# Notes on the Freg (eff–reg) Software

Bruce Mills

December 12, 2000

## 1 Introduction

This document is intended to give an overview of and some context for of the software that I wrote for the *beautification of solid models* project, during the latter part of the year 2000. The main reason for writing it is to inform whomever comes after me on the project. So, if you are that person, then I advise you to read this document through before looking at the code, or attempting to run any code.

## 2 The External Picture

The condition of the code as I leave it is that there is a single top directory, and a number of sub–directories. None of the code explicitly refers to the absolute location of the main directory. So, it should be possible to shift this code as a whole without affecting its operation. In the top directory there is the one *makefile* which should be used to recompile after modifications.

The main executable is *freg* (Pronounced eff–reg) in the top directory. The core function of freg is processing data derived from boundary models. The input is a text stream containing the location of the vertices, and the edge boundary information, for the boundary model. The output is a text stream describing the approximate symmetry of the information, and new locations for the vertices that conform to the equivalent exact symmetry.

The most straight forward example of the use of freg is illustrated in the following command.

```
freg mod/cube4 > cube4.output
```

This command causes the file `cube4` in the `mod` directory to be processed, feeding the output into the file `cube4.output` in the top directory. In the directory `mod` are stored all the model files.

At the time of writing there are a number of reasonably complete files, in particular, `cube1` is an exact cube, `cube4` is an approximate cube as is `cube5`, and `icosa1` is an approximate icosahedron. Of these only `cube5` includes all the aspects expected, it is approximate and it includes the vertex, edge, and face type information. The syntax for these bits of information is explained in the following. I advise that you have a look at `cube5` in conjunction with reading this document, and also have a look at some of the other files. Other than these files, the directory `mod` contains a number of files that just contain vertex geometry information, but can still be fed into freg and the symmetries as a point–set obtained.

**Model File Format**

The `#` character introduces a comment. Any `#` character, and the characters that follow it, until the end–of–line the line in which the `#` occurs, are treated as *comment*, and otherwise ignored.

Except for the comments, which can otherwise be ignored, the syntax of the files is neither column nor line based. That is, padding the data files with spaces and blank lines does not affect the meaning.

The full data file contains the following ...

1. An introductory comment describing the solid in natural language.
2. A list of vertex locations.
3. A comment listing the edge loops.
4. A list of edges.
5. A list of the face types.

Of the three lists, it is valid to have just the first, or the first and the second, or all three. If the third list, the face type information, is missing then it is supposed that all faces are the same type, if the edge information is not there, then the file represents a collection of points in space. This is equivalent to putting an empty `[ ]` collection of edges.

The list of vertex locations starts with a `{` and terminates with a `}`, each individual point location is a list of Cartesian co–ordinates enclosed in parentheses. As in the example below:

```
{
( 12.3 4.5 6)
(1.0 -5.6 3.2)
}
```

Each number is a standard C floating point number. There must be some white–space (space, tab, end–of–line) in between the numbers, but otherwise white space is irrelevant. This is an ordered list, the first element is implicitly taken as the location of vertex 0, the second for vertex 1 and so on.

The structural information about the edges records for each edge the vertices that form its end points, and the two faces that of which it is a common boundary element. This list is delimited by `[` and `]`.

```
[
(1:2 3:4)
(4:1 2:6)
]
```

The edges are numbered implicitly from 0 as for the vertices. In each entry, such as $(1 : 23 : 4)$ indicates the two vertices 1 and 2 that are the end points of the edge, and the two faces 3 and 4 that the edge lies between. The set of indices for faces is declared implicitly by the occurrence of an index for a face in the edge list.

The structure information does not have to be present, if it is not included, then it is simply ignored, which is equivalent to the assumption that we are working with a set of isolated points.

The face type information just lists an integer for each type. These types might represent planes, cylinders, cones, spheres and tori, but also might include further implicit information such as big–spheres and little–spheres, or whatever other information can be used to distinguish faces that can't possibly match. This is assumed preprocessing for freg. From the point of view of freg, a face type is just an integer that labels the face.

```
>
1∼2
2∼3
>
```

The above example shows a listing that indicates that face–1 has type 2, and face–2 has type 3. So `a∼b` can be read as "face `a` has type `b`". The list is enclosed within braces `{ }`, and is an unordered set of type specifications. Any face not explicitly mentioned is by default type 0. Other than being silly, it is not actually a syntax error to duplicate a face number, in this case the later entry over–rides the earlier entry.

