vertex v has a star of facets around it of area A_v. The force due to surface tension on the vertex is

$$F_v = -\frac{\partial A_v}{\partial v}. \tag{16}$$

Since each facet has 3 vertices, the area associated with v is $A_v/3$. Hence the average mean curvature at v is

$$h_v = \frac{1}{2}\frac{F_v}{A_v/3}, \tag{17}$$

and this vertex's contribution to the total integral is

$$E_v = h_v^2 A_v/3 = \frac{1}{4}\frac{F_v^2}{A_v/3}. \tag{18}$$

The corresponding calculation is done for the string model. E_v can be written as an exact function of the vertex cooordinates, so the gradient of E_v can be fed into the total force calculation.

Philosophical note: The squared mean curvature on a triangulated surface is technically infinite, so some kind of approximation scheme is needed. The alternative to locating curvature at vertices is to locate it on the edges, where it really is, and average it over the neighboring facets. But this has the problem that a least area triangulated surface would have nonzero squared curvature, whereas in the vertex formulation it would have zero squared curvature.

Practical note: The above definition of squared mean curvature seems in practice to be subject to instablities. One is that sharp corners grow sharper rather than smoothing out. Another is that some facets want to get very large at the expense of their neighbors. Hence a couple of alternate definitions have been added.

Effective area curvature: The area around a vertex is taken to be the magnitude of the gradient of the volume. This is less than the true area, so makes a larger curvature. This also eliminates the spike instability, since a spike has more area gradient but the same volume gradient. Letting N_v be the volume gradient at v,

$$h_v = \frac{1}{2}\frac{F_v}{||N_v||/3}, \tag{19}$$

and

$$E_v = h_v^2 A_v/3 = \frac{3}{4}\frac{F_v^2}{||N_v||^2} A_v, \tag{20}$$

The facets of the surface must be consistently oriented for this to work, since the evolver needs an "inside" and "outside" of the surface to calculate the volume gradient. This mode is toggled by the "`effective_area ON | OFF`" command. There are still possible instabilities where some facets grow at the expense of others.

Normal curvature: To alleviate the instability of effective_area curvature, the normal_curvature mode considers the area around the vertex to be the component of the volume gradient parallel to the mean curvature vector, rather than the magnitude of the volume gradient. Thus

$$h_v = \frac{1}{2}\frac{F_v||F_v||}{N_v \cdot F_v/3}, \tag{21}$$

$$E_v = \frac{3}{4}\left[\frac{F_v \cdot F_v}{N_v \cdot F_v}\right]^2 A_v. \tag{22}$$

This is still not perfect, but is a lot better. This mode is toggled by the "`normal_curvature ON | OFF`" command. If you have effective area on, then normal curvature will supersede it, but effective area will be in effect when normal curvature is toggled off.

Curvature at boundary: If the edge of the surface is a free boundary on a constraint, then the above calculation gives the proper curvature under the assumption the surface is continued by reflection across the constraint. This permits symmetric surfaces to be represented by one fundamental region. If the edge of the surface is a fixed edge or on a 1-dimensional boundary, then there is no way to calculate the curvature on a boundary vertex from knowledge of neighboring facets. For example, the rings of facets around the bases of a catenoid and a spherical cap may be identical. Therefore curvature is calculated only at interior vertices, and when the surface integral is done, area along the boundary is assigned to the nearest interior vertex.

WARNING: For some extreme shapes, effective_area and normal_curvature modes have problems detecting consistent local surface orientation. The `assume_oriented` toggle lets Evolver assume that the facets have been defined with consistent local orientation.

§7.3.9. Squared Gaussian curvature.

The integral of squared Gaussian curvature over a soapfilm model surface only applies where the star of facets around each vertex is planar. It is calculated as follows. The total Gaussian curvature at a vertex is the angle deficit around the vertex, $2\pi - \sum \theta_i$, where θ_i are the vertex angles of the facets adjacent to the vertex. The average Gaussian curvature is the total Gaussian curvature divided by one third of the area of the star. The average is then squared and integrated over one third of the star.

At boundary: Treated the same way squared mean curvature is.

§7.4. Named quantities and methods.

This section gives details on the calculations of the methods used in named quantities. For those methods where exact evaluation is impossible (due to integrals of functions or quadratic model), evaluation is done by Gaussian integration with the number of evaluation points controlled by the `integration_order` variable. In the formulas below, edge are parameterized by $0 \le u \le 1$ and facets by $0 \le u_1 \le 1, \quad 0 \le u_2 \le 1 - u_1$. S is the differential of the immersion into space, $S_{ij} = \partial x_i / \partial u_j$. J is the Jacobian, $J = (\det(S^T S))^{1/2}$. $\vec{u}_m$ are Gaussian integration points in the domain, $\vec{x}_m$ are the corresponding points in space, and w_m are the Gaussian weights. For the linear model, S and J are constant, but in the quadratic model they are functions of u: $S(\vec{u}_m)$ and $J(\vec{u}_m)$. Likewise for the facet normal $\vec{N}$, which includes the Jacobian factor.

§7.4.1. Vertex value.

Method name `vertex_scalar_integral`. A scalar function $f(\vec{x})$ is evaluated at each vertex the method applies to.

§7.4.2. Edge length.

Method name `edge_tension` or `edge_length`. This method is entirely separate from the default energy calculation for strings. For each edge, the value is the length in Euclidean coordinates. The edge density attribute, used in the default energy calculation, is not used. No metric is used.

$$E = \sum_m w_m J(u_m) \tag{1}$$

However, the `density_edge_length` method does use the edge density ρ:

$$E = \rho \sum_m w_m J(u_m) \tag{2}$$

§7.4.3. Facet area.

Method name `facet_tension` or `facet_area`. This method is entirely separate from the default energy calculation for surfaces. For each facet, the value is the area in Euclidean coordinates. The area density attribute, used in the default energy calculation, is not used. No metric is used.

$$E = \sum_m w_m J(\vec{u}_m) \tag{3}$$

However, the `density_facet_area` method does use the facet density ρ:

$$E = \rho \sum_m w_m J(\vec{u}_m) \tag{4}$$

Method name `facet_area_u`. In the quadratic model, this gives an upper bound of area, for the paranoids who don't trust the regular `facet_area` method. It uses the fact that

$$Area = \int\int (\det(S^T S))^{1/2} du_1 du_2 \leq \left(\int\int \det(S^T S) du_1 du_2\right)^{1/2} \left(\int\int du_1 du_2\right)^{1/2} \tag{5}$$

The integrand on the right is a fourth degree polynomial, and so may be done exactly with 7-point Gaussian integration. This method sets the integral order to at least 4.

Method name `spherical_area`. This is the area of the triangle projected out to the unit sphere, assuming the vertices are all on the unit sphere. The calculation is done in terms of the edge lengths a, b, c (chord lengths, not arcs) by calculating the angle deficit. The formula for angle C is

$$acos\left[\frac{\sqrt{a^2b^2(1-a^2/4)(1-b^2/4)}}{2(a^2+b^2-c^2-ab/2)}\right].$$

Note this formula avoids taking cross products (so it works in any ambient dimension), and avoids dot products of vectors making a small angle.

§7.4.4. Path integrals.

Method name `edge_scalar_integrand`. A scalar integrand $f(\vec{x})$ is evaluated on a edge by Gaussian quadrature:

$$E = \sum_m w_m f(\vec{x}_m) J(u_m). \tag{6}$$

§7.4.5. Line integrals.

Method name `edge_vector_integrand`. A vector integrand $\vec{f}(\vec{x})$ is evaluated on a edge by Gaussian quadrature:

$$E = \sum_m w_m \vec{f}(\vec{x}_m) \cdot \vec{S}(u_m) \tag{7}$$

§7.4.6. Scalar surface integral.

Method name `facet_scalar_integral`. A scalar integrand $f(\vec{x})$ is evaluated on a facet by Gaussian quadrature:

$$E = \sum_m w_m f(\vec{x}_m) J(\vec{u}_m) \tag{8}$$

§7.4.7. Vector surface integral.

Method name `facet_vector_integral`. A vector integrand $\vec{f}(\vec{x})$ is evaluated on a facet by Gaussian quadrature:

$$E = \sum_m w_m \vec{f}(\vec{x}_i) \cdot \vec{N}(\vec{u}_m) \tag{9}$$

For 3D only.

§7.4.8. 2-form surface integral.

Method name `facet_2form_integral`. A 2-form integrand $\omega(\vec{x}_i)$ is evaluated on a facet by Gaussian quadrature:

$$E = \sum_m w_m < A(\vec{u}_m), \omega(\vec{x}_m) >$$

where A is the 2-vector representing the facet, w_i are the Gaussian weights, and x_i are the evaluation points. For any dimensional ambient space.

§7.4.9. General edge integral.

Method name `edge_general_integral`. A scalar function $f(\vec{x}, \vec{t})$ of position and tangent direction is integrated over the edge by Gaussian quadrature:

$$E = \sum_m w_m f(\vec{x}_m, \vec{S}(u_m)) \tag{10}$$

For proper behavior, f should be homogeneous of degree 1 in $\vec{t}$.

§7.4.10. General facet integral.

Method name `facet_general_integral`. A scalar function $f(\vec{x}, \vec{t})$ of position and normal direction is integrated over the edge by Gaussian quadrature:

$$E = \sum_m w_m f(\vec{x}_m, \vec{N}(u_m)) \tag{11}$$

For proper behavior, f should be homogeneous of degree 1 in $\vec{t}$.

§7.4.11. String area integral.

Method name `edge_area`. This is the contribution of one edge to the area of a cell in the string model.

$$E = \int -y dx$$

It includes corrections for torus domains, along the same lines as explained below for `facet_volume`.

§7.4.12. Volume integral.

Method name `facet_volume`. This is the contribution of one facet to the volume of a body, using the Divergence Theorem:

$$E = \int\int z\vec{k} \cdot \vec{N}$$

This is done exactly, without the need for Gaussian integration. Note that this method has no automatic connection to body volumes. If you want to use this to calculate body volumes, you must define a separate quantity for each body. Further, it applies to the positive orientation of the facet, so if your body has negative facets, you must define a separate method instance with modulus -1 for those facets. A little ugly, but maybe it will be cleaned up in a later version.

In a torus domain, wrap corrections are included. Assume the fundamental region is a unit cube. Otherwise, convert to parallelpiped coordinates and multiply the resulting volume by the volume of the fundamenatal parallelpiped. For a body B,

$$V = \int\int\int_B 1 dx dy dz$$

Confine the body to one fundamental region by introducing cut surfaces where the body wraps around. Let M_k be the cut on the k period face of the fundamental region. Actually, one has two copies M_k^- and M_k^+, differing only by translation by the kth period vector. By the Divergence Theorem,

$$\begin{aligned} V &= \int\int_{\partial B} z dx dy + \sum_k \int\int_{M_k^+} z dx dy - \sum_k \int\int_{M_k^-} z dx dy \\ &= \int\int_{\partial B} z dx dy + \int\int_{M_z^+} 1 dx dy. \end{aligned}$$

The surface M_z^+ is bounded by interior curves ∂M_z^+ and by cut lines C_x^+, C_x^-, C_y^+, and C_y^-. So

$$\begin{aligned} V &= \int\int_{\partial B} z dx dy + \int_{\partial M_z^+} x dy \int_{C_x^+} 1 dy \\ &= \int\int_{\partial B} z dx dy + \int_{\partial M_z^+} x dy + \sum y^+ - \sum y^- \end{aligned}$$

where the positive and negative y's are at the endpoints of C_x^+. Note that all of the quantities in the last line can be computed locally on each facet just from the edge wraps. The only ambiguity is in the values of the y's by units of 1. So the final body volumes may be off by multiples of the torus volume. Careful set-up of the problem can avoid this.

§7.4.13. Gravity.

Method name `gravity_method`

This does the same calculation as the usual gravity, but not including incorporating body densities or the gravitational constant. The modulus should be set as the product of the body density and the gravitational constant. Let z_i be the z coordinate of v_i. The following formulas are exact for flat facets.

$$\begin{aligned} E &= \rho \int\int_{facet} \frac{1}{2} z^2 \vec{k} \cdot \vec{dA} \\ &= \frac{\rho}{12}(z_0^2 + z_1^2 + z_2^2 + z_0 z_1 + z_0 z_2 + z_1 z_2)\frac{1}{2}(\vec{k} \cdot \vec{s}_0 \times \vec{s}_1). \end{aligned} \tag{12}$$

It is also exactly calculable in the quadratic model.

§7.4.14. Hooke energy.

Method name `Hooke_energy`. One would often like to require edges to have fixed length. The total length of some set of edges may be constrained by defining a fixed quantity. This is used to fix the total length of an evolving knot, for example. But to have one constraint for each edge would be impractical, since projecting to n constraints requires inverting an $n \times n$ matrix. Instead there is a Hooke's Law energy available to encourage edges to have equal length. Its form per edge is

$$E = |L - L_0|^p \tag{13}$$

where L is the edge length, L_0 is the equilibrium length, embodied as an adjustable parameter 'hooke_length', and the power p is an adjustable parameter 'hooke_power'. The default power is $p = 2$, and the default equilibrium length is the average edge length in the initial datafile. You will want to adjust this, especially if you have a total length constaint. A high modulus will decrease the hooke component of the total energy, since the restoring force is linear in displacement and the energy is quadratic (when $p = 2$). As an extra added bonus, a 'hooke_power' of 0 will give

$$E = -\log|L - L_0|.$$

To give each edge its own equilibrium length, use the hooke2_energy method. Each edge has an equilibrium length extra attribute 'hooke_size'.

To give each edge an energy according to an elastic model,

$$E = \frac{1}{2}\frac{L - L_0)^2}{L_0}$$

use the hooke3_energy method. Each edge has an equilibrium length extra attribute 'hooke_size'. The exponent can be altered from 2 by setting the parameter hooke3_power.

§7.4.15. Local Hooke energy.

Method name `local_hooke_energy`. Energy of edges as springs with equilibrium length being average of lengths of neighbor edges. Actually, the energy is calculated per vertex,

$$E = \left(\frac{L_1 - L_2}{L_1 + L_2}\right)^2 \tag{14}$$

where L_1 and L_2 are the lengths of the edges adjacent to the vertex.

§7.4.16. Integral of mean curvature.

Method name `mean_curvature_integral`. This computes the integral of the signed scalar mean curvature (average of sectional curvatures) over a surface. The computation is exact, in the sense that for a polyhedral surface the mean curvature is concentrated on edges and singular there, but the total mean curvature for an edge is the edge length times its dihedral angle. The contribution of one edge is

$$E = L \arccos\left(\frac{(\vec{a}\cdot\vec{b})(\vec{a}\cdot\vec{b}) - (\vec{a}\cdot\vec{a})(\vec{b}\cdot\vec{c})}{((\vec{a}\cdot\vec{a})(\vec{b}\cdot\vec{b}) - (\vec{a}\cdot\vec{b})^2)^{1/2}((\vec{a}\cdot\vec{a})(\vec{c}\cdot\vec{c}) - (\vec{a}\cdot\vec{c})^2)^{1/2}}\right), \tag{15}$$

where L is the edge length, $\vec{a}$ is the edge, and $\vec{b}, \vec{c}$ are the two adjacent sides from the tail of $\vec{a}$.

§7.4.17. Integral of squared mean curvature.

Method name `sq_mean_curvature`. This is the named method version of the integral of squared mean curvature discussed earlier in this chapter. Including IGNORE_CONSTRAINTS in the method declaration will force the calculation of energy even at fixed points and ignoring constraints.

Method name `eff_area_sq_mean_curvature`. This is the named method version of the integral of squared mean curvature discussed earlier in this chapter, with the effective area discretization of mean curvature.

Method name `normal_sq_mean_curvature`. This is the named method version of the integral of squared mean curvature discussed earlier in this chapter, with the normal curvature discretization of mean curvature.

§7.4.18. Integral of Gaussian curvature.

Method name `Gauss_curvature_integral`. This computes the total Gaussian curvature of a surface with boundary. The Gaussian curvature of a polyhedral surface may be defined at an interior vertex as the angle deficit of the adjacent angles. But as is well-known, total Gaussian curvature can be computed simply in terms of the boundary vertices, which is what is done here. The total Gaussian curvature is implemented as the total geodesic curvature around the boundary of the surface. The contribution of a boundary vertex is

$$E = \left(\sum_i \theta_i\right) - \pi. \tag{16}$$

The total over all boundary vertices is exactly equal to the total angle deficit of all interior vertices plus $2\pi\chi$, where χ is the Euler characteristic of the surface.

§7.4.19. Average crossing number.

Method name `average_crossings`. Between pairs of edges, energy is inverse cube power of distance between midpoints of edges, times triple product of edge vectors and distance vector:

$$E = 1/d^3 * (e1, e2, d). \tag{17}$$

§7.4.20. Linear elastic energy.

Method name `linear_elastic`.

Calculates a linear elastic strain energy for facets based on the Cauchy-Green strain matrix. Let S be Gram matrix of the unstrained facet (dot productss of sides). Let Q be the inverse of S. Let F be Gram matrix of strained facet. Let $C = (FQ-I)/2$, the Cauchy-Green strain tensor. Let v be Poisson ratio. Then energy density is

$$(1/2/(1+v))(Tr(C^2) + v*(TrC)^2/(1-(dim-1)*v))$$

Each facet has extra attribute poisson_ratio and the extra attribute array form_factors[3] = {s11,s12,s22} . If form_factor is not defined by the user, it will be created by Evolver, and the initial facet shape will be assumed to be unstrained.

§7.4.21. Knot energies.

One way of smoothing a knotted curve is to put "electric charge" on it and let it seek its minimum energy position. To prevent the curve from crossing itself and unknotting, the potential energy should have a high barrier (preferably infinite) to curve crossing. This is often done by using a inverse power law potential with a higher power than the standard inverse first power law of electrostatics. A length constraint on the curve is generally necessary to prevent the charges from repelling each other off to infinity.

The Evolver implements several discrete approximations of these potentials. All use the quantity-method feature described in the next section. That is, each type of energy is a method, and these methods can be attached to user-named quantities. The power in the power law is an adjustable parameter `knot_power`, or perhaps some other variable in specific cases. It may be changed interactively with the '`A`' command. The modulus is the multiple of the method added to the quantity. A modulus of 0 inactivates the method. The value of the quantity can be seen with the 'A' command under the name of the quantity.

One of the general problems with discretization is that edges don't know when they are crossing each other. Edges can cross without either their endpoints or midpoints getting close, especially if said edges get long. To keep edge lengths close to equal, there is also an energy called 'hooke energy' described in the Model chapter. Another energy of interest is total squared curvature, which is elastic bending energy. However, it does not provide any barrier to crossing.

All the knot energies proper correspond to double integrals in the continuous limit, so are double sums. Hence the computation time is quadratic in the number of vertices. Hooke energy and square curvature are linear time.

Generally, the continuous integrals are divergent, and in the literature have various regularization terms subtracted off. However, the Evolver discretizations do not have such regularization. The logic is that the discrete versions are finite, and the discete regularization terms are practically constant, so there is no sense in wasting time computing them.

A sample datafile incorporating some of these is `knotty.fe`.

These discrete knot energies are classified below by whether they are computed over all pairs of vertices, edges, or facets.

I. Vertex pair energies. The total energy is the sum over all pairs of vertices v_1, v_2:

$$E = \frac{1}{2} \sum_{v_1 \neq v_2} E_{v_1 v_2}.$$

Ia. Conducting wire. There is a unit charge on each vertex, and edge lengths are not fixed.

$$E_{v_1 v_2} = \frac{1}{|v_1 - v_2|^p}$$

This is approximating the continuous integral

$$E = \int \int \left(\frac{1}{|x(s) - x(t)|^p} - \frac{1}{d(s,t)^p} \right) \rho(s)\rho(t)\, ds\, dt$$

where s, t are arclength parameters, $d(s,t)$ is the shortest arc distance between points, and ρ is charge density. The second term in this integral is a normalization term to prevent infinite energy. It is not present in the discrete version, since the discrete energy is finite and the normalization term is approximately constant. Note that refining doubles the total charge, so refining should be accompanied by reducing the modulus by a factor of 4.

Note that the discrete energy can apply to any dimension surface in any dimension ambient space since it does not use the regularization. It can be used as a dust energy for sphere packing (high knot_power for a hard core potential) or an energy for knotted 2-spheres in 4-space.

The default power is 2.

Datafile line:

```
quantity knot_energy ENERGY modulus 1 global_method knot_energy
```

Ib. Insulating wire.

$$E_{v_1 v_2} = \frac{L_1 L_2}{|v_1 - v_2|^p}$$

where L_1 is the average length of the two edges adjacent to v_1 and L_2 is the average length of the two edges adjacent to v_2. This is approximating the continuous integral

$$E = \int \int \left(\frac{1}{|x(s) - x(t)|^p} - \frac{1}{d(s,t)^p} \right) ds\, dt$$

where s, t are arclength parameters, $d(s,t)$ is the shortest arc distance between points. This corresponds to a uniform charge density, which is why it is an 'insulating wire'. The second term in the integral is a normalization term, which is not included in the discrete version since it is approximately constant. However, a discrete normalization term can be calculated as a separate quantity.

This energy assumes that each vertex has exactly 2 edges attached to it, so it is not suitable for surfaces.

Datafile line:

```
quantity knot_energy ENERGY modulus 1 global_method edge_knot_energy
```

The 'edge' in the method name refers to the edge-weighting of the vertex charges. The default power is 2.

Datafile line for normalization term:

```
quantity norm_term INFO_ONLY modulus 1 global_method uniform_knot_energy_normalizer
```

Ic. Insulating surface.

$$E_{v_1 v_2} = \frac{A_1 A_2}{|v_1 - v_2|^p}$$

where A_1 is the area of the facets adjacent to v_1 and A_2 is the area of the facets adjacent to v_2. This corresponds to a uniform charge density, which is why it is an 'insulating surface'.

Datafile line:

```
quantity knot_energy ENERGY modulus 1 global_method facet_knot_energy
```

The 'facet' in the method name refers to the edge-weighting of the vertex charges. The default power is 4.

