Polyhedronism *

file: `poly.scm` – Dialog by TEX←`Scm` v2.2

Keith Wright

November 22, 2023 at 14:36 EST

Contents

* **Draft: do not copy!** Instead get the latest version from `http://www.free-comp-shop.com`

1 Objectives and Mechanics

This is a test and demonstration of a literate programming [3] tool for the Scheme programming language [13, 14, 15], inspired by Knuth's Web [5, 6].

This program will be distributed under the terms of the GNU General Public Licence (GPL) version 2. But it is not distributed yet. If you have it, keep it to yourself. It's not done yet, but you can copy it if, for some reason, you can't get an updated version.

If you are reading a printed copy, or viewing `poly.ps` or `poly.pdf` on a good display screen, skip ahead to the "Introduction"; otherwise:

You should have files called `poly.scm`, `texscm.scm`, and `tspreamble.tex`. To get a printed copy, run Scheme and then:

```
 > (load "texscm.scm")
 > (ts:layout "poly")
```

That should execute `texscm.scm` and `poly.scm` and produce `poly.tex`.

Back at the shell:

```
> latexps poly
```

You should get a Postscript file called "`poly.ps`" Print it, or look at it in your favorite print previewer.

Alternatively, if `guile` is installed, `make` should work.

1.1 Change Log

Ver.	Date	Init.	Description
0.0	2014-04-09	KW	Fiat Lux
0.1	2014-04-17	KW	It has a picture of each of the Platonic polyhedra. Put it on the web.
0.2	2015-10-09	KW	Fetch it from old tar, rename change comments and re-build
0.3	2015-11-20	KW	Use new "show" and "show-TeX" commands. Add "false position" algorithm in *find-zero* procedure. This removes the last two really stinky kludges.
0.4	2015-11-26	KW	Wrote matrix multiplication **mm* and *drawing-proc* and removed a lot of rotation in the definition of the polyhedra.
1.0	2016-02-04	KW	processed with `pprint` version 1.8, with tabbed indentation. Put on web.
1.1	2022-10-28– –2022-10-28	KW	Dug out old program, made minor corrections, processed with the layout program TEX←`Scm` version 2.2. This now a test for changes made to the layout program.

(**define** *program-title* "`Polyhedronism`")
(**set!** *program-title* "`Polyhedronism`")
(**define** (*show x*)*x*)

(**define** (*show-TeX x*)*x*)

2 Introduction

This is a test case for a literate programming tool [3, 4]. I want to write a program that can be read by a person. Unlike Knuth, I don't want to write the program all at once, but develop it interactively, as is traditional with Scheme and other Lisp-like languages.

This is a syntactically correct and executable Scheme program in the file `poly.scm` which has been processed by the program TEX←`Scm` in the file `texscm.scm`. This text you are reading is a comment reformatted to remove the comment markers and run through TEX.

The part of Scheme program that is not comments consists mostly of Scheme definitions like:

```
(define (factorial n)
  (if (zero? n)
      1
      (* n (factorial (- n 1)))))
```

As you see, the definition, which is Ascii text in the file `poly.scm`, is nicely formatted for printing. Now the *show* command will cause the result of evaluating an expression to be also printed.

(*show* (*factorial* 4))

⇒`24`

(*show* (*factorial* 20))

⇒`2432902008176640000`

The result need not be an integer. It could be a TEX command.

```
(show-TeX '("$$\\int_{-\\infty}^\\infty e^{-x^2} dx  =  \\sqrt{\\pi}$$"))
```

⟶

$$\int_{-\infty}^\infty e^{-x^2} dx = \sqrt{\pi}$$

The result of the expression could be a list of strings, which are concatenated to make the command. Some of those strings could be variable arguments.

```
(define (integral a)
  (list "$$\\int_{-\\infty}^\\infty e^{-"   a "^2x^2} dx ="
      ' "\\frac{\\sqrt{\\pi}}{"   a "}$$"))
```

(*show-TeX* (*integral* "`a`"))

⟶

$$\int_{-\infty}^\infty e^{-a^2x^2} dx = \frac{\sqrt{\pi}}{a}$$

(*show-TeX* (*integral* "`(y+1)`"))

⟶

$$\int_{-\infty}^\infty e^{-(y+1)^2x^2} dx = \frac{\sqrt{\pi}}{(y+1)}$$

Now let's try writing an interesting program using these tools.