The comments, of course, do not affect the interpretation of the file by freg, but I would advise that intuitive information is included since it is of use to a human reading and checking the file. The freg program prints out some of this sort of information, such as face loops, inside comments. However, I also advise that the uncommented information, the information read by freg, should be structured in a non redundant form, so that input and output is cleaner.

### The output format of freg

The current structure of the output is to a certain extent historical, and line–of–thought. Basically as freg is processing the information it prints out various bits and pieces that seemed like a good idea at the time. However, the entire file uses the freg syntax, and can be read back into freg, or extracted by text editing to produce various required files. If you run the command `freg mod/cube4 > cube4.output`, mentioned earlier, then you will get a file of the format as described below.

**1.** The name of the file being processed.

**2.** The geometry and structure of the model.

This includes the production of the face boundary information mentioned above.

**3.** The analysis of the initial model.

Later on freg factors the model, and analyses that as well, but the first piece of output is the results from handling the model exactly as it is given. As it stands, sometimes to get a full analysis of a model the program has to be run to get the tolerancing information, and the factoring, and then this information is extracted and fed back into the program. The programming to make this fully automatic should be fairly straight forward, but at this stage has not been done.

The information that is output at this stage is as follows:

**a.** The tolerance partition

I explain this in the paper that you should have a copy of, and also at other points in the documentation. For the moment this is just the partition in which each vertex is classified in a class of its own.

**b.** The factored set of vertices

Again, in this case it is just the original collection.

**c.** The simplex.

This is the tetrahedron that is used to locate, or *register* the model in space. It is used to reduce the complexity of the algorithm, since points can be located approximately by reference to this tetrahedron, and so not all combinations need to be tried. Again, see the paper for more details on the meaning of this item.

This information is mainly internal data on the execution of the program. In principle it can be used to check the validity of the output, by see how well the tetrahedron covers the object. Typically if the tetrahedron is about one eighth of the volume of the convex hull of the solid, then the processing is ok. But the connection between the tetrahedron and the validity of the processing is tangled. I print this out as a quick check, just in case the tetrahedron is obviously no good. In practice it has shown to be fine.

However, there is also a second reading of the tetrahedron. Later on in discussing the post processing of the freg output it is required to find a non degenerate tetrahedron in the data set. Obviously the above tetrahedron is such, and would be a good one to use for this purpose.

**d.** The node group.

**e.** The edge group.

**f.** The face group.

**g.** The full group.

The actual investigation of the regularity of the model is performed in terms of the ways in which a symmetry can rearrange (permute) the components of the model without breaking it. So for example, all four of the corners of a square can be cycled, but no symmetry of the square will just swap two adjacent corners. The acceptable symmetries also induce permutations of the edges and faces. The first three groups here are the lists of permutations of the vertices, edges and faces which have been found to be compatible with the entire model structure. The full group is the permutations of the vertices that can be obtained if the rest of the structure is ignored.

As an example of this, the file `mod/teseract` is a digital version of the green angular doughnut shape that you should have found (make out of cardboard) on my old desk. The vertices of this have cubic symmetry, but the model as a whole has only the symmetries of a square. However, for a cube, an icosahedron, and any other of the arcamedic solids, the full group and the node group are identical.

An other example is `mod/cube5` in which it is the face type information that causes the lack of symmetry. One of the faces is said to be type 1, which means that it cannot be mapped to any of the other faces, which by default are assumed to be type 0.

The permutations are listed one after the other. The standard format is to place them one on a line with a comment at the end of the line indicating the index of that entry in the list. Each permutation is delimited by `|` at each end. Technically, the permutations are actually of type *injection* which is an freg type that stores a one–to–one integer function. Permutations are a special case, so rather than explain the full syntax of injections only the output generated by a permutation will be covered here.

Each line is in product–of–cycles notation.

For example, the permutation that maps each element in the left hand side to the right hand side, in the following array

$1 \rightarrow 3$
$2 \rightarrow 5$
$3 \rightarrow 4$
$4 \rightarrow 1$
$5 \rightarrow 2$

can be seen to map $1 \rightarrow 3 \rightarrow 4 \rightarrow 1$ which is notated `(1 3 4)`, and also to map $2 \rightarrow 5 \rightarrow 2$ which is notated `(2 5)`. So the line for this would be as follows:

```
(1 3 4)(2 5)
```

With the implicit mapping of the last element back to the first.