II. Edge pair energies. The total energy is the sum over all pairs of edges e_1, e_2 with endpoints $v_{11}, v_{12}, v_{21}, v_{22}$:

$$E = \frac{1}{2} \sum_{e_1 \neq e_2} E_{e_1 e_2}$$

IIa. Box energy. This energy is due to Gregory Buck. It has the form

$$E_{e_1 e_2} = \frac{L_1 L_2}{(d_1 + d_2 + d_3 + d_4 - 2(L_1 + L_2))^p}.$$

Here L_1, L_2 are the edge lengths and d_1, d_2, d_3, d_4 are the distances between endpoints on different edges. This provides an infinite barrier to crossing in the discrete case in a simple form involving only the edge endpoints. The denominator becomes zero for parallel coincident lines or for perpendicular lines on opposite edges of a tetrahedron. This energy should not be turned on until the curve is refined enough that the denominator is always positive. The default power is 2.

Datafile line:

```
quantity buck_energy ENERGY modulus 1 global_method buck_knot_energy
```

IIb. Normal projection energy. This energy is also due to Gregory Buck. It tries to eliminate the need for a normalization term by projecting the energy to the normal to the curve. Its form is

$$E_{e_1 e_2} = \frac{L_1 L_2 \cos^p \theta}{|x_1 - x_2|^p}$$

where x_1, x_2 are the midpoints of the edges and θ is the angle between the normal plane of edge e_1 and the vector $x_1 - x_2$. The default power is 2.

Datafile line:

```
quantity proj_energy ENERGY modulus 1 global_method proj_knot_energy
```

IIc. Conformal circle energy. This energy is due to Peter Doyle, who says it is equivalent in the continuous case to the insulating wire with power 2. Its form is

$$E_{e_1 e_2} = \frac{L_1 L_2 (1 - \cos\alpha)^2}{|x_1 - x_2|^2},$$

where x_1, x_2 are the midpoints of the edges and α is the angle between edge 1 and the circle through x_1 tangent to edge 2 at x_2. Only power 2 is implemented.

Datafile line:

```
quantity circle_energy ENERGY modulus 1 global_method circle_knot_energY
```

III. Facet pair energies. These energies are sums over all pairs of facets.

IIIa. Conformal sphere energy. This is the 2D surface version of the conformal circle energy. Its most general form is

$$E_{f_1 f_2} = \frac{A_1 A_2 (1 - \cos\alpha)^p}{|x_1 - x_2|^q},$$

where A_1, A_2 are the facet areas, x_1, x_2 are the barycenters of the facets, and α is the angle between f_1 and the sphere through x_1 tangent to $f2$ at x_2. The energy is conformally invariant for $p = 1$ and $q = 4$. For $p = 0$ and $q = 1$, one gets electrostatic energy for a uniform charge density. Note that facet self-energies are not included. For electrostatic energy, this is approximately $2.8A^{3/2}$ per facet.

The powers p and q are Evolver variables; see the datafile lines below. The defaults are $p = 1$ and $q = 4$.

There is a nice expression for the general dimensional version of the conformal version of the energy:

$$4E_{f_1f_2} = \frac{A_1A_2}{r^4} - \frac{1}{r^6}det\begin{bmatrix} r\cdot r & r\cdot s_1 & r\cdot s_2 \\ t_1\cdot r & t_1\cdot s_1 & t_1\cdot s_2 \\ t_2\cdot r & t_2\cdot s_1 & t_2\cdot s_2 \end{bmatrix}$$

where $r = |x_1 - x_2|$ and s_1, s_2, t_1, t_2 are the sides of the two facets.

Datafile lines: `parameter surface_knot_power = 1 // this is q  parameter surface_cos_power = 0 // this is p  quantity sphere_energy ENERGY modulus 1 global_method sphere_knot_energy`

§7.5. Volumes.

A body can get volume from a surface integral over its boundary facets and from content integrals of boundary facets edges on constraints. Also important are the gradients of body volumes at vertices. There are two choices built-in surface integrals: default and `SYMMETRIC_CONTENT`.

§7.5.1. Default facet integral.

A facet oriented with normal outward from a body makes a contribution to its volume of

$$V = \int\int_{facet} z\vec{k}\cdot d\vec{A} = \frac{1}{6}(z_0 + z_1 + z_2)\vec{k}\cdot\vec{s}_0\times\vec{s}_1. \tag{1}$$

The gradient of the volume of the body as a function of the vertex v_0 coordinates is

$$\vec{g} = \frac{1}{6}\left((\vec{k}\cdot\vec{s}_0\times\vec{s}_1)\vec{k} + (z_0 + z_1 + z_2)\vec{k}\times\vec{s}_1\right). \tag{2}$$

§7.5.2. Symmetric content facet integral.

A facet oriented with normal outward from a body makes a contribution to its volume of

$$V = \frac{1}{3}\int\int_{facet}(x\vec{i} + y\vec{j} + z\vec{k})\cdot d\vec{A} = \frac{1}{6}\vec{v}_0\cdot\vec{v}_1\times\vec{v}_2. \tag{3}$$

This can be seen most easily by looking at the facet as the base of a tetrahedron with its fourth vertex at the origin. The integral over the three new faces is zero, so the integral over the original facet must be the volume of the tetrahedron.

The gradient of the volume of the body as a function of the vertex v_0 coordinates is

$$\vec{g} = \frac{1}{6}\vec{v}_1\times\vec{v}_2. \tag{4}$$

§7.5.3. Edge content integrals.

Let body b have facet f with outward normal. Suppose f has edge s deemed to have content integrand $\vec{U}$. Then b gets volume

$$V = \int_e \vec{U}\cdot d\vec{s} \quad\approx\quad \sum_i w_i\vec{U}(\vec{x}_i)\vec{s}. \tag{5}$$

The volume gradients for the body are (summation convention again)

$$\begin{aligned} g_j(head) &= w_i(a_i U_{k,j}(\vec{x}_i)s_k + U_j(\vec{x}_i)), \\ g_j(tail) &= w_i((1-a_i)U_{k,j}(\vec{x}_i)s_k - U_j(\vec{x}_i)). \end{aligned} \tag{6}$$

§7.5.4. Volume in torus domain.

The wrapping of the edges across the faces of the unit cell makes the calculation of volumes a bit tricky. Suppose body b has vertices v_k. Ideally, we would like to adjust the vertices by multiples of the unit cell basis vectors to get a body whose volume we could find with regular Euclidean methods. Unfortunately, all we know are the edge wraps, i.e. the differences in the adjustments to endpoints of edges. But this turns out to be enough, if we are a little careful with the initial volumes in the datafile.

Let $\vec{A}_k$ be the vertex adjustment for vertex k, and $\vec{T}_j$ be the wrap vector (difference in endpoint adjustments) for facet-edge j. Let m index facets. Then

$$\begin{aligned} V &= \frac{1}{6}\sum_{facets\ m}(\vec{v}_{m0}+\vec{A}_{m0})\cdot(\vec{v}_{m1}+\vec{A}_{m1})\times(\vec{v}_{m2}+\vec{A}_{m2}) \\ &= \frac{1}{6}(S_1+S_2+S_3+S_4) \end{aligned} \tag{7}$$

where

$$\begin{aligned} S_1 &= \sum_{facets\ m}\vec{v}_{m0}\cdot\vec{v}_{m1}\times\vec{v}_{m2}, \\ S_2 &= \sum_{facet-edges\ j}\vec{v}_{j0}\cdot\vec{v}_{j1}\times\vec{A}_{j2}, \\ S_3 &= \sum_{facet-verts\ k}\vec{v}_{k}\cdot\vec{A}_{k1}\times\vec{A}_{k2}, \\ S_4 &= \sum_{facets\ m}\vec{A}_{m0}\cdot\vec{A}_{m1}\times\vec{A}_{m2}. \end{aligned} \tag{8}$$

The first of these sums is straightforward. The second sum can be regrouped, pairing the two oppositely oriented facet-edges j and $-j$ for each edge together:

$$\begin{aligned} S_2 &= \sum_{edges\ j}\vec{v}_{j0}\cdot\vec{v}_{j1}\times\vec{A}_{j2}+\vec{v}_{-j0}\cdot\vec{v}_{-j1}\times\vec{A}_{-j2} \\ &= \sum_{edges\ j}\vec{v}_{j0}\cdot\vec{v}_{j1}\times(\vec{A}_{j2}-\vec{A}_{-j2}) \\ &= \frac{1}{2}\sum_{edges\ j}\vec{v}_{j0}\cdot\vec{v}_{j1}\times(\vec{T}_{-j2}+\vec{T}_{j1}-\vec{T}_{j2}-\vec{T}_{-j1}) \end{aligned} \tag{9}$$

which can be regrouped into a sum over facet-edges, which can be done facet by facet:

$$S_2 = \sum_{facet-edges\ j}\vec{v}_{j0}\cdot\vec{v}_{j1}\times(\vec{T}_{j1}-\vec{T}_{j2}).$$

The third sum we group terms with a common vertex together, with the inner facet sum over facets around vertex k:

$$\begin{aligned} S_3 &= \sum_{vertices\ k} \left(\vec{v}_k \cdot \sum_{facets i} \vec{A}_{i1} \times \vec{A}_{i2} \right) \\ &= \sum_{vertices\ k} \left(\vec{v}_k \cdot \sum_{facets i} (\vec{A}_{i1} - \vec{A}_{k0}) \times (\vec{A}_{i2} - \vec{A}_{k}) \right) \\ &= \sum_{vertices\ k} \left(\vec{v}_k \cdot \sum_{facets\ i} \vec{T}_{i0} \times -\vec{T}_{i2} \right) \\ &= \sum_{facet-vertices\ k} \vec{v}_k \cdot \vec{T}_{k2} \times \vec{T}_{k0}, \end{aligned} \tag{10}$$

which again can be done facet by facet.

The fourth sum is a constant, and so only needs to be figured once. Also, it is a multiple of the unit cell volume. Therefore, if we assume the volume in the datafile is accurate to within $\frac{1}{12}V_c$, we can calculate the other sums and figure out what the fourth should be.

The body volume gradient at v_0 will be

$$\vec{g} = \frac{1}{6}\left(\sum_{facets\ m\ on\ v} \vec{v}_{m1} \times \vec{v}_{m2} + \frac{1}{2}\vec{v}_{m1} \times (\vec{T}_{m1} - \vec{T}_{m2}) + \vec{T}_{m0} \times \vec{T}_{m2} \right). \tag{11}$$

§7.6. Constraint projection.

Suppose vertex v has constraints that are the zero sets of functions $f_1, \ldots, f_n$.

§7.6.1. Projection of vertex to constraints.

The solution will be by Newton's method. Several steps may be necessary to get within the tolerance desired of the constraint. Inequality constraints are included if and only if the inequality is violated.

For one step, we seek a displacement $\delta\vec{v}$ that is a linear combination of the constraint gradients and will cancel the difference of the vertex from the constraints:

$$\delta\vec{v} = \sum_i c_i \nabla f_i \qquad \text{such that} \qquad \delta\vec{v} \cdot \nabla f_j = -f_j(v) \qquad \text{for each } j. \tag{1}$$

Thus

$$\sum_i c_i \nabla f_i \cdot \nabla f_j = -f_j(v) \qquad \text{for each j.} \tag{2}$$

This is a linear system that can be solved for the c_i and thus $\delta\vec{v}$.

§7.6.2. Projection of vector onto constraint tangent space.

We want to project a vector $\vec{F}$ (say, a force vector) onto the intersection of all the tangent planes of the constraints at a vertex. We project $\vec{F}$ by subtracting a linear combination of constraint gradients:

$$\vec{F}_{proj} = \vec{F} - \sum_i c_i \nabla f_i \qquad \text{such that} \qquad \vec{F}_{proj} \cdot \nabla f_j = 0 \qquad \text{for each } j. \tag{3}$$

This leads to the system of linear equations for the c_i:

$$\sum_i c_i \nabla f_i \cdot \nabla f_j = \vec{F} \cdot \nabla f_j. \tag{4}$$

§7.7. Volume and quantity constraints.

This section explains how volume and quantity constraints are enforced. Henceforth in this section, everything referring to the volume of a body can also refer to the value of a quantity constraint. There are two parts to this enforcement on each iteration: a volume restoring motion that corrects for any volume deviations, and a projection of the vertex motions to the subspace of volume-preserving motions. Let g_{bv} be the gradient of the volume of body b as a function of the position of vertex v. We will also use B as a body index. If the bodies fill a torus, then one body is omitted to prevent singular systems of equations.

§7.7.1. Volume restoring motion.

Let the current excess volume of body b be δb (which may be negative, of course). We want a volume restoring motion $\vec{R}_v$

$$\vec{R}_v = \sum_B c_B \vec{g}_{Bv} \qquad \text{and} \qquad \sum_v \vec{R}_v \cdot \vec{g}_{bv} = -\delta b \qquad \text{for each } b \tag{1}$$

which leads to the linear system for the c_B:

$$\sum_B c_B \sum_v \vec{g}_{Bv} \cdot \vec{g}_{bv} = -\delta b \qquad \text{for each } b. \tag{2}$$

§7.7.2. Motion projection in gradient mode.

The motion in this mode is the scale factor times the force on a vertex. Let $\vec{F}_v$ be the total force at v. We want a projected force $\vec{F}_{vproj}$ such that

$$\vec{F}_{vproj} = \vec{F}_v - \sum_B a_B \vec{g}_{Bv} \qquad \text{and} \qquad \sum_v \vec{F}_{vproj} \cdot \vec{g}_{bv} = 0 \qquad \text{for each } b \tag{3}$$

which leads to the linear system for a_B

$$\sum_B a_B \sum_v \vec{g}_{Bv} \cdot \vec{g}_{bv} = \sum_v \vec{F}_v \cdot \vec{g}_{bv} \qquad \text{for each } b. \tag{4}$$

The coefficients a_B are the pressures in the bodies when the surface is in equilibrium.

§7.7.3. Force projection in mean curvature mode.

The motion in this mode is the scale factor times the force on a vertex divided by the area of the vertex. Let $\vec{F}_v$ be the total force at v and A_v its area. We want a projected force $\vec{F}_{vproj}$ such that

$$\vec{F}_{vproj} = \vec{F}_v - \sum_B a_B \vec{g}_{Bv} \qquad \text{and} \qquad \sum_v \vec{F}_{vproj} \cdot \vec{g}_{bv}/A_v = 0 \qquad \text{for each } b \tag{5}$$

which leads to the linear system for a_b

$$\sum_b a_b \sum_v \vec{g}_{Bv} \cdot \vec{g}_{bv}/A_v = \sum_v \vec{F}_v \cdot \vec{g}_{bv}/A_v \qquad \text{for each } b. \tag{6}$$

The coefficients a_b are the pressures in the bodies even when the surface is not in equilibrium. At equilibrium, the pressures calculated with or without the mean curvature option are equal.

§7.7.4. Pressure at z = 0.

The pressure reported by the `v` command is actually the Lagrange multiplier for the volume constraint. The pressure may vary in a body due to gravitational force, for example. In that case, the Lagrange multiplier is the pressure where other potential energies are 0. For example, if gravitational force is acting, then the

Lagrange multiplier is the pressure at $z = 0$. Let σ be surface tension, H the scalar mean curvature, $\vec{N}$ the unit normal vector to the surface, ρ the body density, G the gravitational constant, and λ the lagrange multiplier. The gradient of area energy is $2\sigma H\vec{N}$, the gradient of gravitational potential energy is $G\rho z\vec{N}$, and the gradient of volume is $\vec{N}$. Therefore at equilibrium

$$2\sigma H\vec{N} + G\rho z\vec{N} = \lambda\vec{N}.$$

At $z = 0$

$$2\sigma H\vec{N} = \lambda\vec{N}.$$

So $pressure(z = 0) = 2\sigma H = \lambda$.

§7.8. Iteration.

This section explains what happens during each iteration (g command). Δt is the current scale factor.

§7.8.1. Fixed scale motion.

1. Do diffusion if called for. For each facet f, let A_f be the area of the facet. Let m_1 and P_1 be the prescribed volume (mass) of a body on one side of f, and m_2 and P_2 those of the body on the other side. Let κ be the diffusion constant. If there is no body on one side, the pressure there is zero or ambient pressure for the ideal gas model. The mass transfers are:

$$dm_1 = -\kappa(P_1 - P_2)A_f\Delta t \qquad \text{and} \qquad dm_2 = -\kappa(P_2 - P_1)A_f\Delta t. \tag{1}$$

2. All the forces on vertices are calculated, as detailed in the **Energies and Forces** section above. If the mean curvature option is active, each force is divided by the vertex's associated area.
3. All forces at `FIXED` vertices are set to zero. Forces on constrained vertices are projected to the constraint tangent spaces. Forces on vertices on boundaries are converted to forces in boundary parameter spaces.
4. The volume restoring motion is calculated, and the forces are projected according to body volume constraints.
5. Jiggle if jiggling is toggled on.
6. Move vertices by volume restoring motion and current scale times force.
7. Enforce constraints on vertices. Do up to 10 projection steps until vertex is within tolerance of constraints.
8. Recalculate volumes, pressures, areas, and energy.

§7.8.2. Optimizing scale motion.

The idea is to find three scale factors that bracket the energy minimum and then use quadratic interpolation to estimate the optimum scale factor. The first four steps are the same as for fixed scale motion.

5. Save current coordinates.
6. Move vertices by volume restoring motion and current scale times force and enforce constraints. Recalculate energy.
7. Restore coordinates, double scale factor, move surface, and recalculate energy. If this energy is higher, keep cutting scale factor in half until energy increases again. If it is lower, keep doubling scale factor until energy increases.
8. Eventually get three scale factors s_1, s_2, s_3 with energies e_1, e_2, e_3 bracketing a minimum. The scale factors are successive doublings: $s_2 = 2s_1$, $s_3 = 2s_2$. Do quadratic interpolation to find approximate optimum:

$$s_{opt} = 0.75s_2(4e_1 - 5e_2 + e_3)/(2e_1 - 3e_2 + e_3). \tag{2}$$

9. Jiggle if jiggling is on.

10. Move vertices by volume restoring motion and optimum scale times force.
11. Project to constraints.
12. Recalculate volumes, pressures, areas, and energy.

§7.8.3. Conjugate gradient mode.

The `U` command toggles conjugate gradient mode. The conjugate gradient method does not follow the gradient downhill, but makes an adjustment using the past history of the minimization. It usually results in much faster convergence. At iteration step i, let S_i be the surface, E_i its energy, $\vec{F}_i(v)$ the force at vertex v, and $\vec{h}_i(v)$ the "history vector" of v. Then

$$\vec{h}_i(v) = \vec{F}_i(v) + \gamma \vec{h}_{i-1} \tag{3}$$

where

$$\gamma = \frac{\sum_v \vec{F}_i(v) \cdot \vec{F}_i(v)}{\sum_v \vec{F}_{i-1}(v) \cdot \vec{F}_{i-1}(v)}. \tag{4}$$

Then a one-dimensional minimization is performed in the direction of $\vec{h}_i$, as described in steps 6,7,8 above. It is important that all volumes and constraints be enforced during the one-dimensional minimization, or else the method can go crazy. The history vector is reset after every surface modification, such as refinement or equiangulation.

The above formula is the Fletcher-Reeves version of conjugate gradient. The version actually used is the Polak-Ribiere version, which differs only in having

$$\gamma = \frac{\sum_v (\vec{F}_i(v) - \vec{F}_{i-1}(v)) \cdot \vec{F}_i(v)}{\sum_v \vec{F}_{i-1}(v) \cdot \vec{F}_{i-1}(v)}. \tag{5}$$

Polak-Ribiere seems to recover much better from stalling. The `ribiere` command toggles between the two versions.

Conjugate gradient blowups: Sometimes conjugate gradient wants to move so far and fast that it loses contact with volume constraints. Before, Evolver did only one or two Newton method projections back to volume constraints each iteration; now it will do up to 10. Ordinary iteration does only one projection still, but you can get the extra projections with the "post_project" toggle. If convergence fails after 10 iterations, you will get a warning message, repeated iterations will stop, and the variable "iteration_counter" will be negative.

§7.9. Hessian iteration.

The "`hessian`" command constructs a quadratic approximation of the energy and solves for the minimum (or whatever the critical point happens to be). See the Hessian section of the Model chapter for remarks on its use. The independent variables used are the displacements of vertex coordinates. For vertices on level set constraints, the coordinates are replaced by set of variables equal in number to the degrees of freedom. Fixed vertices are not represented. Let X denote the vector of independent variables. Let B denote the energy gradient, and let H be the Hessian matrix of second partial derivatives. Hence the energy change is

$$E = \frac{1}{2} X^T H X + B^T X. \tag{1}$$

Global constraints like volumes and fixed named quantities are actually scalar functions of X with fixed target values. Index these constraints by i, and let C_i be the gradient and Q_i the Hessian of constraint i. Let F_i be the current value and F_{i0} be the target value of constraint i. Then we have the constraint conditions on X

$$\frac{1}{2} X^T Q_i X + C_i^T X = F_{i0} - F_i. \tag{2}$$

Let Γ_i be the Lagrange multiplier for constraint i. The Lagrange multiplier is shown in the "pressure" column of the 'v' command. At a critical point we have

$$X^T H + B^T = \sum_i \Gamma_i (X^T Q_i + C_i^T).$$

So we want to solve the system

$$\begin{aligned} (H - \sum_i \Gamma_i Q_i)X - \sum_i \Gamma_i C_i &= -B \\ \frac{1}{2} X^T Q_i X + C_i^T X &= F_{i0} - F_i. \end{aligned} \tag{3}$$

We do this by Newton's method. This means solve the system

$$\begin{pmatrix} H - \sum_i \Gamma_i Q_i & C \\ C^T & 0 \end{pmatrix} \begin{pmatrix} X \\ d\Gamma \end{pmatrix} = \begin{pmatrix} C\Gamma - B \\ F_0 - F \end{pmatrix} \tag{4}$$

where C is the matrix with columns C_i, Γ is the vector of Γ_i, $d\Gamma$ is the change in Γ, and F_0, F are the vectors of constraint values. For convenience, denote $A = H - \sum_i \Gamma_i Q_i$. The solution of this is

$$\begin{aligned} d\Gamma &= (C^T A^{-1} C)^{-1} (C^T A^{-1} (C\Gamma - B) - (F_0 - F)) \\ X &= A^{-1}(C\Gamma - B) - A^{-1} C d\Gamma. \end{aligned} \tag{5}$$

I choose to solve for A^{-1} first for two reasons: (1) A is sparse, while C is not, and (2) finding A^{-1} and $C^T A^{-1} C$ can tell us about the stability of the critical point. Actually, A^{-1} is never found as such. Instead, a Cholesky factorization $A = U^T D U$ is found and used to solve $AY = C$ for $Y = A^{-1}C$ and so forth. Here U is an upper triangular matrix, and D is a diagonal matrix. The factoring is done with routines from the Yale Sparse Matrix Package, modified a bit since the matrix A may not be positive definite. Negative entries in D are allowed, and the total number of them is the *index* of A. Zero entries are also allowed, since many surfaces have degenerate minimums of energy (due to translational invariance, for example).