If European philosophy "consists of a series of footnotes to Plato"[1], then we must admit that the Platonic "dialogs" are not verbatim transcripts. They are ideal dialogs, recalled with mistakes forgotten, and with later interpolations and corrections.

In a similar manner this program, although it takes the form of a Scheme program written by a programmer with results calculated by a computer, is actually more of a fixed point reached by

[1] Whitehead [18, Part II,Ch.I§I,p.63] says it does. (This footnote was added two days after the quotation, which itself was corrected.)

a read-evaluate-print-edit loop, which involves both the programmer and the computer repeatedly making changes.

2.1 Geometric Questions

About thirty years ago I wrote a program for the Macintosh that displayed spinning icosohedrons on the screen. A few years ago I met Brucifer, who lives in a "geodesic dome" that he built. We talked a bit about regular polyhedra. I thought of showing him the program, but of course it had total bit rot. One can't run a Macintosh program from 1984 on a Linux system from 2014.

Later Poly-Alan asked me a question about polyhedra:

> Why does a Dodecahedron have twice as many faces as a cube?

At first, I could not think of any better answer than $12 = 2 \times 6$, but I thought of a way to re-word the question to make the answer more interesting.

Question: Can a dodecahedron be inscribed in a cube in such a way that there is a simple geometric relation, R, between the square faces of the cube and corresponding pairs of pentagonal faces of the 12-hedron such that R is a bijection (one-one onto) between the six squares and the six pairs of 5-gons?

Answer: Yes. I got a 12 sided die from a *Dungeons&Dragons* game, and a transparent box from a toy gyroscope and fit them together. That way I could match the squares and pairs by pushing and turning them until they touched.

The relation R(s,p) is: the edge common to the two pentagons in the pair p lies in the square s. Furthermore, the center of that edge is the center of the square and it is parallel to two of its edges (and of course, perpendicular to the other two). I could compute the length of the pentagon edges in terms of the length of the square edges, which suffices to fix everything.

To prove this is right, the following program explains it more fully, and uses it to draw some pictures of the five Platonic solids.

3 Drawing Polyhedra

This program should do the calculations needed to compute geometric properties of the five Platonic polyhedra, and draw them. The "Platonic" regular polyhedra have regular polygons for faces and each of the vertices (corners) is congruent to each of the others. The Platonic polyhedra are the regular: tetrahedron, hexahedron (cube), octahedron, dodecahedron, and icosahedron. If you have forgotten what they look like there are pictures below.

By the way, the proof that there are exactly five such three dimensional figures is simple. Consider what regular polygon could be used for the faces. If the faces are equilateral triangles then there could be three of them around each vertex. That results in a tetrahedron. There could be four triangles around each vertex, resulting in a octahedron. Five triangles around each vertex results in an icosahedron. Six equilateral triangles around each vertex just flattens out, and results in a triangular tessalation of the plane. Seven triangles around each vertex will not fit at all.

If the faces are square, then three around each vertex make a cube, while four flatten out to make a square tiling. If the faces are pentagons with three around each vertex, then you have a dodecahedron. Four pentagons will not fit at all, and neither will any number of other regular polygons (unless you want to count the flat "bihedrons" which have two polygons of any shape pasted together back-to-back. There are infinitely many of those. Plato did not count them, and neither do I).

If the conditions are relaxed a bit, then other mostly-regular polyhedra are possible, see section 5.

3.1 Simple Cube Drawing

First let's make a vector of (the coordinates of) the vertices of a cube. We only need a three dimensional cube, but we can parameterize by the number of dimensions.