**h.** The rectified object.

Ok, not the entire thing. Rather, just the locations of the vertices in the rectified model.

**i.** The rectified edge length matrix

This is not in any freg data format, and is output as a comment. I have found this to be useful at times to check the program and also to learn more about the symmetries being discovered. It is a symmetric array of floating point numbers, the $(i, j)$th element is the distance between vertex $i$ and vertex $j$. The $(0, 1)$th element is normalised to 2, by scaling the whole structure. This tends to make the numbers come out to be something interesting like $1 + \sqrt{5}/2$ instead of all sorts of obscure numbers.

**4.** The partitioned regularities.

The basic information available for each acceptable partition is the same as that for the initial object.

## 3 Data file syntax

There are a variety of compound data types defined by freg, and each of them has a textual form so that they can be read from and written to human readable text streams.

| | |
|---|---|
| `point` | a single geometric point |
| `pointset` | a collection of points |
| `dualedge` | an edge in a dual graph |
| `dualgraph` | two graphs with the same edge set |
| `realisation` | scheme and geometry of a B–rep |
| `injection` | a one to one integer function |
| `pseudosym` | a symmetry of a dual–graph |
| `heap` | a priority queue implementation |
| `partition` | a partition of a point–set |

The syntax is fairly simple and I will not give a formal definition.

A point is just a list of floating point numbers, the separator is a space, and the list is enclosed in parentheses.

```
(2.3 4.5 -1)
```

A point–set is a list of points, there is no need for a separator, but white space padding (as always) is allowed between symbols (i.e. not actually within a number of course). The list is enclosed within a pair of braces.

```
{(2.3 4.5 -1) (1.2 3 4)}
```

A dual–edge is a pair of pairs. The first pair is a pair of vertices, the second is a pair of faces, but in either case all the base elements are just integers. The two elements of a single pair are joined by a colon, the separator between the pairs is white–space. The whole is enclosed in parentheses.

```
(1:2 3:4)
```

A dual–graph is a list of dual–edges. There is no need for a separator, and the whole is enclosed between a pair of braces.

```
{(1:2 3:4) (2:5 12:56)}
```

A realisation is a triplet. The first element is a point–set, the second element is a dual–graph, and the third element is a list of face–type indicators. There is no enclosure. It is valid to include only the point–set, or the point–set and dual–graph, or all three items. Pragmatically the termination of a realisation is the end–of–file marker. Usually I just keep one realisation in a file. However, any syntactically unacceptable character, if included in the file after a valid realisation, will cause the parser to stop, and return the realisation. For examples of realisations see the files indicated earlier.

An injection is a one to one integer function. An injection can be viewed as a set of chains, where a chain might be open or closed. An open chain is a list of integers, separated by commas, and enclosed in parentheses, a closed chain is a list of integers separated by spaces and enclosed in parentheses. The injection is enclosed within a pair of bars `|`. All the integers must be non–negative, and for a well formated injection, no integer should appear more than once. If it does, then the later mapping overwrites the earlier mapping.

```
(1,2)(3,4,5)(7 8 9 10)
```

As far as the pragmatics are concerned a pseudosym is a collection of permutation, which is why I called it a *pseudo*sym. I.e., it might not be a group. However, in principle the output of freg should, and has in practice been shown to, produce a group. This group nature is not enforced, but it could be in order to make the derived symmetry more physically meaningful, should there arise a case in which the output is not actually a group. Technically, the pseudosym is actually a set of injections, represented as a list, enclosed within a pair of braces. For examples of pseudosyms, see the output of freg from any of the valid test files.

The syntax of a heap only exists as an output format, it would be fairly simple to include an input routine, but I have only used this for debugging purposes. For the record, it is a list of pairs, enclosed within a pair of braces. The first element of the pair is a floating point key, and the second is the integer element. The two elements of the pair are separated by a colon, and enclosed in parentheses.

```
{(2.345:1)(3.126:3)}
```

A partition is a set of disjoint sets of integers. It is notated in the standard nested brace notion for sets of sets, except that the separator is a space instead of a comma.

```
{{0 1 2}{1 5}{3 4}}
```

There are also some data types for storing xfig records. However, the three dimensional version of this was not completed and so I find it unlikely that you would be wanting to use this. I mention it here just in case you look in the code and notice the data types and routines. If you are interested, the idea is to extend the file format to recognise text expressions near nodes as labelling those nodes, and then interpret the text as co–ordinates of the node. It is a straight forward exercise to do, but only if you find that you want to quickly sketch polyhedral skeletons. Otherwise just use ACIS models, and write the routines for extracting the information.