Degeneracy is detected during factoring of the Hessian by the appearance of a zero diagonal entry. If the zero does represent a true degeneracy, then the whole row and column of that zero should also be zero at that stage of the factoring. The factoring routine then puts a 1 on the diagonal to insure an invertible D. This has no effect on any of the solutions. The cutoff value for considering a diagonal element to be zero is given by the variable "`hessian_epsilon`" which may be set by the user with an ordinary assignment command. Its default value is 1e-8.

The index of the constrained Hessian is given by the relation

$$index(H_{constrained}) = index(A) - index(C^T A^{-1} C). \tag{6}$$

If this index is positive, then a warning message is printed. Once the Hessian method has converged, this provides a test for being a local minimum. At a saddle point, the constrained index is positive and it is possible to find a downhill direction. The command `saddle` implements this.

Now a word on handling vertices on level-set constraints. Note that above, X was composed of independent variables. For a vertex on k level-set constraints, there are $n - k$ independent variables, where n is the space dimension. Let the constraints have the quadratic approximation (here X is original space variables)

$$K_i^T X + \frac{1}{2} X^T G_i X = 0, \qquad 1 \le i \le k. \tag{7}$$

and let K have columns K_i. Let P have for columns a basis for the nullspace of K^T (which is the tangent space to all the constraints), and let Y be coordinates in the P basis, so $X = PY$. Y is the set of independent coordinates. Now if we have a quantity E with quadratic representation,

$$E = \frac{1}{2} X^T H X + B^T X, \tag{8}$$

then we want to get an expression in Y, keeping in mind that Y really coordinatizes a tangent space orthogonally projected back to a curved constraint. For a given Y, the projection back will be a linear combination of constraint normals,

$$\Delta X = \sum_i \Gamma_i K_i \tag{9}$$

so

$$K_i^T(PY + K\Gamma) + \frac{1}{2}(PY + K\Gamma)^T G_i(PY + K\Gamma) = 0, \qquad 1 \le i \le k. \tag{10}$$

We keep only terms quadratic in Y, and since $K^T P = 0$ and Γ is quadratic in Y,

$$K_i^T K\Gamma + \frac{1}{2} Y^T P^T G_i PY = 0, \qquad 1 \le i \le k. \tag{11}$$

So

$$\Gamma_j = \sum_i -\frac{1}{2}(K^T K)^{-1}_{ji} Y^T P^T G_i PY, \qquad 1 \le j \le k. \tag{12}$$

Hence we can write E in terms of Y as

$$E = \frac{1}{2} Y^T P^T HPY + \sum_j \sum_i -\frac{1}{2} B^T K_j (K^T K)^{-1}_{ji} Y^T P^T G_i PY + B^T PY. \tag{13}$$

For each vertex, P and

$$\sum_j \sum_i -K_j (K^T K)^{-1}_{ji} Y^T P^T G_i PY \tag{14}$$

are calculated just once, and then used with whatever H's and B's turn up in the course of the calculation of E. Like adjustments are made for all the Q_i's above also.

Some notes for people rash enough to write hessian routines for their own named quantity methods: Hessians can only be written for methods that are evaluated independently on elements. Two vertices are allocated a Hessian entry only if they are joined by an edge. See q_edge_tension_hessian() in hessian.c for an example. For debugging, the command "`hessian_menu`" brings up a little menu. You can turn debugging on, in which case you get a whole lot of information dumped to the screen as the algorithm proceeds. Don't do that with more that 3 or 4 unfixed vertices. The sequence of menu options for one iteration is 1,2,3,4 (stepsize 1). For checking Hessian matrix values, there is a "`hessian_diff`" command at the main prompt which will toggle on the calculation of the Hessian matrix by finite differences.

If `hessian_normal` is toggled on, then each vertex is constrained to move perpendicular to the surface. This eliminates all the fiddly sideways movement of vertices that makes convergence difficult. Highly recommended. Perpendicular is defined as the volume gradient, except at triple junctions and such, which are left with full degrees of freedom.

§7.10. Dirichlet and Sobolev approximate Hessians.

This section describes features not necessary to understanding Evolver operation, but they are included because I think they are interesting.

The problem in using quickly converging iteration methods such as Newton's method (as embodied in the `hessian` command) is that the Hessian of the energy is usually indefinite except when extremely close to a minimum. The idea here is to construct a positive definite quadratic form that is tangent to the area functional and whose Hessian is an approximation of the possibly indefinite area Hessian. The approximation can be solved with standard numerical linear algebra, and the solution will decrease area, if the area is not at a critical point. If the area is at a critical point, then so is the approximation.

Consider an m-dimensional simplex τ in R^n. Let F be a $m \times n$ matrix whose rows are the side vectors of τ emanating from one vertex. Then the area of τ is

$$\mathcal{A}_\tau(F) = \frac{1}{m!}(\det FF^T)^{1/2}.$$

For convenience, let $A = FF^T$.

For the first derivative, let G be a displacement of F, and let α be a parameter. Then $A = (F+\alpha G)(F+\alpha G)^T$ and

$$\begin{aligned} D\mathcal{A}_\tau(G) &= \left.\frac{\partial \mathcal{A}_\tau(F+\alpha G)}{\partial \alpha}\right|_{\alpha=0} = \frac{1}{2}\frac{1}{m!}Tr\left(\frac{\partial A}{\partial \alpha}A^{-1}\right)(\det A)^{1/2} = \frac{1}{2}\frac{1}{m!}Tr((GF^T+FG^T)A^{-1})(\det A)^{1/2} \\ &= \frac{1}{m!}Tr(GF^TA^{-1})(\det A)^{1/2}. \end{aligned}$$

For the second derivative, let G_1 and G_2 be displacements. Then

$$D^2\mathcal{A}_\tau(G_1,G_2) = \left.\frac{\partial^2 \mathcal{A}_\tau(F+\alpha G_1+\beta G_2)}{\partial\alpha\partial\beta}\right|_{\alpha=\beta=0} = \frac{\partial^2}{\partial\alpha\partial\beta}\frac{1}{m!}(\det((F+\alpha G_1+\beta G_2)(F+\alpha G_1+\beta G_2)^T))^{1/2}$$

$$\begin{aligned} &= \left.\frac{\partial^2}{\partial\alpha\partial\beta}\frac{1}{m!}(\det A)^{1/2}\right|_{\alpha=\beta=0} = \left.\frac{\partial}{\partial\alpha}\frac{1}{m!}\frac{1}{2}Tr(\frac{\partial A}{\partial\beta}A^{-1})(\det A)^{1/2}\right|_{\alpha=\beta=0} \\ &= \left.\frac{1}{m!}\left[\frac{1}{2}Tr\left(\frac{\partial^2 A}{\partial\alpha\partial\beta}A^{-1} - \frac{\partial A}{\partial\alpha}A^{-1}\frac{\partial A}{\partial\beta}A^{-1}\right) + \frac{1}{2}Tr\left(\frac{\partial A}{\partial\alpha}A^{-1}\right)\frac{1}{2}Tr\left(\frac{\partial A}{\partial\beta}A^{-1}\right)\right](\det A)^{1/2}\right|_{\alpha=\beta=0} \\ &= \frac{1}{m!}\left[\frac{1}{2}Tr(2G_1G_2^TA^{-1} - (G_1F^T+FG_1^T)A^{-1}(G_2F^T+FG_2^T)A^{-1}) + \frac{1}{4}Tr(2G_1F^TA^{-1})Tr(2G_2F^TA^{-1})\right](\det A)^{1/2} \\ &= \frac{1}{m!}\left[Tr(G_1G_2^TA^{-1} - G_1F^TA^{-1}G_2F^TA^{-1} - FG_1^TA^{-1}G_2F^TA^{-1}) + Tr(G_1F^TA^{-1})Tr(G_2F^TA^{-1})\right](\det A)^{1/2}. \end{aligned}$$

Thus the true area Hessian can be written as the quadratic form

$$\begin{aligned} &< G_1, G_2 >_H = \\ &\qquad \frac{1}{m!}\left[Tr(G_1G_2^TA^{-1} - G_1F^TA^{-1}G_2F^TA^{-1} - FG_1^TA^{-1}G_2F^TA^{-1}) + Tr(G_1F^TA^{-1})Tr(G_2F^TA^{-1})\right](\det A)^{1/2}. \end{aligned}$$

Renka and Neuberger [RN] start with a Sobolev space inner product, and ultimately come down to taking the approximate Hessian to be

$$< G_1, G_2 >_S = \frac{1}{m!}\left[Tr(G_1G_2^TA^{-1} - FG_1^TA^{-1}G_2F^TA^{-1}) + Tr(G_1F^TA^{-1})Tr(G_2F^TA^{-1})\right](\det A)^{1/2}.$$

Note that compared to the true Hessian, this drops the term

$$-Tr(G_1F^TA^{-1}G_2F^TA^{-1})$$

which is responsible for the possible nonpositivity of the Hessian. The positive semidefiniteness may be seen by writing the Sobolev form as

$$< G, G >_S = \frac{1}{m!}\left[Tr(A^{-1/2}G(I - F^TA^{-1}F)G^TA^{-1/2}) + Tr(GF^TA^{-1})^2\right](\det A)^{1/2},$$

since $I - F^TA^{-1}F = I - F^T(FF^T)^{-1}F$ is an orthogonal projection.

There is a similar idea due to Polthier and Pinkall [PP]. Their scheme minimizes the Dirichlet integral of the image simplex over the domain simplex:

$$E_\tau = \int_\tau Tr(\tilde{F}\tilde{F}^T)da.$$

where $\tilde{F}$ is the linear map from the old simplex to the new. Letting F being the old vectors and G being the new vectors, we see that

$$\tilde{F} = GF^{-1}$$

where F^{-1} is defined on the old simplex. Thus

$$E_\tau = Tr(G^T(FF^T)^{-1}G)\frac{1}{m!}(\det(FF^T))^{1/2}.$$

Or, with $A = FF^T$,

$$E_\tau = \frac{1}{m!}Tr(G^T A^{-1} G)(\det A)^{1/2} = \frac{1}{m!}Tr(GG^T A^{-1})(\det A)^{1/2}.$$

Hence the Dirichlet quadratic form is

$$< G_1, G_2 >_D = \frac{1}{m!}Tr(G_1^T A^{-1} G_2)(\det A)^{1/2} = \frac{1}{m!}Tr(G_1 G_2^T A^{-1})(\det A)^{1/2}.$$

This drops even more terms of the true Hessian. So *a priori*, the Sobolev scheme should be a little better.

The iteration scheme is to find a perturbation G of the vertices that minimizes the energy

$$E(G) = \sum_\tau \mathcal{A}_\tau + D\mathcal{A}_\tau(G) + \frac{1}{2} < G, G >_\tau$$

where $< \cdot, \cdot >_\tau$ is the approximate Hessian of your choice on facet τ.

The command `dirichlet` does one iteration minimizing the Dirichlet energy, and `sobolev` does one minimization of the Sobolev energy. In practice, both Sobolev and Dirichlet iteration are good for restoring sanity to badly messed up surfaces. However, as the surface approaches minimality, these iterations become just versions of gradient descent since the true Hessian is not used. Hence they do not converge much faster than ordinary iteration, and slower than the conjugate gradient method on ordinary iteration. The commands `dirichlet_seek` and `sobolev_seek` act as gradient descent methods, doing a line search in the solution direction to find the minimum energy. These could be plugged in to the conjugate gradient method, but I haven't done that yet.

§7.11. Calculating Forces and Torques on Rigid Bodies.

It may be of interest to know the forces and torques a liquid exerts on a solid object, for example the forces exerted on a microchip by liquid solder during the soldering phase of assembly of a circuit board. In general, the force exerted on a rigid body in a system is the negative rate of change of energy of the system with respect to translation of the rigid body while other constraints on the system remain in effect. Likewise, torque around a given axis is the negative rate of change of energy with respect to rotating the body about the axis. Below are five ways of calculating force and torque, expressed as a general rate of change of energy with respect to some parameter q. Other forces on the body due to energies not included in the Evolver datafile, such as gravity, must be calculated separately.

§7.11.1. Method 1. Finite differences.

Algorithm: Evolve the system to a minimum of energy, change the parameter q to $q + \delta q$, re-evolve to a minimum of energy, and calculate the force as

$$F = -\frac{E(q+\delta q) - E(q)}{\delta q},$$

where $E(q)$ is the minimum energy for parameter value q.

Advantages:

- Conceptually simple.
- Often requires no special preparation in the datafile.

Disadvantages:

- The system must be very close to a minimum, else energy reduction due to minimization will contaminate the energy difference due to changing q.
- Re-evolving can be time consuming.
- The finite difference formula can be of limited precision due to nonlinearity of the energy as a function of q. A more precise central difference formula could be used:

$$F = -\frac{E(q+\delta q) - E(q-\delta q)}{2\delta q}.$$

- δq needs to be chosen wisely to maximize precision. A rule of thumb is δq should be the square root of the accuracy of the energy for the one-sided difference, and the cube root for the central difference.
- The datafile may need to be modified so that q is a parameter that can be changed with the desired effect on the system.
- The system is changed, and usually needs to be restored to its original condition.

§7.11.2. Method 2. Principle of Virtual Work by Finite Differences.

If the original system is at equilibrium, it is not necessary to re-evolve the perturbed system to equilibrium. The energy difference from equilibrium of the perturbed system is only of order δq^2 in general, so it is only necessary to enforce the constraints.

Algorithm: Evolve the system to a minimum of energy, change the parameter q to $q + \delta q$, enforce any constraints (say by doing one iteration with a scale factor of 0), and calculate the force as

$$F = -\frac{E(q+\delta q) - E(q)}{\delta q}.$$

Advantages:

- Often requires no special preparation in the datafile.
- The system need not be very close to a minimum, since there is no further evolution to contaminate the energy difference.
- No time consuming re-evolving.

Disadvantages:

- The finite difference formula can be of limited precision due to nonlinearity of the energy as a function of q. A more precise central difference formula could be used:

$$F = -\frac{E(q+\delta q) - E(q-\delta q)}{2\delta q}.$$

- δq needs to be chosen wisely to maximize precision. A rule of thumb is δq should be the square root of the accuracy of the energy for the one-sided difference, and the cube root for the central difference.
- The datafile may need to be modified so that q is a parameter that can be changed with the desired effect on the system.
- The system is changed, and usually needs to be restored to its original condition.

§7.11.3. Method 3. Principle of Virtual Work using Lagrange Multipliers.

This is the same as Method 2, except that adjustment to global constraints (such as volumes) is done using the corresponding Lagrange multipliers rather than doing a projection to the constraints. Pointwise constraints must still be done, but those can be done with the 'recalc' command rather than 'g'. A brief review of Lagrange multipliers: Suppose the system is at equilibrium, and let X be the perturbation vector. Let B be the energy gradient vector, so that in matrix notation

$$E = B^T X$$

to first order. Let C_i, $i = 1, \dots, k$, be the gradient vectors of the constraints. Then at equilibrium, the energy gradient must be a linear combination of the constraint gradients

$$B = \sum_i \Gamma_i C_i$$

where the Γ_i are called the Lagrange multipliers. Now back to business. After the perturbation, let X be the deviation from the new equilibrium. Then the deviations in the constraints will be

$$\delta V_i = C_i^T X$$

so the deviation from the new equilibrium energy will be

$$E = B^T X = \sum_i \Gamma_i C_i X = \sum_i \Gamma_i \delta V_i.$$

Hence we can calculate the force as

$$F = -\frac{E(q + \delta q) - \sum_i \Gamma_i \delta V_i - E(q)}{\delta q}.$$

Algorithm: Evolve the system to a minimum of energy, change the parameter q to $q + \delta q$, enforce pointwise constraints with recalc, doing one iteration with a scale factor of 0),

$$F = -\frac{E(q + \delta q) - \sum_i \Gamma_i \delta V_i - E(q)}{\delta q}.$$

Recall that the Lagrange multipliers for body volumes are available under the body attribute "pressure".

Advantages:

- Often requires no special preparation in the datafile.
- The system need not be very close to a minimum, since there is no further evolution to contaminate the energy difference.
- No time consuming re-evolving.
- Avoids an actual projection to global constraints.

Disadvantages:

- The finite difference formula can be of limited precision due to nonlinearity of the energy as a function of q. A more precise central difference formula could be used:

$$F = -\frac{E(q + \delta q) - E(q - \delta q)}{2\delta q}$$

  where the energies include the Lagrange multiplier corrections.
- δq needs to be chosen wisely to maximize precision. A rule of thumb is δq should be the square root of the accuracy of the energy for the one-sided difference, and the cube root for the central difference.
- The datafile may need to be modified so that q is a parameter that can be changed with the desired effect on the system.
- The system is changed, and usually needs to be restored to its original condition.

§7.11.4. Method 4. Explicit forces.

The force on a body is exerted physically by pressure, surface tension, etc., so one could calculate the net force by adding all those up. That is, calculate the area of liquid contact and multiply by the pressure, find the vector force that is exerted by surface tension on each edge on the body, etc.

Algorithm: Identify all the forces acting on the body (surface tension, pressure, etc.) and calculate explicitly from the equilibrium surface.

Advantages:

- The system need not be very close to a minimum, since there is no further evolution to contaminate the energy difference.
- No time consuming re-evolving.
- Does not change the system.
- Full precision, since no differencing is done.

Disadvantages:

- From tests, this method is extremely inaccurate and slow to converge at higher refinement.
- Requires user to explicitly identify and calculate all forces.
- It can be very difficult to do some forces explicitly, such as surface tension, in the quadratic and higher order lagrange models.

§7.11.5. Method 5. Variational formulation.

This method calculates the rate of change of energy in the manner of the Calculus of Variations, setting up a vectorfield perturbation and writing the derivatives of energy and constrained quantities as integrals over the surface. By the Principle of Virtual Work, the force is

$$F = -[E'(q) - \sum_i \Gamma_i V_i'(q)].$$

Algorithm: For each component of the energy, create a named method whose value is the derivative of the component with respect to the parameter. Also do this for each global quantity. Use these derivatives to create a force quantity following the Principle of Virtual Work formula.

Advantages:

- The system need not be very close to a minimum, since there is no further evolution to contaminate the energy difference.
- No time consuming re-evolving.
- Does not change the system.
- Full precision, since no differencing is done.
- Don't have to guess at forces, since one can systematically write variations for all energy and constraint components.

Disadvantages:

- Requires user to include quantities for the variations of all energies and global constraints in the datafile.

Note on perturbations: When changing a parameter, it is best to deform the surface as uniformly as possible. If you simply change a parameter governing the height of a boundary, for example, then all the deformation is inflicted on just the adjacent facets. If the deformations are much smaller than the facet size, then this is not too significant. But for elegance, a uniform deformation is nice. For example, suppose one has a catenoid whose top is at z_{max} and whose bottom is at z_{min}. Suppose z_{max} is to be increased by dz. A uniform deformation would be

$$z \to z + \frac{z - z_{min}}{z_{max} - z_{min}} dz.$$

So a command sequence to do the perturbation would be

```
dz := 0.00001;
foreach vertex do set z z+(z-zmin)/(zmax-zmin)*dz;
zmax := zmax + dz;
```

```
recalc;
```

Note on choosing finite difference perturbation size: Let E be the energy as a function of parameter q. The one-sided finite difference force calculation is

$$F = -\frac{E(q+dq) - E(q) + \epsilon}{dq},$$

where ϵ represents the error in the energy difference due to limited machine precision. Plugging in Taylor series terms,

$$\begin{aligned} F &= -\frac{E(q) + E'(q)dq + E''(q)dq^2/2 - E(q) + \epsilon}{dq} \\ &= -E'(q) - (E''(q)dq/2 + \epsilon/dq). \end{aligned}$$

The error term is minimized for

$$dq = \sqrt{2\epsilon/E''(q)},$$

hence the rule of thumb $dq = \epsilon^{1/2}$ with error $\approx \epsilon^{1/2}$. For 15 digit machine precision, one could take $dq \approx 10^{-7}$ and expect at most 7 or 8 digits of precision in the force.

The two-sided finite difference force calculation is

$$F = -\frac{E(q+dq) - E(q-dq) + \epsilon}{2dq}.$$

Plugging in Taylor series terms,

$$\begin{aligned} F &= -\frac{E(q) + E'(q)dq + E''(q)dq^2/2 + E'''(q)dq^3/6 - (E(q) - E'(q)dq + E''(q)dq^2/2 - E'''(q)dq^3/6) + \epsilon}{2dq} \\ &= -E'(q) - (E'''(q)dq^2/6 + \epsilon/2dq). \end{aligned}$$

The error term is minimized for

$$dq = (1.5\epsilon/E'''(q))^{1/3},$$

hence the rule of thumb $dq = \epsilon^{1/3}$ with error $\approx \epsilon^{2/3}$. For 15 digit machine precision, one could take $dq \approx 10^{-5}$ and expect at most 10 digits of precision in the force.

§7.11.6. Example of variational integrals.