We choose the origin at the center of the cube with rectangular coordinates oriented parallel to the sides and units half the length of an edge. The other reasonable choice, to put the origin at one corner, does not work so well when rotated.

The vertices of an $n+1$ dimensional cube are two copies of the vertices of an n dimensional cube with another coordinate added to each vertex. The value of that extra coordinate is -1 in one copy, and $+1$ in the other copy. There are 2^n points, each with n coordinates.

The base case is a little tricky: a zero dimensional cube has one vertex, which doesn't need coordinates, because it is the only point in all of zero-dimensional space.

```
(define (ncube-verts dim)
  (if (zero? dim)
     '(())
     (let ( (face (ncube-verts (- dim 1)))
           (add-n (lambda (n pts) (map (lambda (pt) (cons n pt))
                                        pts) ) ) )
       (append (add-n -1 face)(add-n 1 face)) )))
```

Now make a vector of coordinates of the vertices of a three dimensional cube.

```
(define cube-verts (apply vector (ncube-verts 3)))
(show cube-verts)
⇒#((-1 -1 -1) (-1 -1 1) (-1 1 -1) (-1 1 1) (1 -1 -1) (1 -1 1) (1 1 -1) (1 1 1))
```

Note that the coordinates of vertex n are the binary representation of n, with -1 in place of 0.

We also need a list of the edges. The edges are pairs of indices into the the vector of vertices. It might be more elegant to construct this list by a recursive algorithm, similar to the one that generates the vertex coordinates. It is quicker and shorter to just type them in.

```
(define cube-edges '((0 1)(2 3)(0 2)(1 3)
                    (4 5)(6 7)(4 6)(5 7)
                    (0 4)(1 5)(2 6)(3 7)))
```

It would be nice to just use the standard Scheme *number→string* procedure to write a number. Unfortunately that can sometimes write in a format that TEX will not read. For example 0.5 might print as "1/2". Cute, but it chokes TEX.

The *texnumber* procedure converts the number x to a string that we hope can be read by TEX. It works well enough for the application at hand, but may fail badly if given a strange number. Since we don't really know much about the *number→string* procedure except that the Scheme *string→number* procedure will convert it back to an equivalent number, it's difficult to really do this right without entirely re-writing the standard procedure.

Any real number will be written as decimal digits, followed by a decimal point and six more digits. Who cares if the drawing is off by a micron? TEX doesn't need complex numbers anyway. This assumes that *number→string* will convert an exact integer to a string of decimal digits, with no funny tricks.

```
(define (texnumber x)
  (define (zeropad n s)
    (string-append (substring "00000000" 0 (- 6 (string-length s))) s))
  (if (< x 0)
    (string-append "-" (texnumber (abs x)))
    (let* ( (whole (inexact→exact (floor x)))
            (frac (inexact→exact (floor (* 1000000 (- x whole))))) )
      (string-append (number→string whole) "." (zeropad 6 (number→string frac))) )))
```

It works like this:

```
(show (texnumber 1/200))
⇒0.005000
```

We need a procedure to draw lines in the document. The *draw-edges* procedure produces a list of character strings which contain the TEX commands to draw straight lines between given points. using the XY-pic [12] TEX macros. Each string in the list is converted to a line which can be inserted into this document by the the "draw-TeX" command.

There is no need to describe all of XY-pic. Lines are drawn by giving the endpoints as two pairs of coordinates separated by a semi-colon then a double asterisk and a connection object.