Regarding the determination of the constituents of an ACIS model, see the routine `deconstruct` in the file `utility.cc` in the directory `/home/scmbim/sil/bruce`, which also has some other ACIS related code, such as construction of the convex hull of a collection of points. In this directory you will also find the code for the various sample objects that I constructed some time back. Each object has been designed as a parameterised class of objects with some `#define` statements to set the values. Using this a variety of slightly irregular objects can be constructed that would be good for testing the code. On the other hand, Frank also has some material along these lines, and you may want to work with him on this.

## 4 What is still to be done

As it stands, the purpose and current function of the freg software is to solve an essentially combinatorial problem that narrows down the search space when looking for symmetries. Pragmatically, every reasonable symmetry will be included in the output of freg, but some extra symmetries may be suggested that work in a combinatorial fashion, but not geometrically. A small amount of post processing is required to determine which is which.

The basic steps for the in–context use of freg is as follows:

The file `freg.cc` is an example front end to the library functions. It is intended that you can begin by looking at this file to find out how the code is intended to be used. It is then your choice as to whether you use freg direct, modify it, or just totally re–write the front end to your own requirements. This file is less than 200 lines long, and so is not a lot of code in itself.

Begin with an ACIS model, and use `deconstruct`, or similar code that you write yourself, to obtain a list of components of the model. You need a list of faces, edges and vertices, as well as the location of each vertex, and the information about how the edges connect vertices to vertices, and faces to faces. This information could be written to a file in freg format, or directly inserted into freg data structures in code. Either way, the idea is to get the information into an freg realisation type.

Run `find_regularities` on this realisation, and you get in the return variables `node_group`, `edge_group`, `face_group` and `full_group` a suggested symmetry. The full group is just the collection of all permutations of the vertices that are compatible with the tolerance (indicated in the point–set). The node group is the set of permutations of the vertices that are compatible with the tolerance, and with the rest of the structure, including face types. Symmetries such as the continuous rotation of a cylinder, that are not induced from the vertices are not considered here. Some mention has been made of this matter in the paper that I wrote. The edge and face groups are those permutations of the edges and faces that are induced by the node group.

Armed with this information, the idea is to go through each edge permutation and make sure that the edge matching indicated is geometrically valid. In order to do this you have to work out what the isometry is that is implied by the permutation. The way to do this is to find four points in a non–degenerate tetrahedron, and look at the locations of the points to which they are sent by the permutation in question. This mapping of one geometric tetrahedron to another implies a unique isometry. Picking one of the points to to act as the origin, the translation is the distance between the two locations of the origin, and the linear part is obtained by multiplying out the appropriate matrices.

Thus, for each edge permutation, find the isometry, and determine if the isometry does express an acceptable transformation of one edge into another. Do this also for each face permutation. It would probably be best to go through the permutations one by one, and for each permutation check its action on every edge and face, thus only generating each isometry once. This also helps with the logical consistency of the conclusion.

Having checked all of this, and dropped out those permutations that are not compatible with the geometry, you have the full symmetry of the boundary model. Remember that when you drop an edge permutation out of the edge group, you also must drop the corresponding node permutation and face permutation (i.e. the ones with the same index). Similarly, drop edge and node permutations when dropping a face permutation that is not compatible with the geometry.

Having done this you should node have a pseudosym type (the reduced node group) which represents, in a non–redundant manner, the symmetry of the boundary representation, including the geometry. Now feed the vertex set into the `rectify` method of the pseudosym, from this you get a new set of vertices that are compatible with the symmetry. Adjusting the faces to go through these new vertices should produce the desired model. Consistencies such as planar faces on polyhedron should be automatically satisfied, but if the object was too distorted there may be some issue. If the reconstruction is not geometrically possible, then there is most likely something awkward about the model, since the new vertices will not be very far from the old.