Here we compute the force of the `column.fe` example using variational integrals (method 5). The energy has surface tension and gravitational components,

$$E = \int\int_S T dA + \int\int_S \rho G \frac{z^2}{2}\vec{k} \cdot \vec{dA}.$$

where T is the surface tension of the free surface S, ρ is the gravitational constant, and G is the acceleration of gravity. Given a variation vectorfield $\vec{g}$, the unconstrained variation of the energy is

$$\delta E = T \int\int_S (div\vec{g} - \vec{N} \cdot D\vec{g} \cdot \vec{N})dA + \rho G \int\int_S \left(div(\frac{z^2}{2}\vec{k})\vec{g} - curl(\vec{g} \times \frac{z^2}{2}\vec{k}) \right) \cdot \vec{dA},$$

where $\vec{N}$ is the unit normal. This variation may be obtained by considering the two integrals separately. For the area integral, the variation is the divergence of $\vec{g}$ in the tangent plane, and $div\vec{g} - \vec{N} \cdot D\vec{g} \cdot \vec{N}$ is an invariant way to write that; Dg is the matrix of first partials of $\vec{g}$. The second term is obtained by noting that the original surface and the perturbed surface almost form the boundary of a thin region. The variation is obtained by applying the divergence theorem to the region, and adding in the missing boundary strip by means of Stokes' Theorem.

Of course, we have to correct for any volume change, and since the pressure P is the Lagrange multiplier relating volume and energy changes at a minimum,

$$\delta E = P\delta V = P\int\int_S \vec{g}\cdot d\vec{A}.$$

Projecting volume back to the volume constraint subtracts this, so the net change is

$$\delta E = T\int\int_S (div\vec{g} - \vec{N}\cdot D\vec{g}\cdot\vec{N})dA + \rho G\int\int_S \left((div\frac{z^2}{2}\vec{k})\vec{g} - curl(\vec{g}\times\frac{z^2}{2}\vec{k})\right)\cdot d\vec{A} - P\int\int_S \vec{g}\cdot d\vec{A}.$$

We shall take the perturbation to be $\vec{g} = (z+ZH)\vec{j}/2ZH$, which leaves the bottom plane fixed and gives a unit shift to the top plane. It is important when working with piecewise linear surfaces that the perturbations be likewise piecewise linear. The force, then, is

$$\delta E = T\int\int_S -\frac{N_z N_y}{2ZH}dA + \rho G\int\int_S -\frac{z^2}{4ZH}\vec{k}\cdot d\vec{A} - P\int\int_S \frac{z+ZH}{2ZH}\vec{j}\cdot d\vec{A}.$$

The $N_z N_y$ term requires the use of the "`facet_general_integral`" method, which integrates an arbitrary scalar function of position and normal vector over facets. This method requires that the integrand be homogeneous of degree 1 in the unnormalized facet normal $\vec{N}$, i.e. that the integrand be

$$-\frac{1}{2ZH}\frac{N_z N_y}{|\vec{N}|}.$$

The third term of δE is automatically zero by the Divergence Theorem, since the integrand is zero on the top and bottom pads. The two remaining terms are defined as two method instances in the datafile below, and combined into one quantity "`forcey`".

Example

The datafile below calculates the vertical force exerted by a catenoid (without body), for which exact values can be calculated analytically. All five methods are done. Following is a table of the errors in the force as a function of refinement and the representation used. "Lagrange order" refers to the order of polynomial used to represent facets; 1 is linear, and 2 is quadratic.

```
// catforce.fe

// Evolver data for catenoid.
// For testing vertical force on upper ring calculated various
// ways and compared to exact solution.

PARAMETER  ZMAX = 0.7
PARAMETER  ZMIN = -0.7
PARAMETER  RMAX = cosh(zmax)
parameter  true_area =   pi*(zmax+sinh(2*zmax)/2-zmin-sinh(2*zmin)/2)

// for putting vertices exactly on catenoid
constraint 1
formula:  sqrt(x^2+y^2) = cosh(z)

// for restoring after perturbation
define vertex attribute old_coord real[3]
```

```
// following method instances and quantity for vertical force
method_instance darea method facet_general_integral
scalar_integrand:  1/(zmax-zmin)*(x4^2+x5^2)/sqrt(x4^2+x5^2+x6^2)

quantity forcez info_only global_method darea

boundary 1 parameters 1     //  upper ring
x1:    RMAX * cos(p1)
x2:    RMAX * sin(p1)
x3:    ZMAX


boundary 2 parameters 1    //   lower ring
x1:    RMAX * cos(p1)
x2:    RMAX * sin(p1)
x3:    ZMIN


vertices   // given in terms of boundary parameter
1    0.00  boundary 1   fixed
2    pi/3  boundary 1   fixed
3  2*pi/3  boundary 1   fixed
4    pi    boundary 1   fixed
5  4*pi/3  boundary 1   fixed
6  5*pi/3  boundary 1   fixed
7    0.00  boundary 2   fixed
8    pi/3  boundary 2   fixed
9  2*pi/3  boundary 2   fixed
10   pi    boundary 2   fixed
11 4*pi/3  boundary 2   fixed
12 5*pi/3  boundary 2   fixed

edges
1    1  2  boundary 1   fixed
2    2  3  boundary 1   fixed
3    3  4  boundary 1   fixed
4    4  5  boundary 1   fixed
5    5  6  boundary 1   fixed
6    6  1  boundary 1   fixed
7    7  8  boundary 2   fixed
8    8  9  boundary 2   fixed
9    9  10 boundary 2   fixed
10   10 11 boundary 2   fixed
11   11 12 boundary 2   fixed
12   12 7  boundary 2   fixed
13   1  7
14   2  8
15   3  9
16   4  10
17   5  11
18   6  12

faces
```

```
1   1 14 -7 -13
2   2 15 -8 -14
3   3 16 -9 -15
4   4 17 -10 -16
5   5 18 -11 -17
6   6 13 -12 -18


read
hessian_normal

// For saving coordinates before perturbation
save_coords :=  foreach vertex vv do
set vv.old_coord[1] x;
set vv.old_coord[2] y;
set vv.old_coord[3] z;


// For restoring coordinates after perturbation
restore_coords :=  foreach vertex vv do
set vv.x old_coord[1];
set vv.y old_coord[2];
set vv.z old_coord[3];



// Force by central difference of energy minima
method1 :=  save_coords;
dzmax := 0.00001;
zmax := zmax - dzmax;
optimize 1;
hessian; hessian;
lo_energy := total_energy;
restore_coords;
zmax := zmax + 2*dzmax;
hessian; hessian;
hi_energy := total_energy;
restore_coords;
zmax := zmax - dzmax;
force1 := -(hi_energy - lo_energy)/2/dzmax;


// Force by central difference and Principle of Virtual Work
method2 :=  save_coords;
old_scale := scale;
dzmax := 0.00001;
zmax := zmax - dzmax;
recalc; m 0; g;
lo_energy := total_energy;
restore_coords;
zmax := zmax + 2*dzmax;
recalc; m 0; g;
hi_energy := total_energy;
```

```
restore_coords;
zmax := zmax - dzmax;
scale := old_scale; optimize 1;
force2 := -(hi_energy - lo_energy)/2/dzmax;


// Force by central difference and Principle of Virtual Work and
// Lagrange multipliers
method3 :=  save_coords;
dzmax := 0.00001;
zmax := zmax - dzmax;
recalc;
lo_energy := total_energy ;
restore_coords;
zmax := zmax + 2*dzmax;
recalc;
hi_energy := total_energy ;
restore_coords;
zmax := zmax - dzmax;
force3 := -(hi_energy - lo_energy)/2/dzmax;

// Using same overall perturbation as forcez
method3a :=  save_coords;
dzmax := 0.00001;
set vertex z z-(z-zmin)/(zmax-zmin)*dzmax;
zmax := zmax - dzmax;
recalc;
lo_energy := total_energy ;
restore_coords;
zmax := zmax + dzmax;
set vertex z z+(z-zmin)/(zmax-zmin)*dzmax;
zmax := zmax + dzmax;
recalc;
hi_energy := total_energy ;
restore_coords;
zmax := zmax - dzmax;
force3a:= -(hi_energy - lo_energy)/2/dzmax;


// Force by explicit calculation of forces.    Surface tension force
//  only here, since pressure not acting on chip in lateral direction.
method4 :=
force4 := sum(edge ee where on_boundary 1,
(ee.y*ee.facet[1].x - ee.x*ee.facet[1].y)/ee.facet[1].area);


// Force by variational integrals:
method5 :=  force5 := -forcez.value;


// all methods at once
methods :=
quiet on; method1; method2; method3; method3a;
```

```
if (linear) then method4;
method5;
quiet off;
printf "\n\nSummary:\n" ;
printf "True force:                %-#20.15g\n",-2*pi*rmax/cosh(zmax);
printf "Force by method 1:    %-#20.15g\n",force1;
printf "Force by method 2:    %-#20.15g\n",force2;
printf "Force by method 3:    %-#20.15g\n",force3;
printf "Force by method 3a:  %-#20.15g\n",force3a;
if (linear) then printf "Force by method 4:    %-#20.15g\n",force4;
printf "Force by method 5:    %-#20.15g\n",force5;
```

Table 1. Errors in calculated force by method 3 (uniform stretch with Lagrange multipliers) for $dq = 10^{-5}$ and central difference:

	Lagrange order							
refinement	1	2	3	4	5	6	7	8
0	8.5e-1	3.8e-3	6.2e-3	3.6e-3	1.7e-4	1.7e-5	1.1e-5	6.1e-6
1	2.0e-1	1.8e-3	3.0e-4	1.0e-5	1.0e-6	7.8e-8	1.5e-8	1.7e-9
2	5.1e-2	1.2e-4	2.1e-5	1.8e-7	1.3e-8	3.2e-10	5.5e-10	3.8e-10
3	1.3e-2	6.9e-6						
4	4.3e-3	4.3e-7						
5	8.2e-4							

Table 2. Time of calculation (r; g5; hessian; hessian; hessian) in seconds:

	Lagrange order							
refinement	1	2	3	4	5	6	7	8
0			1.272	3.094	9.204	20.289	32.598	106.563
1	0.340	3.184	4.777	12.127	36.212	80.155	131.710	425.142
2	1.291	12.277	19.328	49.190	145.850	326.570	523.593	1789.710
3	6.479	51.384	83.740	209.982				
4	19.839	227.046						
5	98.441							

§7.12. Equiangulation.

Triangulations work best when the facets are close to equilateral (that is, equiangular). Given a set of vertices, how do you make them into a triangulation that has triangles as nearly as possible equilateral? In the plane, the answer is the Delaunay triangulation, in which the circumcircle of each triangle contains no other vertex. See [S]. It is almost always unique. It can be constructed by local operations beginning with any triangulation. Consider any edge as the diagonal of the quadrilateral formed by its adjacent triangles. If the angles of the two vertices off of the edge add to more than π, then the circumcircle criterion is violated, and the diagonal should be switched to form a replacement pair of facets.

Suppose now we have a triangulation of a curved surface in space. For any edge with two adjacent facets, we switch the edge to the other diagonal of the skew quadrilateral if the sum of the angles at the off vertices is more than π. Let a be the length of the common edge, b and c the lengths of the other sides of one triangle, and d and e the lengths of the other sides of the other triangle. Let θ_1 and θ_2 be the off angles. Then by the law of cosines

$$a^2 = b^2 + c^2 - 2bc\cos\theta_1, \qquad a^2 = d^2 + e^2 - 2de\cos\theta_2. \tag{1}$$

The condition $\theta_1 + \theta_2 > \pi$ is equivalent to $\cos\theta_1 + \cos\theta_2 < 0$. So we switch if

$$\frac{b^2 + c^2 - a^2}{bc} + \frac{d^2 + e^2 - a^2}{de} < 0. \tag{2}$$

This is guaranteed to terminate since a switch reduces the radii of the circumcircles and there are a finite number of triangulations on a finite number of vertices. When using the `u` command to equiangulate, you should use it several times until no changes happen. If you do get a genuine infinite loop, report it as a bug.

§7.13. Dihedral angle.

The dihedral angle attribute of an edge is defined in the soap film model for any edge with two adjacent facets. (Other edges return a dihedral angle of 0.) Letting the common edge be $\vec{a}$, and letting $\vec{b}$ and $\vec{c}$ be the other edge vectors from the base of $\vec{a}$, we use the inner product of 2-vectors:

$$\cos\theta = \frac{< \vec{a}\wedge\vec{b}, \vec{c}\wedge\vec{a} >}{||\vec{a}\wedge\vec{b}||||\vec{c}\wedge\vec{a}||} = \frac{\begin{vmatrix} \vec{a}\cdot\vec{c} & \vec{a}\cdot\vec{a} \\ \vec{b}\cdot\vec{c} & \vec{b}\cdot\vec{a} \end{vmatrix}}{\begin{vmatrix} \vec{a}\cdot\vec{a} & \vec{a}\cdot\vec{b} \\ \vec{b}\cdot\vec{a} & \vec{b}\cdot\vec{b} \end{vmatrix}^{1/2} \begin{vmatrix} \vec{c}\cdot\vec{c} & \vec{c}\cdot\vec{a} \\ \vec{a}\cdot\vec{c} & \vec{a}\cdot\vec{a} \end{vmatrix}^{1/2}}.$$

If the denominator is 0 or $\cos\theta > 1$, then $\theta = 0$. If $\cos\theta < -1$, then $\theta = \pi$. Note that this definition works in any ambient dimension.

In the string model, this is the angle from straightness of two edges at a vertex. If there are less than two edges, the value is 0. If two or more edges, the value is 2*asin(F/2), where F is the magnitude of the net force on the vertex, assuming each edge has tension 1. Upper limit clamped to pi.

§7.14. Area normalization.

By default, the gradients of quantities like energy and volume are calculated simply as the gradient of the quantity as a function of vertex position. This gives the force on a vertex, for example. But sometimes it is desirable to have force per area instead, for example in directly estimating mean curvature. The mean curvature vectorfield $\vec{h}$ of a surface is defined to be the gradient of the area of the surface in the sense that if the surface is deformed with an instantaneous velocity $\vec{u}$ then the rate of change of surface area is

$$\frac{dA}{dt} = \int\int_{surface} \vec{u}\cdot\vec{h}dA. \tag{1}$$

According to the triangulation formulation, with $\vec{F}_v$ being the force on vertex v, the rate of change is

$$\frac{dA}{dt} = \sum_v u(v)\cdot -\vec{F}_v. \tag{2}$$

Hence as approximation to $\vec{h}$, we take

$$\vec{h} = -\vec{F}_v/dA_v \tag{3}$$

and take the area dA_v associated with a vertex to be one-third of the total areas of the facets surrounding the vertex. Since each facet has three vertices, this allocates all area. Command "a" can be used to toggle area normalization.

§7.15. Hidden surfaces.

This section describes the algorithm used by the painter graphics interface to hide hidden surfaces.

1. Each facet has the viewing transformation applied to it.

2. Each facet has its maximum and minimum in each dimension recorded (bounding box).

3. Facets are sorted on their maximum depth (rear of bounding box).

4. List is traversed in depth order, back to front. If a facet's bounding box does not overlap any others', it is drawn. If it does overlap, a linear program tries to find some point where one facet is directly in front of the other. If there is such a point and the order there is backward, the facets are switched on the list. The

traversal then restarts there. If it makes over 10 switches without drawing a facet, it gives up and draws the current facet anyway.

§7.16. Extrapolation.

The final energy value at refinement step n is saved as E_n. Assuming the difference from minimum energy follows a power law of unknown power, three successive values are used to extrapolate to the minimum energy E_0:

$$\begin{aligned} E_n &= E_0 + ac^b, \\ E_{n-1} &= E_0 + a(2c)^b, \\ E_{n-2} &= E_0 + a(4c)^b. \end{aligned} \tag{1}$$

Hence

$$E_0 = E_n - \frac{(E_n - E_{n-1})^2}{E_n - 2E_{n-1} + E_{n-2}}. \tag{2}$$

§7.17. Curvature test.

This only applies to the quadratic model.

§7.18. Annealing (jiggling).

This happens at each command `j`, or at each iteration if the jiggling option (command `J`)is activated. Each coordinate of each non-`FIXED` vertex is moved by

$$\delta x = gTL \tag{1}$$

where g is a random value from the standard Gaussian distribution (calculated from the sum of five random values from the uniform distribution on [0,1]), T is the current temperature, and L is a characteristic length that starts as the diameter of the surface and is cut in half at each refinement.

§7.19. Long wavelength perturbations (long jiggling).

This is command `longj`. An amplitude $\vec{A}$ and a wavelength $\vec{L}$ are chosen at random from a unit sphere by picking random vectors in a cube of side 2 centered at the origin and rejecting attempts that are outside the unit sphere. $\vec{A}$ and $\vec{L}$ are then multiplied by the size of the surface to give them the proper scale. The wavelength is inverted to a wavevector $\vec{w}$. The amplitude is multiplied by the temperature. A random phase ψ is picked. Then each vertex $\vec{v}$ is moved by $\vec{A}\sin(\vec{v}\cdot\vec{w}+\psi)$.

§7.20. Homothety.

This scales the total volume of all bodies to 1 by multiplying each coordinate by

$$\left(\sum_{bodies} V_{body}\right)^{-1/3}. \tag{1}$$

In the string model, the factor is

$$\left(\sum_{bodies} V_{body}\right)^{-1/2}. \tag{2}$$

§7.21. Popping non-minimal edges.

All facets are assumed to have equal tension. This looks for edges with more that three facets that are not fixed or on boundaries or constraints. When found, such an edge is split with a new facet in between. The two closest old facets are attached to the new edge. This is repeated until only three facets are on the original edge. Each split is propagated along the multiple junction line as far as possible. If it is impossible to propagate beyond either endpoint, the edge is subdivided to provide a vertex which can be split.

§7.22. Popping non-minimal vertex cones.

String model.

Any vertex with four or more edges is tested. The edges into the vertex are separated into two contiguous groups. The endpoints of one group are transfered to a new vertex, and a short edge is introduced between the new and old vertex. All groupings are tested to see which one pulls apart the most. This method works when the edges have non-equal tensions.

Soapfilm model.

All facets are assumed to have equal tension. This assumes that all edges have at most three facets adjoining, so edge popping should be done first. Here the facet and edge structure around each vertex is analyzed to find which have the wrong topology. The analysis is done by looking at the network formed by the intersection of the facets and edges containing the vertex with the surface of a small sphere around the vertex. The numbers of sides of the cells of the network are counted. A simple plane vertex has two cells of one side each. A triple edge vertex has three cells of two sides each. A tetrahedral point has four cells with three sides each. Any other configuration is popped.

The popping itself is done by replacing the vertex with a hollow formed by truncating each cell-cone except the cell with the largest solid angle. In case the network is disconnected, the solid angles of all the cells will add up to over 4π. Then the vertex is duplicated and the different components are assigned to different vertices. This lets necks shrink to zero thickness and pull apart.

§7.23. Refining.

First, all edges are subdivided by inserting a midpoint. Hence all facet temporarily have six sides. For edges on constraints, the midpoints get the same set of constraints, and are projected to them. For edges on boundaries, the parameters of the midpoint are calculated by projecting the vector from the edge tail to the midpoint back into the parameter space and adding that to the tail parameters. This avoids averaging parameters of endpoints, which gives bad results when done with boundaries that wrap around themselves, such as circles. The new edges and the midpoint inherits their attributes from the edge.

Second, each facet is subdivided into four facets by connecting the new midpoints into a central facet. The new edges inherit their attributes from the facet.

§7.24. Refining in the simplex model.

The goal is to divide an n-dimensional simplex into 2^n equal volume simplices, although these cannot be congruent for $n > 2$. First, each 1-dimensional edge is subdivided by inserting a new vertex. This vertex inherits the intersection of the attributes of the edge endpoints (i.e. fixity and constraints). Second, the $n+1$ tips of the old simplex are lopped off, leaving a "core". This core is recursively subdivided. An edge midpoint is picked as a base point. Faces of the core not adjacent to the base point are either simplices (where tips got lopped off) or cores of one lower dimension. The core is given a conical subdivision from the base point to simplicial subdivisions of the nonadjacent faces. The base point is always picked to have the lowest id number, in order that refinements of adjacent simplices are consistent.

§7.25. Removing tiny edges.

The edge list is scanned, looking for edges shorter than the cutoff length. It is NOT removed if any of the following are true:

1. Both endpoints are `FIXED`.
2. The edge is `FIXED`.
3. The boundary and constraint attributes of the endpoints and the edge are not identical. For an edge that passes these tests, one endpoint is eliminated. If one endpoint is `FIXED`, that is the one kept. All the facets around the edge are eliminated (collapsing their other two edges to one), and the edge itself is eliminated.

§7.26. Weeding small triangles.

This is meant to help eliminate narrow triangles that tiny edge removal can't because they don't have short edges. This procedure finds the shortest edge of a triangle and eliminates it by the same process as the regular tiny edge removal. If that fails, it tries the other edges.

§7.27. Vertex averaging.

For each vertex, this computes a new position as the area-weighted average of the centroids of the facets adjoining the vertex. `FIXED` vertices or vertices on boundaries are not moved. Vertices on triple lines are only averaged with neighboring vertices along the triple line. Tetrahedral points do not get moved. For normal vertices, the scheme is as follows: If vertex v is connected by edges to vertices $v_1, \ldots, v_n$, then the new position is calculated as

$$\vec{v}_{avg} = \frac{\sum_{facets} area \cdot \vec{x}_{centroid}}{\sum_{facets} area}. \tag{1}$$

The volume one one side of all the facets around the vertex calculated as a cone from the vertex is

$$V = \sum_{facets\ f} \vec{v} \cdot \vec{N}_f \tag{2}$$

where $\vec{N}_f$ is the facet normal representing its area. The total normal $\vec{N}$ is

$$\vec{N} = \sum_{facets\ f} \vec{N}_f. \tag{3}$$

To preserve volume, we subtract a multiple λ of the total normal to the average position:

$$(\vec{v}_{avg} - \lambda \vec{N}) \cdot \vec{N} = \vec{v} \cdot \vec{N}, \tag{4}$$

so

$$\lambda = \frac{\vec{v}_{avg} \cdot \vec{N} - \vec{v} \cdot \vec{N}}{\vec{N} \cdot \vec{N}}. \tag{5}$$

Then the new vertex position is

$$\vec{v}_{new} = (\vec{v}_{avg} - \lambda \vec{N}). \tag{6}$$

Constrained vertices are then projected to their constraints.