After the line drawing command comes a comma and another such command. The command can be empty, which means that an extra comma at the end causes no harm. In fact, it helps, because after a sequence of edges another sequence can be appended without further punctuation.

```
(define (draw-edges points edges line-type drawing)
   (define (d-point pn)
     (let ( (p (vector-ref points pn)) )
       (string-append "(" (texnumber (car p)) "," (texnumber (cadr p)) ")")))
  (if (null? edges)
      drawing
      (draw-edges points (cdr edges) line-type
         (let ((edge (car edges)))
           (append drawing
              (list (string-append
                       (d-point (car edge)) ";"
                       (d-point (cadr edge)) "**@{" line-type "},")))))))
```

We can see how that works (and the layout doesn't) by:

```
(draw-edges (vector '(0 0) '(0 1) '(1 0))                              '((0 1)(0 2)) "-" '
(show (draw-edges (vector '(0 0) '(0 1) '(1 0))
                          '((0 1)(0 2)) "-" '() ) )
⇒((0.000000,0.000000);(0.000000,1.000000)**@{-}, (0.000000,0.000000);(1.000000,0.000000)**@{-},
```

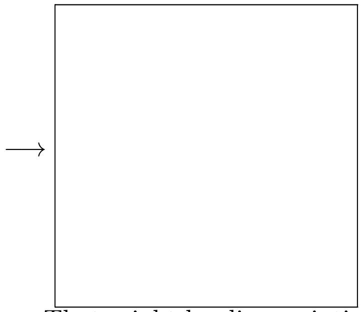
The next and the preceding procedures were originally one, but I separated set up from edge drawing so that *draw-edges* could be re-used later. The *line-type* was built-in, as it is always `"-"`, but it will be changed later, so it was made a parameter.

The odd command in the second line sets the origin in the usual place and the point with coordinates $x = 1, y = 0$ is 2cm left and 0cm up.

```
(define (draw-simple points edges)
   (append
    (list "\\begin{xy}"
          "0;<2cm,0cm>:")
    (draw-edges points edges "-" '())
    (list "\\end{xy}%") ))
```

Now give it a simple test of actual drawing

```
(show-TeX (draw-simple cube-verts cube-edges))
```



That might be disappointing, but should have been expected. The reason the cube looks so flat is that it is drawn by projecting away the z co-ordinate. To look better it must be viewed from a different angle.

3.2 Rotating the Drawing

Define three procedures to produce rotation matrices around the x, y, and z axes. The rotation angle is given in "pi-rads", which are radians times π. Thus $1/2$ is a right angle, while $1/6$ is $30°$.

These are called "elementary rotation matrices", and are described in many books [11, 1]. Unfortunately, the sign conventions are not consistent between these references. The conventions here agree with Battin [1], page 95.

Perhaps the different conventions are due, in part, to the fact that there are two ways to interpret multiplication by a matrix. It can be seen as computing the coordinates of the same points in a rotated coordinate system, or rotated points in the same coordinate system.

In the present context it seems easist to visualize the coordinates as fixed in the paper, x to the right, y toward the top, and z perpendicular to the paper toward the viewer. The elementary matrices rotate the drawn object around one of these axes, in a right handed direction (when the thumb of your right hand points in the positive direction along the axis, the fingers curl in the direction of rotation.

```
(define pi 3.14159265358979)
(define (x-rot a)
  (let ( (s (sin (* pi a))) (c (cos (* pi a))) )
    (list   (list (+ 1) (+ 0) (+ 0))
            (list (+ 0) (+ c) (- s))
            (list (+ 0) (+ s) (+ c)) )))
(define (y-rot a)
  (let ( (s (sin (* pi a))) (c (cos (* pi a))) )
    (list   (list (+ c) (+ 0) (+ s))
            (list (+ 0) (+ 1) (+ 0))
            (list (- s) (+ 0) (+ c)) )))
(define (z-rot a)
  (let ( (s (sin (* pi a))) (c (cos (* pi a))) )
    (list (list (+ c) (- s) (+ 0))
          (list (+ s) (+ c) (+ 0))
          (list (+ 0) (+ 0) (+ 1)) )))
```

Procedure **mv* multiplies a matrix with a vector.

```
(define (*mv m v)
  (map (lambda (row)(apply + (map * row v))) m))
```