The "`rawv`" command will do vertex averaging as just described, but without conserving volumes. The "`rawestv`" command will also ignore all restrictions about like constraints and triple edges, but it will not move fixed points or points on boundaries. Points on constraints will be projected back to them.

§7.28. Zooming in on vertex.

First, all vertices beyond the cutoff distance from the given vertex are deleted. Then all edges and facets containing any deleted vertices are deleted. Any remaining edge that had a facet deleted from it is made `FIXED`.

§7.29. Mobility and approximate curvature.

This section derives the resistance and mobility matrices used in approximate curvature. The derivation is done in the full generality of vertex constraints, boundaries, volume and quantity constraints, and effective area.

First, we must define the inner product or metric on global vectors, which is the resistance matrix. Let $\vec{W}$ be a global vectorfield, that is, $\vec{W}$ has a component vector $\vec{W}_v$ at each vertex v. At each vertex v, let $\vec{U}_{v,i}$ be a basis for the degrees of freedom of motion of vertex v due to level set constraints or boundaries. Consider one edge e in the string model. Let P be the projection operator onto the normal of e if effective area is in effect, and let P be the identity map if not. Then the inner product of two vector fields is

$$\begin{aligned} <\vec{W},\vec{V}> &= L\int_0^1 P((1-t)\vec{W}_a + t\vec{W}_b)\cdot P((1-t)\vec{V}_a + t\vec{V}_b)dt \\ &= L(\frac{1}{3}P\vec{W}_a\cdot P\vec{V}_a + \frac{1}{6}P\vec{W}_a\cdot P\vec{V}_b + \frac{1}{6}P\vec{W}_b\cdot P\vec{V}_a + \frac{1}{3}P\vec{W}_b\cdot P\vec{V}_b). \end{aligned} \tag{1}$$

If the vectors are expressed in terms of the bases U_v with coefficient column matrices W_v and V_v, then

$$\begin{aligned} <\vec{W},\vec{V}>= L(\frac{1}{3}\sum_{i,j} W_{ai}\vec{U}_{a,i}\cdot P\vec{U}_{a,j}V_{aj} + \frac{1}{6}\sum_{i,j} W_{ai}\vec{U}_{a,i}\cdot P\vec{U}_{b,j}V_{bj} \\ +\frac{1}{6}\sum_{i,j} W_{bi}\vec{U}_{b,i}\cdot P\vec{U}_{a,j}V_{aj} + \frac{1}{3}\sum_{i,j} W_{bi}\vec{U}_{b,i}\cdot P\vec{U}_{b,j}V_{bj}). \end{aligned} \tag{2}$$

So the components of the resistance matrix S are

$$\begin{aligned} S_{ai,aj} &= \frac{L}{3}\vec{U}_{a,i}\cdot P\vec{U}_{a,j}, \\ S_{ai,bj} &= \frac{L}{6}\vec{U}_{a,i}\cdot P\vec{U}_{b,j}. \end{aligned} \tag{3}$$

In the soapfilm model, for a facet of area A,

$$\begin{aligned} S_{ai,aj} &= \frac{A}{6}\vec{U}_{a,i}\cdot P\vec{U}_{a,j}, \\ S_{ai,bj} &= \frac{A}{12}\vec{U}_{a,i}\cdot P\vec{U}_{b,j}. \end{aligned} \tag{4}$$

The total matrix S is formed by adding the contributions from all edges. Note that S has a block on its diagonal for each vertex and a block for each pair of edges joined by an edge. Hence it is reasonably sparse. S is a symmetric positive semidefinite matrix, positive definite if effective area is not in effect.

Now suppose we have a form H to convert to a vector. H is represented as a covector H_v at each vertex v. The mobility matrix $M = (M_{bj,ai})$ is the inverse of S. The first step is to find the coefficients in the dual basis of U by contracting:

$$h_{a,i} =< \vec{U}_{a,i}, H_v > . \tag{5}$$

The U basis components of the vector dual to H are then

$$c_{b,j} = M_{bj,ai}h_{a,i}. \tag{6}$$

Actually, the components are found by solving the sparse system

$$S_{ai,bj}c_{a,i} = h_{a,i}. \tag{7}$$

The foregoing ignores volume and quantity constraints. Suppose B_k are gradient forms for the volume and quantity constraints. We need to project the motion in such a way as to guarantee that when the velocity is zero, the energy gradient is a linear combination of constraint gradients.

$$proj(H) = H - \lambda_k B_k. \tag{8}$$

The projected velocity is

$$V = M(H - \lambda_k B_k). \tag{9}$$

The condition for motion tangent to the constraints is

$$B_m M(H - \lambda_k B_k) = 0 \qquad \text{for each } m. \tag{10}$$

Therefore

$$B_m M B_k \lambda_k = B_m M H, \tag{11}$$

and so

$$\lambda_k = (B_m M B_k)^{-1} B_m M H. \tag{12}$$

Chapter 8. Named Methods and Quantities.

§8.1. Introduction.

Named methods and quantities are a systematic scheme of calculating global quantities such as area, volume, and surface integrals that supplements the original ad hoc scheme in the Evolver. Briefly, *named methods* are built-in functions, and *named quantities* are combinations of *instances* of methods. See the `ringblob.fe` datafile for an example. The original ad hoc calculations are still the default where they exist, but all new quantities are being added in the named quantity scheme. Some new features will work only with named quantities, in particular many Hessian calculations. To convert everything to named quantities, start Evolver with the `-q` option or use the `convert_to_quantities` command. This has not been made the default since named quantities can be slower than the originals. It is planned that eventually all energies and global constraints will be converted to named quantity system. However, existing syntax will remain valid wherever possible. Starting Evolver with the -q option will do this conversion now.

The sample datafiles `qcube.fe`, `qmound.fe`, and `ringblob.fe` contains some examples of named quantities and instances. The first two are quantity versions of `cube.fe` and `mound.fe`. These illustrate the most general and useful methods, namely facet_vector_integral, facet_scalar_integral, and edge_vector_integral, rather than the faster but more specialized methods such as facet_area. My advice is that the user stick to the old implicit methods for area, volume, and gravitational energy, and use named quantities only for specialized circumstances.

§8.2. Named methods.

A *method* is a way of calculating a scalar value from some particular type of element (vertex, edge, facet, body). Each method is implemented internally as a set of functions for calculating the value and its gradient as a function of vertex positions. The most common methods also have Hessian functions. Methods are referred to by their names. See the **Implemented methods** list below for the available methods. Adding a new method involves writing C routines to calculate the value and the gradient (and maybe the Hessian) as functions of vertex coordinates, adding the function declarations to `quantity.h`, and adding a structure to the method declaration array in `quantity.c`. All the other syntax for invoking it from the datafile is already in place.

§8.3. Method instances.

A *method instance* is the sum of a particular method applied to a particular set of geometric elements. Some methods (like `facet_area`) are completely self-contained. Others (like `facet_vector_integral`) require the user to specify some further information. For these, each instance has a specification of this further information. Method instances are defined in the datafile, and may either be unnamed parts of named quantity definitions or separate named method instances for inclusion in named quantities. The separate named version is useful if you want to inspect instance values for the whole surface or individual elements.

An instance total value can be printed with the `A` command, or may be referred to as "instance-name.value" in commands. The instance name itself may be used as an element attribute. For example, supposing there is an instance named `moment`, which applies to facets. Then typical commands would be

```
print moment.value
print facet[3].moment
list facet where moment > 0.1
```

Every method instance has a *modulus*, which is multiplied times the basic method value to give the instance value. A modulus of 0 causes the entire instance calculation to be omitted whenever quantities are calculated. The modulus may be set in the datafile or with the `A` command or by assignment. Example commands:

```
print moment.modulus
moment.modulus := 1.3
```

Some methods, those that logically depend on the orientation of the element, can be applied with a relative orientation. When applied to individual elements in the datafile, a negative orientation is indicated by a '-' after the instance name. When applied at runtime with the `set` command, the orientation will be negative if the element is generated with negative orientation, i.e. `set body[1].facet method_instance qqq`. The methods currently implementing this feature are: edge_vector_integral, string_gravity,facet_vector_integral, facet_2form_integral, facet_volume,facet_torus_volume, simplex_vector_integral, simplex_k_vector_integral, edge_k_vector_integral, gravity_method, and full_gravity_method.

§8.4. Named quantities.

A *named quantity* is the sum total of various method instances, although usually just one instance is involved. The instances need not apply to the same type of element; for example, both facet and edge integrals may be needed to define a volume quantity. Each named quantity is one of three types:

`energy` quantities which are added to the total energy of the surface;

`fixed` quantities that are constrained to a fixed target value (by Newton steps at each iteration); and

`info_only` quantities whose values are merely reported to the user. This type is initially set in a quantity's datafile declaration. A quantity can be toggled between fixed and info_only with the "`fix` *quantityname*" and "`unfix` *quantityname*" commands.

The value of a quantity may be displayed with the `A` or `v` commands, or as an expression "*quantityname*`.value`". Furthermore, using the quantity name as an element attribute evaluates to the sum of all the applicable component instance values on that element. For example, supposing there is a quantity named `vol`, one could do

```
print vol.value
print facet[2].vol
histogram(facet,vol)
```

Each quantity has a *modulus*, which is just a scalar multiplier for the sum of all instance values. A modulus of 0 will turn off calculation of all the instances. The modulus can be set in the datafile declaration, with the `A` command, or by assignment:

```
quantityname.modulus := 1.2
```

Each fixed quantity has a target value, to which the Evolver attempts to constraint the quantity value. Each time an iteration is done (`g` command or the various Hessian commands), Newton's Method is used to project the surface to the constrained values. The target value can be displayed with the `A` or `v` commands, or as "*quantityname*`.target`". It can be changed with the `A` command or by assignment. Example:

```
print qname.target
qname.target := 3.12
```

A quantity can have a constant value added to it, similar to the body attribute `volconst`. This quantity attribute is also called `volconst`. It is useful for adding in known values of say integrals that are omitted from the actual calculation. It can be set in the quantity's datafile definition, or by an assignment command.

Each fixed quantity has a Lagrange multiplier associated to it. The Lagrange multiplier of a constraint is the rate of energy change with respect to the constraint target value. For a volume constraint, the Lagrange multiplier is just the pressure. Lagrange multipliers are calculated whenever an iteration step is done. They may be displayed by the `v` command in the "`pressure`" column, or as an expression "*quantityname*`.pressure`".

A fixed quantity can have a tolerance attribute, which is used to judge convergence. A surface is deemed converged when the sum of all ratios of quantity discrepancies to tolerances is less than 1. This sum also includes bodies of fixed volume. If the tolerance is not set or is negative, the value of the variable `target_tolerance` is used, which has a default value of 0.0001.

§8.5. Implemented methods.

The currently implemented methods are listed here, grouped somewhat by nature. Within each group, they are more or less in order of importance.

0-dimensional ¡ul¿

`<a href="#vertex_scalar_integral">vertex_scalar_integral` ¡/ul¿

1-dimensional

edge_tension, edge_length
density_edge_length
edge_scalar_integral
edge_vector_integral
edge_general_integral
edge_area
edge_torus_area
string_gravity
hooke_energy
hooke2_energy
hooke3_energy
local_hooke_energy
dihedral_hooke
sqcurve_string
metric_edge_length
klein_length

2-dimensional

facet_tension, facet_area
density_facet_area
facet_volume
facet_scalar_integral
facet_vector_integral
facet_2form_integral
facet_general_integral
facet_torus_volume
gravity_method
full_gravity_method
facet_area_u
density_facet_area_u
gap_energy
metric_facet_area
klein_area
dirichlet_area
sobolev_area
pos_area_hess
spherical_area

2-D Curvatures

mean_curvature_integral

sq_mean_curvature

eff_area_sq_mean_curvature

normal_sq_mean_curvature

star_sq_mean_curvature

star_eff_area_sq_mean_curvature

star_normal_sq_mean_curvature

gauss_curvature_integral

sq_gauss_curvature

General dimensions

simplex_vector_integral

simplex_k_vector_integral

edge_k_vector_integral

Knot energies

knot_energy

uniform_knot_energy

uniform_knot_energy_normalizer

uniform_knot_normalizer1

uniform_knot_normalizer2

edge_edge_knot_energy, edge_knot_energy

edge_knot_energy_normalizer

simon_knot_energy_normalizer

facet_knot_energy

facet_knot_energy_fix

buck_knot_energy

proj_knot_energy

circle_knot_energy

sphere_knot_energy

sin_knot_energy

curvature_binormal

ddd_gamma_sq

edge_min_knot_energy

true_average_crossings

true_writhe

twist

writhe

curvature_function

Weird and miscellaneous

wulff_energy

linear_elastic

linear_elastic_B

area_square

stress_integral

```
carter_energy
charge_gradient
johndust
ackerman
```

§8.6. Method descriptions.

The descriptions below of the individual methods give a mathematical definition of the method, what type of element it applies to, definition parameters, which types of models it applies to, any restrictions on the dimension of ambient space, and whether the method has a Hessian implemented. Fuller mathematical formulas for some of the methods may be found in the Technical Reference chapter. Unless specifically noted, a method has the gradient implemented, and hence may be used for an energy or a constraint. The definition parameters are usually scalar or vector integrands (see the Datafile chapter for full syntax). Some methods also depend on global variables as noted. The sample datafile declarations given are for simple cases; full syntax is given in the Datafile chapter. Remember in the samples that for quantities not declared global, the quantity has to be individually applied to the desired elements.

0-dimensional

vertex_scalar_integral. Description: Function value at a vertex. This actually produces a sum over vertices, but as a mathematician, I think of a sum over vertices as a point-weighted integral. Element: vertex. Parameters: `scalar_integrand`. Models: linear, quadratic, Lagrange, simplex. Ambient dimension: any. Hessian: yes. Example datafile declaration:

```
quantity point_value energy method vertex_scalar_integral
scalar_integrand:  x^2 + y^2 - 2x + 3
```

1-dimensional

edge_tension or **edge_length**. Description: Length of edge. Quadratic model uses Gaussian quadrature of order `integral_order_1D`. Element: edge. Parameters: none. Models: linear, quadratic, Lagrange. Ambient dimension: any. Hessian: yes. Example datafile declaration:

```
quantity len energy method edge_length global
```

density_edge_length. Description: Length of edge. Quadratic model; uses Gaussian quadrature of order `integral_order_1D`. Element: edge. Parameters: none. Models: linear, quadratic, Lagrange. Ambient dimension: any. Hessian: yes. Example datafile declaration:

```
quantity len energy method density_edge_length global
```

edge_scalar_integral. Description: Integral of a scalar function over arclength. Uses Gaussian quadrature of order `integral_order_1D`. Element: Parameters: Models: linear, quadratic, Lagrange. Ambient dimension: any. Hessian: yes. Type: edge. Parameters: `scalar_integrand`. Example datafile declaration:

```
quantity edge_sint energy method edge_scalar_integral
scalar_integrand:  x^2 - 3*y + 4
```

edge_vector_integral. Description: Integral of a vectorfield over an oriented edge. Uses Gaussian quadrature of order `integral_order_1D`. Element: edge. Parameters: `vector_integrand`. Models: linear, quadratic, Lagrange. Ambient dimension: any. Hessian: yes. Orientable: yes. Example datafile declaration:

```
quantity edge_vint energy method edge_vector_integral
```

```
vector_integrand:
q1:  0
q2:  0
q3:  z^2/2
```

edge_general_integral. Description: Integral of a scalar function of position and tangent over an edge. The components of the tangent vector are represented by continuing the coordinate indices. That is, in 3D the position coordinates are x1,x2,x3 and the tangent components are x4,x5,x6. For proper behavior, the integrand should be homogeneous of degree 1 in the tangent components. Uses Gaussian quadrature of order `integral_order_1D`. Element: edge. Parameters: `scalar_integrand`. Models: linear, quadratic, Lagrange. Ambient dimension: any. Hessian: yes. Example datafile declaration: the edge length in 3D could be calculated with this quantity:

```
quantity arclength energy method edge_general_integral
scalar_integrand:  sqrt(x4^2 + x5^2 + x6^2)
```

edge_area. Description: For calculating the area of a body in the string model. Implemented as the exact integral $\int -y dx$ over the edge. Valid for torus model, but not general symmetry groups. Element: edge. Parameters: none. Models: linear, quadratic, Lagrange. Ambient dimension: 2. Hessian: yes. Example datafile declaration:

```
quantity cell1_area fixed = 1.3 method edge_area
```

edge_torus_area. Description: For 2D torus string model body area calculations. Contains adjustments for torus wraps. Element: edge. Parameters: none. Models: torus; string; linear,quadratic,Lagrange. Ambient dimension: 2. Hessian: no. Example datafile declaration:

```
quantity cell_area fixed = 1.3 method edge_torus_area
```

string_gravity. Description: To calculate the gravitational potential energy of a body in the string model. Uses differences in body densities. Does not use gravitational constant G as modulus (unless invoked as internal quantity by `convert_to_quantities`). Element: edge. Parameters: none. Models: string linear, quadratic, lagrange. Ambient dimension: 2. Hessian: yes. Orientable: yes. Example datafile declaration:

```
quantity cell_grav energy modulus 980*8.5 method string_gravity
```

hooke_energy. Description: One would often like to require edges to have fixed length. The total length of some set of edges may be constrained by defining a fixed quantity. This is used to fix the total length of an evolving knot, for example. But to have one constraint for each edge would be impractical, since projecting to n constraints requires inverting an n x n matrix. Instead there is a Hooke's Law energy available to encourage edges to have equal length. Its form per edge is

$$E = |L - L_0|^p,$$

where L is the edge length, L_0 is the equilibrium length, embodied as an adjustable parameter `hooke_length`, and the power p is an adjustable parameter `hooke_power`. The default power is $p = 2$, and the default equilibrium length is the average edge length in the initial datafile. You will want to adjust this, especially if you have a total length constaint. A high modulus will decrease the hooke component of the total energy, since the restoring force is linear in displacement and the energy is quadratic (when $p = 2$). As an extra added bonus, a `hooke_power` of 0 will give $E = -\log|L - L_0|$. See hooke2_energy for individual edge equilibrium lengths. Element: edge. Parameters: none. Models: linear. Ambient dimension: any. Hessian: yes. Example datafile declaration:

```
parameter hooke_length 0.3   // will apply to all edges
parameter hooke_power  2     // the default
quantity slinky energy method hooke_energy global
```

hooke2_energy. Description: Same as hooke_energy, but each edge has an equilibrium length extra attribute `hooke_size` (which the user need not declare). If the user does not set `hooke_size` by the time the method is first called, the value will default to the current length. `Hooke_size` is not automatically adjusted by refining. Element: edge. Parameters: none. Models: linear. Ambient dimension: any. Hessian: yes. Example datafile declaration:

```
parameter hooke_power  2     // the default
quantity slinky energy method hooke2_energy global
...
read
r;r;set edge hooke_size length
```

hooke3_energy. Description: Same as hooke2_energy, but uses an elastic model instead of a spring. The energy is energy = 0.5*(length-hooke_size)2̂/hooke_size. The exponent can be altered from 2 by setting the parameter `hooke3_power`. Element: edge. Parameters: none. Models: linear. Ambient dimension: any. Hessian: yes. Example datafile declaration:

```
parameter hooke3_power  2     // the default
quantity slinky energy method hooke3_energy global
...
read
r;r;set edge hooke_size length
```

local_hooke_energy. Description: Energy of edges as springs with equilibrium length being average of lengths of neighbor edges. Actually, the energy is calculated per vertex,

$$E = (\frac{L_1 - L_2}{L_1 + L_2})^2$$

where L_1 and L_2 are the lengths of the edges adjacent to the vertex. Meant for loops of string. (by John Sullivan) Element: vertex. Parameters: none. Models: linear. Ambient dimension: any. Hessian: no. Example datafile declaration:

```
quantity slinky energy method local_hooke_energy global
```

dihedral_hooke. Description: Energy of an edge is edge length times square of angle between normals of adjacent facets. Actually, e = (1 - cos(angle))*length. Element: edge. Parameters: none. Models: linear. Ambient dimension: any. Hessian: yes . Example datafile declaration:

```
quantity bender energy method dihedral_hooke global
```

sqcurve_string. Description: Integral of squared curvature in string model. Assumes two edges per vertex, so don't use with triple points. Value zero at endpoint of curve. Calue is calculated as if the exterior angle at the vertex is evenly spread over the adjacent half-edges. More precisely, if s1 and s2 are the adjacent edge lengths and t is the exterior angle, value = 4*(1 - cos(t))/(s1+s2). Other powers of the curvature can be specified by using the parameter `parameter_1` in the instance definition. Element: vertex. Parameters: `parameter_1`. Models: linear. Ambient dimension: any. Hessian: yes. Example datafile declaration:

```
quantity sq energy method sqcurve_string global
parameter_1 3
```

metric_edge_length. Description: In the string model with a Riemannian metric, this is the length of an edge. Element: edge. Parameters: none. Models: linear,quadratic,simplex. Ambient dimension: any. Hessian: yes. Example datafile declaration:

```
string
space_dimension 2
metric
1+x^2 y
y   1+y^2
quantity mel energy method metric_edge_length global
```

klein_length. Description: Edge length in Klein hyperbolic plane model. Does not depend on `klein_metric` being declared. Vertices should be inside unit sphere. Element: edge. Parameters: none. Models: linear. Ambient dimension: 2. Hessian: no. Example datafile declaration:

```
quantity kleinlen energy method klein_length global
```

2-dimensional

facet_tension, facet_area. Description: Area of facet. Does not multiply by facet density; density_facet_area does that. Quadratic model uses Gaussian cubature of order `integral_order_2D`. Beware that this is an approximation to the area, and if the facets in the quadratic or Lagrange model get too distorted, it can be a bad approximation. Furthermore, facets can distort themselves in seeking the lowest numerical area. By default, changing the model to quadratic or Lagrange will set an appropriate `integral_order_2D`. Element: facet. Parameters: none. Models: linear, quadratic, Lagrange, simplex. Ambient dimension: any. Hessian: yes. Example datafile declaration:

```
quantity farea energy method facet_area global
```

density_facet_area. Description: Area of facet, multiplied by its density. Otherwise same as facet_area. Element: facet. Parameters: none. Models: linear, quadratic, Lagrange, simplex. Ambient dimension: any. Hessian: yes. Example datafile declaration:

```
quantity farea energy method density_facet_area global
```

facet_volume. Description: Integral $\int z\,dx\,dy$ over an oriented facet. Valid in the torus domain. Not valid for other symmetry groups. Element: facet. Parameters: none. Models: linear, quadratic, Lagrange. Ambient dimension: 3. Hessian: yes. Orientable: yes. Example datafile declaration:

```
quantity vol fixed = 1.3 method facet_volume
```

facet_scalar_integral. Description: Integral of a scalar function over facet area. Uses Gaussian cubature of order `integral_order_2D`. Element: facet. Parameters: `scalar_integrand`. Models: linear, quadratic, Lagrange. Ambient dimension: any. Hessian: yes. Example datafile declaration:

```
quantity fint energy method facet_scalar_integral global
scalar_integrand:  x^2+y^2
```

facet_vector_integral. Description: Integral of a vectorfield inner product with the surface normal over a facet. The normal is the right-hand rule normal of the facet as defined in the datafile. Uses Gaussian cubature of order `integral_order_2D`. Element: facet. Parameters: `vector_integrand`. Models: linear, quadratic, Lagrange, simplex. Ambient dimension: any. Hessian: yes. Orientable: yes. Example datafile declaration, for volume equivalent:

```
quantity fvint energy method facet_vector_integrand
vector_integrand:
q1:  0
q2:  0
q3:  z
```

facet_2form_integral. Description: Integral of a 2-form over a facet. Meant for ambient dimensions higher than 3. Uses Gaussian cubature of order `integral_order_2D`. Element: facet. Parameters: `form_integrand` (components in lexicographic order). Models: linear, Lagrange, simplex. Ambient dimension: any. Hessian: yes. Orientable: yes. Example datafile declaration in 4D:

```
quantity formex energy method facet_2form_integral
form_integrand:
q1:  x2     // 12 component
q1:  0      // 13 component
q1:  x4     // 14 component
q1:  0      // 23 component
q1:  0      // 24 component
q1:  x3*x2  // 34 component
```

facet_general_integral. Description: Integral of a scalar function of position and normal vector over a facet. Uses Gaussian cubature of order `integral_order_2D`. The components of the normal vector are represented by continuing the coordinate indices. That is, in 3D the position coordinates are x1,x2,x3 and the normal components are x4,x5,x6. For proper behavior, the integrand should be homogeneous of degree 1 in the normal components. Element: facet. Parameters: `scalar_integrand`. Models: linear, quadratic, Lagrange. Ambient dimension: any. Hessian: yes. Example: The facet area could be calculated with this quantity:

```
quantity arclength energy method edge_general_integral
scalar_integrand:  sqrt(x4^2 + x5^2 + x6^2)
```

facet_torus_volume. Description: For 3D soapfilm model, calculates body volume integral for a facet, with corrections for edge wraps. Element: facet. Parameters: none. Models: linear,quadratic,lagrange. Ambient dimension: 3. Hessian: yes. Orientable: yes. Example datafile declaration:

```
quantity body_vol energy method facet_torus_volume
```

gravity_method, full_gravity_method. Description: Gravitational energy, integral $\int \rho z^2/2 dx\, dy$ over a facet, where ρ is difference in adjacent body densities. Note: this method uses the gravitational constant as the modulus if invoked as full_gravity_method. Just gravity_method does not automatically use the gravitational constant. Element: facet. Parameters: none. Models: linear, quadratic, Lagrange. Ambient dimension: 3. Hessian: yes. Orientable: yes. Example datafile declaration:

```
quantity grav energy modulus 980*8.5 method gravity_method
```

facet_area_u, density_facet_area_u. Description: Area of facet. In quadratic model, it is an upper bound of area, by the Schwarz Inequality. For the paranoid. Same as facet_area in linear model. Sets `integral_order_2D` to 6, since it doesn't work well with less. Using the density_facet_area_u name automatically incorporates the facet tension, but facet_area_u doesn't. Element: facet. Parameters: none. Models: linear, quadratic. Ambient dimension: any. Hessian: yes. Example datafile declaration:

```
quantity area_u energy method facet_area_u global
```

gap_energy. Description: Implementation of gap energy, which is designed to keep edges from short-cutting curved constraint surfaces. This method serves the same purpose as declaring a constraint `convex`. Automatically incorporates the gap constant set in the datafile or by the `k` command. Element: edge. Parameters: none. Models: linear. Ambient dimension: any. Hessian: no. Example datafile declaration:

```
quantity gappy energy method gap_energy global
```

metric_facet_area. Description: For a Riemannian metric, this is the area of a facet. Element: edge. Parameters: none. Models: linear,quadratic,simplex. Ambient dimension: any. Hessian: yes. Example datafile declaration:

```
metric
1+x^2 0 z
0 1+y^2 0
z 0 1+z^2
quantity mfa energy method metric_facet_area global
```

klein_area. Description: Facet area in Klein hyperbolic 3D space model. Does not depend on `klein_metric` being declared in the datafile. Vertices should be inside the unit sphere. Element: facet. Parameters: none. Models: linear. Ambient dimension: 3. Hessian: no. Example datafile declaration:

```
quantity kleinarea energy method klein_area global
```

dirichlet_area. Description: Same as the facet_tension method, but the Hessian is modified to be guaranteed positive definite, after the scheme of Polthier and Pinkall [PP]. The energy is taken to be the Dirichlet integral of the perturbation from the current surface, which is exactly quadratic and positive definite. Hence the hessian command always works, but final convergence may be slow (no faster than regular iteration) since it is only an approximate Hessian. Also see the `dirichlet` command. Element: facet. Parameters: none. Models: linear. Ambient dimension: any. Hessian: yes. Example datafile declaration:

```
quantity dirarea energy method dirichlet_area global
```

sobolev_area. Description: Same as the facet_tension method, but the Hessian is modified to be guaranteed positive definite, after the scheme of Renka and Neuberger. [RN]. Hence the `hessian` command always works, but final convergence may be slow (no faster than regular iteration) since it is only an approximate Hessian. Also see the `sobolev` command. Element: facet. Parameters: none. Models: linear. Ambient dimension: any. Hessian: yes. Example datafile declaration:

```
quantity sobarea energy method sobolev_area global
```

pos_area_hess. Description: Same as the facet_area method, but the Hessian can be adjusted various ways by setting the variables `fgagfa_coeff`, `gfa_2_coeff`, `gfagfa_coeff`, and `gfaafg_coeff`. This will make sense if you look at the Dirichlet section of the Technical Reference chapter of the printed manual. The default values of the coefficients are -1, 1, -1, and 0 respectively. Element: facet. Parameters: none. Models: linear. Ambient dimension: any. Hessian: yes. Example datafile declaration:

```
quantity parea energy method pos_area_hess global
```

spherical_area. Description: The spherical area of a triangle projected out to a unit sphere. The user must have the vertices on the unit sphere. First meant for minimal surfaces in 4D that are to be mapped to surfaces of constant mean curvature in 3D. Element: facet. Parameters: none. Models: linear. Ambient dimension: any. Hessian: no. Example datafile declaration:

```
constraint 1  formula:  x^2 + y^2 + z^2 = 1
quantity spharea energy method spherical_area global
```

Surface curvature functions

mean_curvature_integral. Description: Integral of signed scalar mean curvature of a 2D surface. The computation is exact, in the sense that for a polyhedral surface the mean curvature is concentrated on edges and singular there, but the total mean curvature for an edge is the edge length times its dihedral angle. Element: edge. Parameters: none. Models: linear. Ambient dimension: any. Hessian: no. Example datafile declaration:

```
quantity mci energy method mean_curvature_integral
```

sq_mean_curvature. Description: Integral of squared mean curvature of a surface. The integral of squared mean curvature in the soapfilm model is calculated as follows: Each vertex v has a star of facets around it of area A_v. The force due to surface tension on the vertex is

$$F_v = -\frac{\partial A_v}{\partial v}.$$

Since each facet has 3 vertices, the area associated with v is $A_v/3$. Hence the average mean curvature at v is

$$h_v = \frac{1}{2}\frac{F_v}{A_v/3},$$

and this vertex's contribution to the total integral is

$$E_v = h_v^2 A_v/3 = \frac{1}{4}\frac{F_v^2}{A_v/3}.$$

Philosophical note: The squared mean curvature on a triangulated surface is technically infinite, so some kind of approximation scheme is needed. The alternative to locating curvature at vertices is to locate it on the edges, where it really is, and average it over the neighboring facets. But this has the problem that a least area triangulated surface would have nonzero squared curvature, whereas in the vertex formulation it would have zero squared curvature.

Practical note: The above definition of squared mean curvature seems in practice to be subject to instablities. One is that sharp corners grow sharper rather than smoothing out. Another is that some facets want to get very large at the expense of their neighbors. Hence a couple of alternate definitions have been added.

Curvature at boundary: If the edge of the surface is a free boundary on a constraint, then the above calculation gives the proper curvature under the assumption the surface is continued by reflection across the constraint. This permits symmetric surfaces to be represented by one fundamental region. If the edge of the surface is a fixed edge or on a 1-dimensional boundary, then there is no way to calculate the curvature on a boundary vertex from knowledge of neighboring facets. For example, the rings of facets around the bases of a catenoid and a spherical cap may be identical. Therefore curvature is calculated only at interior vertices, and when the surface integral is done, area along the boundary is assigned to the nearest interior vertex. However, including `ignore_fixed` or `ignore_constraints` in the method declaration will force the calculation of energy even at fixed points or ignoring constraints respectively.

If the parameter `h_zero` is defined, then the value per vertex is the same as for the following method, **eff_area_sq_mean_curvature**.

Element: vertex. Parameters: `ignore_constraints`, `ignore_fixed`. Models: linear. Ambient dimension: any. Hessian: no. Example datafile declaration:

```
quantity sqc energy method sq_mean_curvature global
```

eff_area_sq_mean_curvature. Description: Integral of squared mean curvature of a surface, with a slightly different definition from sq_mean_curvature or normal_sq_mean_curvature. The area around a vertex is taken to be the magnitude of the gradient of the volume. This is less than the true area, so makes a larger curvature. This also eliminates the spike instability, since a spike has more area gradient but the same volume gradient. Letting N_v be the volume gradient at vertex v,

$$h_v = \frac{1}{2}\frac{F_v}{||N_v||/3},$$

and

$$E_v = h_v^2 A_v/3 = \frac{3}{4}\frac{F_v^2}{||N_v||^2}A_v.$$

The facets of the surface must be consistently oriented for this to work, since the evolver needs an 'inside' and 'outside' of the surface to calculate the volume gradient. There are still possible instabilities where some facets grow at the expense of others.

If the parameter `h_zero` is defined, then the value per vertex is

$$E_v = (h_v - h_0)^2 A_v/3 = 3\left(\frac{F_v \cdot N_v}{2||N_v||^2} - h_0\right)^2 A_v.$$

This does not reduce to the non-`h_zero` formula when `h_zero` has the value zero, but users should feel lucky to have any `h_zero` version at all.

If the vertex is on one or several constraints, the F_v and N_v are projected to the constraints, essentially making the constraints act as mirror symmetry planes.

WARNING: For some extreme shapes, Evolver may have problems detecting consistent local surface orientation. The `assume_oriented` toggle lets Evolver assume that the facets have been defined with consistent local orientation.

Element: vertex. Parameters: none. Models: linear. Ambient dimension: any. Hessian: no. Example datafile declaration:

```
quantity effsq energy method eff_area_sq_mean_curvature global
```

normal_sq_mean_curvature. Description: Integral of squared mean curvature of a surface, with a slightly different definition from sq_mean_curvature or eff_area_sq_mean_curvature. To alleviate the instability of eff_area_sq_mean_curvature, normal_sq_mean_curvature considers the area around the vertex to be the component of the volume gradient parallel to the mean curvature vector, rather than the magnitude of the volume gradient. Thus

$$h_v = \frac{1}{2}\frac{F_v^2}{N_v \cdot F_v/3}$$

$$E_v = \frac{3}{4}\left[\frac{F_v \cdot F_v}{N_v \cdot F_v}\right]^2 A_v.$$

This is still not perfect, but is a lot better.

If the parameter `h_zero` is defined, then the value per vertex is

$$E_v = (h_v - h_0)^2 A_v/3 = \left[\frac{3}{2}\frac{F_v^2}{N_v \cdot F_v} - h_0\right]^2 \frac{A_v}{3}$$

If the vertex is on one or several constraints, the F_v and N_v are projected to the constraints, essentially making the constraints act as mirror symmetry planes.

WARNING: For some extreme shapes, Evolver may have problems detecting consistent local surface orientation. The `assume_oriented` toggle lets Evolver assume that the facets have been defined with consistent local orientation.

Element: vertex. Parameters: none. Models: linear. Ambient dimension: any. Hessian: no. Example datafile declaration:

```
quantity nsq energy method normal_sq_mean_curvature global
```

star_sq_mean_curvature. Description: Integral of squared mean curvature over a surface. This is a different implementation of sq_mean_curvature which is more suitable for parallel calculation and has a Hessian. But it assumes a closed surface, i.e. each vertex it is applied to should have a complete star of facets around it. This method does not use the `h_zero` parameter.

Element: vertex. Parameters: none. Models: linear. Ambient dimension: any. Hessian: yes. Example datafile declaration:

```
quantity starsq energy method star_sq_mean_curvature global
```

star_eff_area_sq_mean_curvature. Description: Integral of squared mean curvature over a surface. This is a different implementation of eff_area_sq_mean_curvature which is more suitable for parallel calculation and has a Hessian. But assumes a closed surface, i.e. each vertex it is applied to should have a complete star of facets around it. This method does not use the `h_zero` parameter.

Element: vertex. Parameters: none. Models: linear. Ambient dimension: any. Hessian: yes. Example datafile declaration:

```
quantity seffsq energy method star_eff_area_sq_mean_curvature global
```

star_normal_sq_mean_curvature. Description: Integral of squared mean curvature over a surface. This is a different implementation of normal_sq_mean_curvature which is more suitable for parallel calculation and has a Hessian. But it assumes a closed surface, i.e. each vertex it is applied to should have a complete star of facets around it. This method can use `h_zero`.

Element: vertex. Parameters: none. Models: linear. Ambient dimension: any. Hessian: yes. Example datafile declaration:

```
quantity stnsq energy method star_normal_sq_mean_curvature global
```

gauss_curvature_integral. Description: This computes the total Gaussian curvature of a surface with boundary. The Gaussian curvature of a polyhedral surface may be defined at an interior vertex as the angle deficit of the adjacent angles. But as is well-known, total Gaussian curvature can be computed simply in terms of the boundary vertices, which is what is done here. The total Gaussian curvature is implemented as the total geodesic curvature around the boundary of the surface. The contribution of a boundary vertex is

$$E = (\sum_i \theta_i) - \pi.$$

The total over all boundary vertices is exactly equal to the total angle deficit of all interior vertices plus $2\pi\chi$, where χ is the Euler characteristic of the surface. Element: facet. Parameters: none. Models: linear. Ambient dimension: any. Hessian: no. Example datafile declaration:

```
quantity gint energy method gauss_curvature_integral global
```

sq_gauss_curvature. Description: Computes the integral of the squared Gaussian curvature. At each vertex, the Gaussian curvature is calculated as the angle defect divided by one third of the total area of the adjacent facets. This is then squared and weighted with one third of the area of the adjacent facets. This method works only on closed surfaces with no singularities due to the way it calculates the angle defect. Element: vertex. Parameters: none. Models: linear. Ambient dimension: any. Hessian: no. Example datafile declaration:

```
quantity sqg energy method sq_gauss_curvature global
```

Simplex model methods

simplex_vector_integral. Description: Integral of a vectorfield over a (n-1)-dimensional simplicial facet in n-space. Vectorfield is dotted with normal of facet; actually the side vectors of the simplex and the integrand vector are formed into a determinant. Element: facet. Parameters: `vector_integrand`. Models: simplex. Ambient dimension: any. Hessian: no. Orientable: yes. Example datafile declaration, for 4-volume under a 3D surface in 4D:

```
quantity xvint energy method simplex_vector_integral
vector_integrand:
q1:  0
q2:  0
q3:  0
q4:  x4
```

simplex_k_vector_integral. Description: Integral of a simple (n-k)-vector over an oriented k-dimensional simplicial facet in n-space. The vector integrand lists the components of each of the k vectors sequentially. Evaluation is done by forming a determinant whose first k rows are k vectors spanning the facet, and last (n-k) rows are vectors of the integrand. Element: facet. Parameters: `k_vector_order`, `vector_integrand`. Models: simplex. Ambient dimension: any. Hessian: yes. Orientable: yes. Example datafile declaration, for 3D surface in 5D:

```
quantity kvec energy method simplex_k_vector_integral
k_vector_order 3
vector_integrand:
q1:  0   // first vector
q2:  0
q3:  0
q4:  0
q5:  x4
q6:  0   // second vector
q7:  0
q8:  0
q9:  x3
q10:  0
```

edge_k_vector_integral. Description: Integral of a simple (n-k)-vector over an oriented k-dimensional simplicial edge in n-space. The vector integrand lists the components of each of the k vectors sequentially. Evaluation is done by forming a determinant whose first k rows are k vectors spanning the edge, and last (n-k) rows are vectors of the integrand. Element: edge. Parameters: `k_vector_order`, `vector_integrand`. Models: linear, quadratic, simplex. Ambient dimension: any. Hessian: yes. Orientable: yes. Example datafile declaration, for 3D edges of a 4D surface in 5D:

```
quantity kvec energy method edge_k_vector_integral
k_vector_order 3
vector_integrand:
q1:  0   // first vector
q2:  0
q3:  0
q4:  0
q5:  x4
```

```
q6:  0   // second vector
q7:  0
q8:  0
q9:  x3
q10:  0
```

knot_energy. Description: An "electrostatic" energy in which vertices are endowed with equal charges. Inverse power law of potential is adjustable via the global parameter `knot_power`, default value 2 (which is not electrostatic, but the knot theorists like it). If the extra attribute `node_charge` is defined for vertices, then that value is used for the vertex charge. Use of this energy is not restricted to knots; it has been used to embed complicated network graphs in space. Element: vertex. Parameters: none. Models: linear. Ambient dimension: any. Hessian: yes. Example datafile declaration:

```
parameter knot_power  2     // the default
quantity knotten energy method knot_energy global
```

uniform_knot_energy, edge_knot_energy. Description: A knot energy where vertex charge is proportional to neighboring edge length. This simulates an electrostatic charge uniformly distributed along a wire. Inverse power law of potential is adjustable via the global parameter `knot_power` (default 2). Element: vertex. Parameters: none. Models: linear. Ambient dimension: any. Hessian: no. Example datafile declaration:

```
parameter knot_power  2     // the default
quantity knotten energy method uniform_knot_energy global
```

uniform_knot_energy_normalizer. Description: Supposed to approximate the part of uniform_knot_energy that is singular in the continuous limit. Element: vertex. Parameters: Models: linear. Ambient dimension: any. Hessian: no. Example datafile declaration:

```
parameter knot_power  2     // the default
quantity knottenorm energy method uniform_knot_energy global
method uniform_knot_energy_normalizer global
```

uniform_knot_normalizer1. Description: Calculates internal knot energy to normalize singular divergence of integral of uniform_knot_energy. Actually a synonym for uniform_knot_energy_normalizer. No gradient. Element: vertex. Parameters: none. Models: linear. Ambient dimension: 3. Hessian: no. Example datafile declaration:

```
parameter knot_power  2     // the default
quantity knottenorm energy method uniform_knot_energy global
method uniform_knot_energy_normalizer1 global
```

uniform_knot_normalizer2. Description: Calculates internal knot energy to normalize singular divergence of integral of uniform_knot_energy a different way from uniform_knot_energy_normalizer. Element: edge. Parameters: none. Models: linear. Ambient dimension: 3. Hessian: no. Example datafile declaration:

```
parameter knot_power  2     // the default
quantity knottenorm energy method uniform_knot_energy global
method uniform_knot_energy_normalizer2 global
```

edge_edge_knot_energy, edge_knot_energy. Description: Between pairs of edges, energy is inverse square power of distance between midpoints of edges. Can also be called just edge_knot_energy. See also edge_knot_energy_normalizer. (by John Sullivan) Element: edge. Parameters: none. Models: linear. Ambient dimension: any. Hessian: no. Example datafile declaration:

```
quantity knotten energy method edge_edge_knot_energy global
```

edge_knot_energy_normalizer. Description: Calculates internal knot energy to normalize singular divergence of integral of edge_edge_knot_energy. Element: edge. Parameters: none. Models: linear. Ambient dimension: 3. Hessian: no. Example datafile declaration:

```
quantity knotten energy method edge_edge_knot_energy global
method edge_knot_energy_normalizer global
```

simon_knot_energy_normalizer. Description: Another normalization of edge_knot_energy, which I don't feel like deciphering right now. Element: edge. Parameters: none. Models: string linear. Ambient dimension: 3. Hessian: no. Example datafile declaration:

```
quantity kenergy energy method edge_knotenergy global
method simon_knot_energy_normalizer global
```

facet_knot_energy. Description: Charge on vertex is proportional to area of neighboring facets. Meant for knotted surfaces in 4D. Power law of potential is adjustable via the global parameter `knot_power`. See also facet_knot_energy_fix. Element: vertex. Parameters: none. Models: linear. Ambient dimension: any. Hessian: no. Example datafile declaration:

```
parameter knot_power  2     // the default
quantity knotten energy method facet_knot_energy global
```

facet_knot_energy_fix. Description: Provides adjacent vertex correction to facet_knot_energy. Element: vertex. Parameters: none. Models: linear. Ambient dimension: any. Hessian: no. Example datafile declaration:

```
parameter knot_power  2     // the default
quantity knotten energy method facet_knot_energy global
method facet_knot_energy_fix global
```

buck_knot_energy. Description: Energy between pair of edges given by formula suggested by Greg Buck. Power law of potential is adjustable via the global parameter `knot_power`. Element: edge. Parameters: none. Models: linear. Ambient dimension: any. Hessian: no. Example datafile declaration:

```
parameter knot_power  2     // the default
quantity knotten energy method buck_knot_energy global
```

proj_knot_energy. Description: This energy is due to Gregory Buck. It tries to eliminate the need for a normalization term by projecting the energy to the normal to the curve. Its form is

$$E_{e_1 e_2} = \frac{L_1 L_2 \cos^p \theta}{|x_1 - x_2|^p}$$

where x_1, x_2 are the midpoints of the edges and θ is the angle between the normal plane of edge e_1 and the vector $x_1 - x_2$. The default power is 2. Power law of potential is adjustable via the global parameter `knot_power`. Element: edge. Parameters: none. Models: linear. Ambient dimension: any. Hessian: no. Example datafile declaration:

```
parameter knot_power  2     // the default
quantity knotten energy method proj_knot_energy global
```

circle_knot_energy. Description: This energy is due to Peter Doyle, who says it is equivalent in the continuous case to the insulating wire with power 2. Its form is

$$E_{e_1 e_2} = \frac{L_1 L_2 (1-\cos\alpha)^2}{|x_1 - x_2|^2},$$

where x_1, x_2 are the midpoints of the edges and α is the angle between edge 1 and the circle through x_1 tangent to edge 2 at x_2. Only power 2 is implemented. Element: edge. Parameters: none. Models: linear. Ambient dimension: any. Hessian: no. Example datafile declaration:

```
quantity knotten energy method circle_knot_energy global
```

sphere_knot_energy. Description: This is the 2D surface version of the circle energy. Its most general form is

$$E_{f_1 f_2} = \frac{A_1 A_2 (1-\cos\alpha)^p}{|x_1 - x_2|^q},$$

where A_1, A_2 are the facet areas, x_1, x_2 are the barycenters of the facets, and α is the angle between f_1 and the sphere through x_1 tangent to f_2 at x_2. The energy is conformally invariant for $p = 1$ and $q = 4$. For $p = 0$ and $q = 1$, one gets electrostatic energy for a uniform charge density. Note that facet self-energies are not included. For electrostatic energy, this is approximately $2.8A^{3/2}$ per facet.

The powers p and q are Evolver variables `surface_knot_power` and `surface_cos_power` respectively. The defaults are $p = 1$ and $q = 4$. Element: facet. Parameters: none. Models: linear. Ambient dimension: any. Hessian: no. Example datafile declaration:

```
parameter surface_knot_power  1     // the default
parameter surface_cos_power  4     // the default
quantity knotten energy method sphere_knot_energy global
```

sin_knot_energy. Description: Another weird way to calculate a nonsingular energy between midpoints of pairs of edges. (by John Sullivan) Element: edge. Parameters: none. Models: linear. Ambient dimension: any. Hessian: no. Example datafile declaration:

```
quantity knotten energy method sin_knot_energy global
```

curvature_binormal. Description: For string model. Evaluates to zero energy, but the force calculated is the mean curvature vector rotated to the binormal direction. Element: vertex. Parameters: none. Models: linear. Ambient dimension: 3. Hessian: no. Example datafile declaration:

```
quantity curbi energy method curvature_binormal global
```

ddd_gamma_sq. Description: Third derivative of curve position as function of arclength, squared. Element: vertex. Parameters: none. Models: string, linear. Ambient dimension: 3. Hessian: no. Example datafile declaration:

```
quantity ddd energy method ddd_gamma_sq global
```

edge_min_knot_energy. Description: Between pairs of edges, energy is inverse square power of distance between closest points of edges.

$$Energy = \frac{1}{d^2}|e_1||e_2|$$

This should be roughly the same as edge_edge_knot_energy, but distances are calculated from edge midpoints there. This is not a smooth function, so we don't try to compute a gradient. DO NOT use as an energy;

use just for info_only quantities. Element: edge. Parameters: none. Models: linear. Ambient dimension: 3. Hessian: no. Example datafile declaration:

```
quantity eminknot info_only method edge_min_knot_energy global
```

true_average_crossings. Description: Calculates the average crossing number of an edge with respect to all other edges, averaged over all projections. Knot stuff. No gradient, so use just in info_only quantities. Element: edge. Parameters: none. Models: linear. Ambient dimension: 3. Hessian: no. Example datafile declaration:

```
quantity true_cross info_only method true_average_crossings global
```

true_writhe. Description: For calculating the writhe of a link or knot. No gradient, so use just in info_only quantities. Element: edge. Parameters: none. Models: linear. Ambient dimension: 3. Hessian: no. Example datafile declaration:

```
quantity twrithe info_only method true_average_crossings global
```

twist. Description: Another average crossing number calculation. No gradient, so use just in info_only quantities. Element: edge. Parameters: none. Models: linear. Ambient dimension: 3. Hessian: no. Example datafile declaration:

```
quantity twister info_only method twist global
```

writhe. Description: An average crossing number calculation. This one does have a gradient. Suggested by Hermann Gluck. Programmed by John Sullivan. Between pairs of edges, energy is inverse cube power of distance between midpoints of edges, times triple product of edge vectors and distance vector.

$$E = 1/d^3 * (e1, e2, d)$$

Element: edge. Parameters: none. Models: linear. Ambient dimension: 3. Hessian: no. Example datafile declaration:

```
quantity writhy energy method writhe global
```

curvature_function. Description: Calculates forces as function of mean and Gaussian curvatures at vertices. Function may be changed by user by altering `teix.c`. No energy, just forces. Element: vertex. Parameters: none. Models: linear. Ambient dimension: any. Hessian: no. Example datafile declaration:

```
quantity curfun energy method curvature_function global
```

average_crossings. Description: To calculate the average crossing number in all projections of a knot. (by John Sullivan) Element: edge. Parameters: none. Models: linear. Ambient dimension: 3. Hessian: no. Example datafile declaration:

```
quantity across energy method average_crossings global
```

Weird and miscellaneous.

wulff_energy. Description: Method version of Wulff energy. If Wulff filename is not given in top section of datafile, then the user will be prompted for it. Element: facet. Parameters: none. Models: linear. Ambient dimension: 3. Hessian: no. Example datafile declaration:

```
wulff "crystal.wlf"
```

```
quantity wolf energy method wulff_energy global
```

linear_elastic. Description: To calculate the linear elastic strain energy energy for facets based on the Cauchy-Green strain matrix. Let S be Gram matrix of unstrained facet (dots of sides). Let Q be the inverse of S. Let F be Gram matrix of strained facet. Let $C = (FQ - I)/2$, the Cauchy-Green strain tensor. Let v be Poisson ratio. Then energy density is

$$(1/2/(1+v))(Tr(C^2) + v * (TrC)^2/(1 - (dim - 1) * v)).$$

Each facet has extra attribute poisson_ratio and extra attribute array form_factors[3] = s11,s12,s22, which are the entries in S. That is, s11 = dot(v2-v1,v2-v1), s12 = dot(v2-v1,v3-v1), and s22 = dot(v3-v1,v3-v1). If form_factors is not defined by the user, it will be created by Evolver, and the initial facet shape will be assumed to be unstrained. For another variation, see linear_elastic_B. Element: facet. Parameters: none. Models: linear. Ambient dimension: 3. Hessian: yes. Example datafile declaration:

```
quantity lastic energy method linear_elastic global
```

linear_elastic_B. Description: A variation of the linear_elastic method. To calculate the linear elastic strain energy energy for facets based on the Cauchy-Green strain matrix. Let S be Gram matrix of unstrained facet (dots of sides). Let Q be the inverse of S. Let F be Gram matrix of strained facet. Let $C = (FQ - I)/2$, the Cauchy-Green strain tensor. Let v be Poisson ratio. Then energy density is

$$(1/2/(1+v))(Tr(C^2) + v * (TrC)^2/(1 - (dim - 1) * v)).$$

Here each facet has extra attribute poisson_ratio and each vertex has two extra coordinates for the unstrained position. Hence the entire surface must be set up as five dimensional. Element: facet. Parameters: none. Models: linear. Ambient dimension: 3. Hessian: yes. Example datafile declaration:

```
space_dimension 5
quantity lastic energy method linear_elastic_B global
```

area_square. Description: Energy of a facet is the square of the facet area. Element: facet. Parameters: none. Models: linear. Ambient dimension: 3. Hessian: no. Example datafile declaration:

```
quantity asquare energy method area_square global
```

carter_energy. Description: Craig Carter's energy. Given bodies B_1 and B_2 in R^3, define the energy

$$E = \int_{B_1}\int_{B_2} \frac{1}{|z_1 - z_2|^p} d^3z_2 d^3z_1$$

This reduces to

$$E = \frac{1}{(3-p)(2-p)} \sum_{F_2 \in \partial B_2} \sum_{F_1 \in \partial B_1} N_1 \cdot N_2 \int_{F_2}\int_{F_1} \frac{1}{|z_1 - z_2|^{p-2}} d^2z_1 d^2z_2.$$

And if we crudely approximate with centroids $\bar{z}_1$ and $\bar{z}_2$,

$$E = \frac{1}{(3-p)(2-p)} \sum_{F_2 \in \partial B_2} \sum_{F_1 \in \partial B_1} \frac{A_1 \cdot A_2}{|\bar{z}_1 - \bar{z}_2|^{p-2}},$$

where A_1 and A_2 are unnormalized area vectors for the facets. The power p is set by the variable `carter_power` (default 6). Element: facet. Parameters: none. Models: linear. Ambient dimension: 3. Hessian: no. Example datafile declaration:

```
parameter carter_power  6     // the default
quantity craig energy method carter_energy global
```

charge_gradient. Description: This energy is the gradient squared of the knot_energy method, assuming the points are constrained to the unit sphere. Element: vertex. Parameters: none. Models: linear. Ambient dimension: any. Hessian: no. Example datafile declaration:

```
parameter knot_power  2     // the default
quantity knotten energy method knot_energy global
```

johndust. Description: For all point pairs (meant to be on a sphere),

$$E = (pi - asin(d/2))/d,$$

where d is chord distance. For point packing problems on the sphere. Element: vertex. Parameters: none. Models: linear. Ambient dimension: any. Hessian: no. Example datafile declaration:

```
constraint 1 formula:  x^2+y^2+z^2 = 1
quantity jms energy method johndust global
```

stress_integral. Description: Hmm. Looks like this one calculates integrals of components of a stress tensor. The `scalar_integrand` value is set as an integer standing for which component to do. See the function stress_integral in `method3.c` for details. Does not have a gradient, so should be used for just info_only quantities. Element: facet. Parameters: `scalar_integrand`. Models: linear. Ambient dimension: 3. Hessian: no. Example datafile declaration:

```
quantity stressy info_only method stress_integral global
```

ackerman. Description: Not actually an energy, but a kludge to put inertia on vertices. Uses extra velocity coordinates to represent vertex in phase space. Invocation actually transfers computed forces from space coordinates to velocity coordinates, so forces become acceleration instead of velocity. Element: vertex. Parameters: none. Models: linear. Ambient dimension: any. Hessian: no. Example datafile declaration:

```
quantity jeremy energy method ackerman global
```

Chapter 9. Helpful hints and notes.

When writing a new datafile, use the `d` command to check that it is being read as you think it should be.

Run at low resolution before refining. Having many triangles not only takes a long time to calculate, but motion can propagate only one triangle per iteration. Also run the linear model first, and use the quadratic model for final polishing (if at all).

Conjugate gradient should be used only when regular motion has settled down. Conjugate gradient assumes a quadratic energy function, and may get confused when it's not. Conjugate gradient may need to be toggled off and on to make it forget its history.

Judging convergence is tough. If iterations run at a more or less constant scale factor and energy isn't changing much, and running in conjugate gradient mode for a long time doesn't change much, then you're probably in good shape. The `hessian` command may help.

Watch out for edges shrinking to zero length and triangles to zero area; they can stall progress. If the optimal scale factor heads toward 0, try equianglation (`u`), vertex averaging (`V`), small triangle weeding (`w`), or the tiny edge removing (`t`). Always precede and follow `w` with `t`.

Facets entirely on constraints tend not to move, and moving free boundaries tend to overrun such facets, leading to problems. Use constraint integrals rather than facets on constraints where possible.

If your edges on curved constraints try to shortcut the curve, try declaring the constraint `CONVEX` to add a penalty for shortcutting. Another approach is to add an energy integrand term to the constraint that is zero on the constraint and positive off it. Integrands are evaluted at several points in the interior of an edge, not the endpoints, so this gives an energy penalty for going away from the edge. Hopefully, it will not change the nature of your solution.

Use constraints for two-dimensional free boundaries, and use parametric boundaries for wire frames.

Use the `d` command to save your surface at critical points.

Run checks (`C`) after modifying a surface just to make sure everything worked right.

Choose your units so that extreme sizes do not occur. Constraint tolerances and the like may have to be adjusted. Also, some of the graphics displays (like geomview) work in 32-bit reals, and have trouble with objects of size 1e-5.

Chapter 10. Bugs.

There are no known outright bugs at present. but they undoubtably exist. When you run across one, I would like to hear about it. Bug reports should be submitted by email to `brakke@susqu.edu`. Please include the Evolver version number, a description of the problem, the initial data file, and the sequence of commands necessary to reproduce the problem.

There are a few shortcomings, however:

Not all features are implemented in all models.

Vertex and edge popping is not elegant or complete for vertices on boundaries or constraints.

Zero length edges and zero area triangles stall things.

Surfaces can intersect each other without knowing it.

Convergence to a minimum energy can be difficult to judge.

Chapter 11. Version history.

Version 1.0 August 4, 1989

First public version.

Version 1.01 August 22, 1989

Various bug fixes.

Constraint integral specification changed to have just components, not density.

Facet-edge specification made obsolete (but still legal).

Vertex motion adjustment for quadratic model.

Zoom factor changed from 2 to 1.2.

Initial datafile reading revamped to use lex and yacc. Has simple macros. Comments must be delimited. Expressions now in algebraic form, not RPN.

Version 1.10 October 18, 1989

More bug fixes.

Graphics driver for HP98731 contributed by Eric Haines.

Datafile reading:

Equations permitted as constraints.

Backslash line splicing.

Constant folding in expressions.

Version 1.11 May 18, 1990

More bug fixes.

x,y,z accepted as synonyms for x1,x2,x3

Version 1.12 May 25, 1990

Datafile coordinates may be given as expressions.

Constraints revamped. Number of constraints per element raised to sizeof(int). One-sided constraints may be applied selectively. `GLOBAL` constraints automatically apply to all vertices (they count in number limit).

Version 1.2 June 17, 1990

Comments removed from input stream before lex analyzer. So comments in macros safe now.

Scale factor upper bound adjustable when toggling with `m` command.

Adjustable constants implemented. Syntax in data file: `PARAMETER` name = constant-expression These are treated as constants in expressions, but can be changed with the `A` command. Useful for dynamically changing contact angles and other stuff in constraint expressions.

Symmetric content evaluation added for linear model. Volumes evaluated as surface integral of $(x\vec{i}+y\vec{j}+z\vec{k})/3$ rather than of $z\vec{k}$. Use keyword `SYMMETRIC CONTENT` in datafile. Permits more accurate evaluation of lunes.

Return value of sprintf not used, as this varies among systems.

Arbitrary surface energy integrands added. Vector field integrated over designated facets. Syntax

surface energy *n*
e1: *expression*
e2: *expression*
e3: *expression*

Designate facet by following facet definition with "energy n", in same manner as giving constraints. Linear model only. This useful for changing direction of gravity, by putting in Divergence Theorem equivlent surface integrand for gravitational potential energy and putting adjustable constant in for direction.

The 'q' or 'x' exit command gives you a chance to load another datafile, continue with the current configuration, or exit.

You can change the initial upper limit on the 'scale' factor by putting a line in the data file:

SCALE LIMIT *value*

value must be a number (no expression).

The 'V' command (for Vertex averaging) will replace each unconstrained vertex by the average of its neighboring vertices (connected by edges). Good for uncramping skinny triangles sometimes.

TENSION allowed as synonym for DENSITY in setting facet surface tension.

Binary save/reload disabled, since it is out of date and a binary file turns out twice as large as the ascii dump file.

Version 1.21 June 30, 1990

Shared memory interface added for MinneView (which is a public domain 3D graphics viewer for Irises written by the Geometry Project at the Minnesota Supercomputer Institute.)

Histogram added for ridge notcher (option 'n').

Ported to NeXT (no screen graphics, but PostScript output files can be displayed).

Automatic energy recalculation added after all options that change energy.

Version 1.3 July 30, 1990

Reptition counts before letters in graphics commands.

Clockwise (c) and counterclockwise (C) rotations added to graphics.

Printout during quadratic search for optimum scale suppressed.

'i' command added for information. 'v' reports just volumes.

Extrapolation command 'e' fixed.

Constant expressions permitted wherever real value needed.

'F' command to log commands to file.

-ffilename command line option to read commands from file. Take commands from stdin afterwards.

Dump file real values printed to 15 decimal places so accuracy not lost.

Adjustable accuracy on edge integrals via `INTEGRAL_ORDER` keyword in datafile for setting order of Gaussian quadrature.

Datafile now allows specification of constraint tolerance with `CONSTRAINT_TOLERANCE`.

All two-word keywords made into single words by connecting the words with ' _'.

Parsing of datafile continues after errors.

'b' command permits editing body volumes.

'v' command prints 'none' for prescribed volume of those bodies with no prescribed volume.

Manual in TEXformat with PostScript figures.

`CONVEX` gap energies and forces fixed up. $k = 1$ best approximation to area.

Vertex popping fixed to handle disjoint components of tangent cones. Also not to screw up its data structures after a modification.

Version 1.31

Added long-wavelength random jiggle (command `jj`).

Fixed simultaneous Minneview to handle torus display options.

Simultaneous MinneView can be stopped and restarted.

Fixed bug in Gaussian quadrature of content integrals.

Fixed equiangulation to work on arbitrarily small triangles.

Histograms adjusted to current length and area order of magnitude.

Zero area triangle test area cutoff adjusted to scale of surface.

Vertex averaging improved to preserve volumes.

Small edge removal bug fixed. In collapsing a facet, will preferentially keep an edge on a constraint.

`VOLCONST` adjustment to body volumes added to datafile.

"Quantities" added. A quantity is a sum of vector integrals over facets and edges. Can be simply tallied for informational purposes, or can be used as mathematical constraints with fixed values. Good for center of mass, moment of inertia, magnetic flux, etc.

Conjugate gradient energy minimization added. Use the `U` command to toggle between gradient descent and this.

Version 1.4 posted August 20, 1990

Version 1.41 September 22, 1990

Bug fixes on vertex popping and element list management.

Long jiggle command jj lets user put in own numbers, use random numbers, or use previous numbers.

Version 1.42

Bug fixes:

segment violation on error message at end of data file.

STRING model not graphing third dimension

On SOAPFILM model, edges without facets are displayed. Useful for showing wire boundaries beyond surface. Such edges will generate warnings when datafile is read, however.

Version 1.5 May 15, 1991

dimensional surfaces can live in N-dimensional space. See `SPACE_DIMENSION`.

Simplex model added to represent k-dimensional surfaces in N-dimensional space.

Embryonic query language added. Does `list` and `show`. Also set and unset attributes.

Background metric on space added for string model only.

Commands can be read from a file with `read "`*filename*`"`

Version 1.6 June 20, 1991

Riemannian metric extended to all dimensional surfaces.

Refine, delete added to query language. List query has same format output as datafile dump.

Quotient spaces added.

Piping output added to commands and queries.

Version 1.63 July 21, 1991

SET FACET COLOR query added. Colors include CLEAR. Also SET FACET TRANSPARENCY for Iris and like.

Squared mean curvature added as possible energy.

Queries can refer to elements by original number of parent in datafile.

Adjustable constants can take values from a file according to ID number of element applied to.

Version 1.64 August 2, 1991

Revised notch command (n) to subdivide adjacent facets instead of edge itself. Supposed to be followed with equiangulation.

Put in K command to subdivide longest edges of skinny triangles (as judged by their smallest angle).

Added VALENCE attribute for edges for queries. Is the number of facets on an edge.

Version 1.65 August 20, 1991

Added minimization by using Newton's method on Hessian matrix of energy. Only for no-constraint area minimization with no other energies. Command "`hessian`".

NeXT version given graphical interface.

User-defined functions of coordinates added. See `userfunc.c`.

Version 1.76 March 20, 1992

Autopopping and autochopping in string model for automatic evolution.

Phase-dependent grain boundary energies.

Approximate polyhedral curvature.

Stability test for approximate curvature.

Squared Gaussian curvature as part of energy only, not force.

"system" command to execute shell commands.

"check_increase" to guard against blowup during interation.

"effective_area" to count only resistance to motion normal to surface.

Runge-Kutta iteration.

Version 1.80 July 25, 1992

Command and query language much extended.

`geomview` interface added.

Fixed area added as a constraint.

Multiple viewing transforms can be specified in the datafile so one fundamental region of a symmetric surface can be displayed as the whole surface.

Commands can be included at the end of the datafile, introduced by the keyword READ.

Version 1.83 September 9, 1992

Some alternate definitions of squared curvature added. Invoked by "`effective_area ON | OFF`" or "`normal_curvature ON | OFF`".

Version 1.84 September 10, 1992

Shaded colors added to xgraph and cheygraph.

Version 1.85 September 29, 1992

Restriction of motion to surface normal added. Toggle "tt normal_motion".

Squared mean curvature, Gaussian curvature, and squared Gaussian curvature extended to surfaces with boundary.

Datafile element attribute "`bare`" added for vertices and edges in soapfilm model so they won't generate erroneous warnings.

Force calculation added for squared Gaussian curvature, so it can be used in the energy.

All prompts that require real value responses now accept arbitrary expressions.

Version 1.86 October 19, 1992

User-defined mobility added, both scalar and tensor forms.

Default squared curvature works for 2-surfaces in R^n.

Version 1.87 October 27, 1992

"`close_show`" command added to close show window (the native graphics window, not geomview).

Graphics command checks string for illegal characters before doing any transformations.

Dihedral angles now work for 2-surfaces in any dimension.

Permanent variable assignments may be made with "`::=`" instead of "`:=`". Such assignments will not be forgotten when a new surface is begun.

Conditional expressions of C form added: expr ? expr : expr. Useful for patching constraints together.

Version 1.88 December 16, 1992

"SET BACKGROUND color" command added for native graphics.

View transformation generators and expressions added.

Exact bounding box calculated for PostScript files.

Version 1.88a January 6, 1993

Default constraint_tolerance lowered from 1e-5 to 1e-12.

Fixed bug in volume constraint calculation introduced in 1.88.

Version 1.89 February 18, 1993

Postscript draws fixed and boundary edges in interior of surface. All internal graphics should be consistent in the special edges they draw (bare edges, triple edges, etc.).

Viewing matrix can be read from datafile and will be dumped. Keyword VIEW_MATRIX

Mod operator '%' added, and functions floor(), ceil().

'rebody' command added to recalculate connected bodies after neck pinching and any other body disruption.

If squared mean curvature part of energy, then squared mean curvature at a vertex is available as a query attribute as "sqcurve".

These quantities can now be used in command expressions: vertex_count, edge_count, facet_count, body_count, total_energy, total_area, total_length, scale.

Dump file records defined procedures in 'read' section at end.

Knot energies added, both conducting and insulating wire.

'dissolve' command added to erase elements and leave gaps in surface, unlike delete command, which closes gaps. Can only dissolve elements not neede by higher dimensional elements.

Command repeat numbers have been restricted to just three types of commands: 1. single letter commands that don't have optional arguments (l,t,j,m,n,w have optional arguments) 2. command list in braces 3. user-defined procedure names This is to prevent disasters like list vertex 1293 which before would produce 1293 full lists of all vertices.

"dump" without argument will dump to default file name, which is datafile name with .dmp extension.

SIGTERM is caught and causes dump to default dump file and exit. Useful for interrupting scripts running in the background with kill -TERM. Likewise for SIGHUP.

Version 1.90 April 2, 1993

Conjugate gradient ON/OFF state saved in dump file. Note that conjugate gradient history vector is not saved.

Notching and "dihedral" attribute apply to vertices in the string model.

FOREACH iterator added. Syntax: FOREACH element [name] [WHERE expr] DO command

LOAD command added. Syntax: LOAD "filename". Useful for starting new surfaces, especially in scripts.

PRINTF command added for formatted printing. Syntax: PRINTF "format string",expr,expr,... expr is floating point, so use %f or %g formats.

String variables have been added. Can be used where quoted strings needed. Can be assigned to. SPRINTF is version of PRINTF giving string output.

A view transformation matrix in the datafile may be preceded by "color n" to give that transform a color (overrides any facet color).

In queries, element attribute "oid" added, which returns a signed version of id.

Many knot energies added. Also a "hooke energy" that keeps edges near a uniform length.

PostScript output optionally includes color.

Version 1.91 May 31, 1993

Two sides of facets can have different colors. '`COLOR`' applies to both sides, '`FRONTCOLOR`' and '`BACKCOLOR`' to different sides.

Attributes of individual named elements can be set inside loops, i.e. foreach facet ff do set ff color red

Every time a command changed a global variable, the surface was being recalculated. This slowed down scripts immensely. So now the only variables that cause recalculation are 1) adjustable parameters defined in the datafile 2) quantity moduli and parameters

History commands now echoed.

Surface area can be minimized by minimizing Dirichlet integral, according to scheme of Polthier and Pinkall. Command '`dirichlet`'.

To reduce need for explicit line-splicing on long commands, the parser now keeps track of depth of brace and parenthesis nesting, and will call for more input if a line ends inside nest. So if you want to type a multiline command, start with " and end with " many lines later. Also does auto line-splicing if certain tokens are last token in line (such as '+').

'`facet_knot_energy_fix`' method added.

Command assignment fixed to assign only one command. so "`ggg := g; g`" will be the same as "`{ggg := g} ; g`" and not "`ggg := {g;g}` ".

Queries can run through edges and facets adjacent to a vertex, and facets of a body, as in "`list vertices vv where max(vv.facet,color==red) > 0`"

Improved NeXT terminal interface. `-u` option for no graphics, `-t` for terminal and graphics.

view_4d command to toggle sending full 4D coordinates to geomview. Default is OFF.

Version 1.92 July 31, 1993

SGI parallelism enabled for named quantity calculations.

method-instance scheme introduced.

Torus periods can be specified with expressions using adjustable parameters.

Verbs (list,refine,delete,dissolve) may apply to single elements: foreach edge ee do refine ee

FIX and UNFIX may be used as verbs:

`fix vertices where on_constraint 1`

Toggle command names may now be used as boolean values in expressions in commands. Also new boolean read-only variables: torus, torus_filled, symmetry_group, simplex_representation. New numeric read-only variables: space_dimension, surface_dimension, integration_order.

Mac version repeated commands interruptable with control-'.'.

Macintosh and Dos versions pipe to a file instead of a command:

```
Enter command:  list vertices | "filename"
```

For a torus domain, the torus periods may be specified using expressions with parameters, so the fundamental cell may be changed interactively. Do a `recalc` after changing such a parameter to update the torus periods.

`gv_binary` toggle for binary/ascii data to geomview. Default ON for binary, which is faster. Ascii mode useful for debugging.

Version 1.93 December 13, 1993

"history" command added to print command history. Single-letter commands now included in history for convenience. But history does not record responses to prompts commands may issue.

command repeat counts can now be expressions

More internal variables for counters on command events: equi_count, delete_count, notch_count, dissolve_count, pop_count, where_count.

Quantities for scalar and vector integrands over edges and facets added.

Quantity names may now be used as element attributes. Value of total quantity must be referred to as "total quantityname".

DOS version has improved graphics. Recognizes higher resolution and more colors.

Can apply named quantities to elements with SET command.

Undocumented user_attr.

Version 1.94 January 24, 1994

New named quantity methods: vertex_scalar_integrand, facet_2form_integral.

Hessians for named quantity methods: edge_length, facet_area, vertex_scalar_integral, edge_scalar_integral, edge_vector_integral, facet_scalar_integral, facet_vector_integral, facet_2form_integral, gravity_method.

Added edge wrap as readable attribute.

Added coordinate attributes for edges and facets. Interpreted as edge vector components and facet normal components.

Commands are added to history list after being successfully parsed, rather than after successful execution.

Unfound files are treated as errors rather than prompting for new name, except for datafiles.

New arithmetic operators: mod (synonym for %), imod, idiv. New arithmetic function: atan2(y,x).

Show conditions for edges and facets are saved in read section of datafile.

Total energy is in a comment at the top of a dump file.

PostScript output in case of string model in 3D has option for doing bordered crossings.

'`w`' is synonym for coordinate `x4`.

Version 1.95 June 24, 1994

Named quantity methods for squared curvature: sq_mean_curvature, eff_area_sq_mean_curvature, normal_sq_mean_curvature. All work with h_zero.

New quantities edge_general_integral, facet_volume, facet_torus_volume, facet_general, stress_integral, edge_area, edge_torus_area. Also quadratic model versions of basic quantities. Also hessian.

Quadratic midpt for edges included in dump. Optional in datafile.

'midv' attribute for edges in quadratic model.

'iteration_counter' variable for printing current repetition number.

New math functions: tanh, asinh, acosh, atanh.

Extra attributes, indexable. Can have non-dumping extra attributes.

'quietgo' toggle to suppress output of just the 'g' command.

'ribiere' toggle for conjugate gradient. Does not initiate CG. Also extra projections to constraints in conjugate gradient or post-project mode, with iteration_counter set to -1 if nonconvergence in 10 projections.

'assume_oriented' toggle for square mean curvature.

Label option added to PostScript output.

Short-circuit evaluation of AND and OR.

Delete facet more aggressive; tries to eliminate all edges of facet in increasing length order until it succeeds.

Version 1.96 September 22, 1994

'G' command takes numerical argument instead of repetition count.

`hessian_menu` command has experimental stuff for hessian, eigenvalues, and eigenvectors. `Saddle` will seek along lowest eigenvector without the need to go into the menu.

sprintf, printf now accept string args, but they are pushed as 8 bytes, so %s in format string should be followed by half.

'Extra' attributes now inherited for same type element.

Added 'jiggle' toggle, and jiggle_temperature internal variable.

Added total_time internal variable. Settable also.

Compound quantities allowing quantity energy to be a function of method instances.

Indexing on element generators.

Permitted element attributes in datafile expressions for quantities, etc, altho only works in named quantities.

hessian_normal flag for hessian motion constrained to surface normal at non-singular points.

Made ribiere default mode for conjugate gradient.

geomview "string" command added to let scripts send commands to geomview.

The print command also accepts strings. Example: print datafilename

The name of the current datafile can be referred to in commands as 'datafilename' wherever a string can be used.

Version 1.97 December 16, 1994

Named quantities can refer to qqq.value, qqq.target, qqq.pressure, and qqq.modulus.

Alternative Hessian factoring (via ysmp toggle), with Bunch-Kauffman option.

lanczos and eigenprobe commands added.

Dump saves states of toggles.

Version 1.98 March 15, 1995

Lanczos(t,n) and ritz(t,n) commands added. Linear metric for eigenvalues.

-q option to turn everything into quantities.

geomview picking.

break and continue commands.

P, M commands take arguments

random_seed variable added for user control of random numbers.

Added 12-pt degree 6 and 28-pt degree 11 integration rules for facets. Made default facet cubature 12-pt 5th degree. Much more stable for area. Separate integral_order_1D and integral_order_2D variables.

Version 1.99 July 19, 1995

"target", "volconst" attributes for bodies and quantities.

Higher degree cubature formulas.

`linear_metric` for hessian.

`edgeswap` command.

Torus translations added automatically to view transform generators.

Element structures only allocate needed storage.

`convert_to_quantities` command.

Constraint limit raised to 127.

Toggles print previous values.

Hessian operations save direction of motion. Eigenvalue saved in `last_eigenvalue`, stepsize in `last_hessian_scale`. `Eigenprobe`(λ, n) finds eigenvector. Hessian menu option G minimizes squared gradient instead of energy for `saddle` and `hessian_seek`. Can pick Ritz vector in hessian menu. `Saddle` and `hessian_seek` commands take stepsize limit arguments.

`move` command.

`-DSDIM`n compiler option to hardwire dimension for optimization.

Hessian calculation parallelized for SGI_MULTI mode.

`Bottominfo` command.

Redefinition of single-letter commands.

`geompipe` toggle.

`P` command takes argument to short-circuit menu.

Version 2.00 April 30, 1996

HTML version of documentation, also used by help command.

Lagrange model.

Quad precision when compiled with -DLONGDOUBLE.

Windows NT version.

'X' command prints extra attribute dictionary.

optimizing_variable introduced.

postscript command with toggles instead of interactive (as in P 3)

"return" command for ending current command.

Version 2.01 August 15, 1996

Mac 68K and Power PC versions.

"node_charge" vertex attribute for knot_energy. Useful for spreading graphs out.

new_vertex, etc.

V modified; vertex_average; works better on constrained vertices and quadratic edge midpoints.

Simplex equiangulation in 3D.

no_refine attribute for edges and vertices.

>> redirection

dynamic link libraries for functions.

DOS, Windows version leaves alphanumeric escape sequences alone.

Can print transform_expr, transform_count.

Version 2.10 July 10, 1998

Window NT/95/98 version using OpenGL has much better graphics. Can rotate, translate, or zoom with left mouse button, pick elements with right button, even do cross-eyed stereo. Type 'h' in the graphics window for a command summary. Also made catenoid icon for the program.

C-style assignment operators +=, -=, *=, /= work where reasonable.

Parameterized boundaries can have energy and content integrals, in the same way level set constraints have had.

Command output redirection to a file using ¿¿ for append, ¿¿¿ for overwrite.

A variable can be toggled between optimizing or non-optimizing at run time with "unfix varname" and "fix varname" respectively.

"keep_macros" in datafile header keeps macros active after datafile.

Command line option -i will keep element ids the same as in the datafile, rather than renumbering consecutively as is the default.

If you want to reorder elements in the internal lists (the way elements are listed by, say, 'list vertices', you can define the extra attributes vertex_order_key, edge_order_key, facet_order_key, body_order_key, facetedge_order_key, give them all appropriate values, and then give the command reorder_storage. See reorder.cmd in the distribution fe directory.

'renumber_all' renumbers elements in internal list order.

Read-only internal variable 'random' for random numbers uniformly distributed between 0 and 1.

Added warnings about using keywords as identifiers in the datafile. The datafile will still run, but your commands will misinterpret those identifiers.

All keywords are in the on-line help. Do 'help "keyword"' if you want to check on a potential keyword. 'help' also recognizes identifiers defined by the user in the current surface. For testing in scripts, there is a function is_defined(stringexpr) that returns 1 if a name is already in use, 0 if not.

In the datafile, making an edge FIXED no longer fixes its endpoints. This is compatible with how fixed facets and fixing edges at run time have always worked.

For named methods that logically depend on the orientation of the element (i.e. facet_vector_integral, etc.), the relative orientation of the element when the method is applied is recorded. Default is positive in the datafile, unless a '-' is added after the name of a method or quantity applied to an individual element.

Version 2.11 March 1, 1999

Added "verbose" flag for action messages from pop edge, pop vertex, delete, notch, refine, dissolve, edgeswap, unstar.

IGNORE_FIXED and IGNORE_CONSTRAINT flags for sq_mean_curvature methods.

Old fixed_area finally gotten rid of.

Assignment statements permissible at start of expressions; useful for common subexpressions in constraint and quantity integrands.

hessian_slant_cutoff variable for controlling hessian on constraints.

-e option to echo input; useful for piped input.

Automatic conversion to named quantities when needed; suppressed by -a- option.

"scale" attribute for optimizing parameters for impedance matching of scale factors.

Version 2.14 August 18, 1999

hessian_normal is now defaults to ON.

Put in automatic convert_to_quantities; still a place or two where can't convert on the fly. Command line option -a- disables.

Added "verbose" toggle command for action messages from pop edge, pop vertex, delete, notch, refine, dissolve, and edgeswap.

The "dissolve" command will now dissolve facets on bodies in the soapfilm model, and edges on facets in the string model.

Edges have frontbody and backbody attributes in the string model.

There is a chdir command to change the working directory.

Element extra attributes can be declared with code to evaluate their values.

Chapter 12. Bibliography.

[A] F. Almgren, "Minimal surface forms," *The Mathematical Intelligencer*, vol.4, no. 4 (1982), 164–172.

[AV] V. I. Arnol'd, *Catastrophe Theory*, 3rd ed., Springer-Verlag, 1992.

[AT] F. Almgren and J. Taylor, "The geometry of soap films and soap bubbles," *Scientific American*, July 1976, 82–93.

[B1] K. Brakke, *The motion of a surface by its mean curvature*, Princeton University Press, Princeton, NJ (1977).

[B2] K. Brakke, "The surface evolver," *Experimental Mathematics*, vol. 1, no. 2 (1992), 141–165.

[B3] K. A. Brakke, "Minimal surfaces, corners, and wires," *Journal of Geometric Analysis* **2** (1992) 11-36.

[B4] K. A. Brakke, "The opaque cube problem video," *Computing Optimal Geometries*, J. E. Taylor, ed., American Mathematical Society, Providence RI, 1991.

[B5] K. A. Brakke, "Grain growth with the Surface Evolver," *Video Proceedings of the Workshop on Computational Crystal Growing*, J. E. Taylor, ed., American Mathematical Society, Providence RI, 1992.

[B6] K. A. Brakke, "The opaque cube problem,", *Am. Math. Monthly* **99** (Nov. 1992), 866-871.

[B6] K. A. Brakke and J. M. Sullivan, "Using Symmetry Features of the Surface Evolver to Study Foams", in Mathematics and Visualization, ed. K. Polthier and H. Hege, Springer Verlag, Berlin, (1997).

[C] M. Callahan, P. Concus and R. Finn, "Energy minimizing capillary surfaces for exotic containers," *Computing Optimal Geometries*, American Mathematical Society, Providence RI, 1991.

[DL] D. Dobkin and M. Laszlo, "Primitives for the manipulation of three-dimensional subdivisions," technical report CS-TR-089-87, Department of Computer Science, Princeton University, Princeton, New Jersey. (April 1987).

[FT] H. J. Frost and C. V. Thompson, "Computer Simulation of Grain Growth," Current Opinion in Solid State and Materials Science 3 (1996), 361.

[FHW] M. Freedman, X. He, and Z. Wang, "On the energy of knots and unknots," *Annals of Mathematics* **139** no. 1 (1994) 1–50.

[FS] G. Francis, J. M. Sullivan, R. B. Kusner, K. A. Brakke, C. Hartman, and G. Chappell, "The Minimax Sphere Eversion", in Mathematics and Visualization, ed. K. Polthier and H. Hege, Springer Verlag, Berlin, (1997).

[HK] J. Hale and H. Kocak, *Dynamics and Bifurcations*, Springer-Verlag, 1991.

[HKS] L. Hsu, R. Kusner, and J. Sullivan, "Minimizing the squared mean curvature integral for surfaces in space forms," *Experimental Mathematics* **1** (1991) 191–208.

[KR1] A. Kraynik and D. Reinelt, "The linear elastic behaviour of a bidisperse Weiare-Phelan soap foam", submitted to *Chem. Eng. Comm.*

[KR2] A. Kraynik and D. Reinelt, "Simple shearing flow of a dry Kelvin soap foam", submitted to *J. Fluid Mech.*

[KS1] R. Kusner and J. M. Sullivan, "Comparing the Weaire-Phelan Equal-Volume Foam to Kelvin's Foam", *Forma* (The Society for Science on Form, Japan) ed. R. Takaki and D. Weaire, KTK Scientific Publishers. As book by Taylor and Francis Ltd (1996).

[KS2] R. Kusner and J. M. Sullivan, "Mobius Energies for Knots and Links, Surfaces and Submanifolds" in Geometric Topology, Proceedings of the Georgia International Topology Conference, August 1993, ed. W. Kazez, AMS/IP Studies in Advanced Mathematics, vol. 2, part 1 (1997) 570–604.

[MB] X. Michalet and D. Bensimon, "Observation of stable shapes and conformal diffusion in genus 2 vesicles", *Science* **269** (4 Aug 1995) 666–668.

[MH] H. Mittelmann, "Symmetric capillary surfaces in a cube", *Math. Comp. Simulation* **35** (1993) 139–152.

[MF1] F. Morgan, "Harnack-type mass bounds and Bernstein theorems for area-minimizing flat chains modulo ν," *Comm. Part. Diff. Eq.* **11** (1986) 1257–1283.

[MT] F. Morgan and J. E. Taylor, "Destabilization of the tetrahedral point junction by positive triple junction line energy," *Scripta Metall. Mater.* **25** (1991) 1907-1910.

[PWB] R. Phelan, D. Weaire, and K. Brakke, "Computation of equilibrium foam structures using the Surface Evolver," *Experimental Mathematics* **4** (1995) 181-192.

[P] W. Press, B. Flannery, S. Teukolsky, and W. Vetterling, *Numerical Recipes in C*, Cambridge University Press, New York, 1988.

[PP] K. Polthier and U. Pinkall,"Computing discrete minimal surfaces and their conjugates," SFB 288 Preprint 49, Berlin, 1993.

[RN] R. J. Renka and J. W. Neuberger, "Minimal surfaces and Sobolev gradients", *SIAM J. Sci. Comput.* **16** (1995) 1412–1427.

[RSB] L. M. Racz, J. Szekely, and K. A. Brakke, "A General Statement of the Problem and Description of a Proposed Method of Calculation for Some Meniscus Problems in Materials Processing", ISIJ International, Vol. 33, No. 2, (February 1993) 328–335.

[S] R. Sibson, "Locally equiangular triangulations", *Comput. J.* **21** (1978) 243–245.

[SG] G. Strang, *Linear Algebra and its Applications*, Academic Press (1988).

[SJ] J. M. Sullivan, "Sphere Packings Give an Explicit Bound for the Besicovitch Covering Theorem", *J. Geometric Analysis* **4** (1993) 219–231.

[T1] J. E. Taylor, "The structure of singularities in soap-bubble-like and soap-film-like minimal surfaces," *Ann. Math.* **103** (1976), 489-539.

[T2] J. E. Taylor, "Constructing crystalline minimal surfaces," *Seminar on Minimal Submanifolds*, E. Bombieri, ed., *Annals of Math. Studies* **105** (1983), 271–288.

[T3] J. E. Taylor, "Constructions and conjectures in crystalline nondifferentiable geometry," *Proceedings of the Conference in Differential Geometry*, Rio de Janiero, 1988, Pittman Publishing Ltd.

[Te] J. Tegart, "Three-dimensional fluid interfaces in cylindrical containers," AIAA paper AIAA-91-2174, 27th Joint Propulsion Conference, Sacramento, CA, June 1991.

[TW] W. Thompson (Lord Kelvin), "On the division of space with minimum possible error," *Acta Math.* **11**(1887), 121–134.

[U] A. Underwood, "Constructing barriers to minimal surfaces from polyhedral data," Ph.D. thesis, Princeton, 1993.

[WM] D. Weaire and S. McMurry, "Some Fundamentals of Grain Growth," in Solid State Physics: Advances in Research and Applications (eds. H. Ehrenreich and F. Spaepen), Vol. 50, Academic Press, Boston (1997).

[WP] D. Weaire and R. Phelan, “A counter-example to Kelvin’s conjecture on minimal surfaces”, *Phil. Mag. Letters* **69** (1994), 107-110.

Chapter 13. Index.

Surface Evolver keywords are in typewriter font.