Procedure *mm multiplies matrices. It chokes if given a ragged list of rows, or incompatible matrices. It may look strange if you expect to see $\sum_{j=1}^{n} a_{ij} b_{jk}$. This procedure uses no indexing. It concatenates columns obtained by multiplying the first matrix with successive columns of the second matrix.

```
(define (*mm a b)
  (if (null? (car b))
      (map (lambda (x) '()) a)
      (map cons
           (*mv a (map car b))
           (*mm a (map cdr b)))))
```

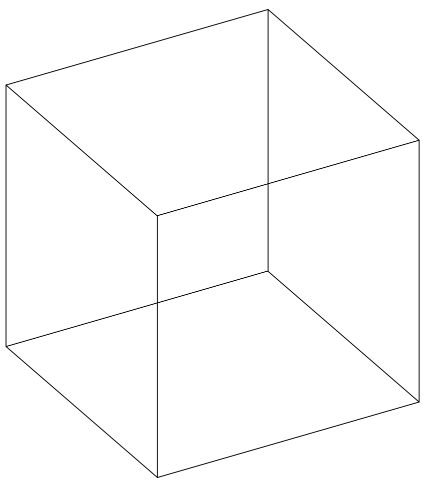
Procedures *vmap* applies a function to each member of a vector and returns a (same size) vector of the results, *rot* uses that to rotate a vector of points i.e. compute a vector of new coordinates.

```
(define (vmap fn v)
  (list→vector (map fn (vector→list v))))
(define (rot rotation points)
  (vmap (lambda (p) (*mv rotation p)) points) )
```

Let's give that a try by making a cube that is rotated $\frac{1}{3}\pi$ radians around two axes.

```
(define rcube-verts (rot (x-rot 1/6) (rot (y-rot 1/6) cube-verts)) )
(show-TeX (draw-simple rcube-verts cube-edges))
```

⟶



3.3 More Elaborate Drawing

That works, but to go any further we need a little more flexibility in the drawing procedure.

The procedure *draw-polyhedra* takes any number of polyhedra and a rotation. It draws all of the polyhedra in one picture with the same rotation applied to each of them.

A polyhedron is represented as a list consisting of a vector of vertex coordinates, a list of edges, an optional line format character which shows how the edges are to be drawn, and an optional 'N which causes the vertices to be numbered by their positions in the vertex vector. The line format characters are "-", ".", or "=", which produce an ordinary line, a dotted line, or a double line, respectively.

```
(define (draw-polyhedra-cmds rotation . polyhedra)
  (let draw ( (polys polyhedra)
              (drawing (list)) )
    (if (null? polys)
        drawing
        (draw (cdr polys)
              (append
               drawing
               (let* ( (poly (car polys))
                       (points (if rotation
                                   (rot rotation (car poly))
                                   (car poly)))
                       (edges (cadr poly))
                       (line-type (if (< 2 (length poly))
                                      (caddr poly)
                                      "-")) )
                 (define (d-point pn)
                   (let ( (p (vector-ref points pn)) )
                     (string-append "(" (texnumber (car p)) ","
                                    (texnumber (cadr p)) ")")))
                 (define (label-points points)
                   (let ((res ""))
                     (do ( (n 0 (+ 1 n)) )
                         ((>= n (vector-length points)) res)
                       (set! res
                             (string-append res (d-point n)
                                            "*{" (number->string n) "};")))))
                 (append
                  (if (< 3 (length poly))
                      (list (label-points points))
                      '())
                  (draw-edges points edges line-type '()) )) )))))
```

In contrast to the *draw-simple*, the *drawing-proc* procedure does not produce a drawing, instead it returns a procedure which when called will produce a drawing at particular size and possibly with a rotation applied before any further rotation supplied as a parameter.

```
(define (drawing-proc size tilt)
    (lambda (rot . polys)
      (let ((rotilt (or (and tilt rot (*mm rot tilt)) tilt rot)))
        (append
         (list
          "\\begin{xy}"
          (string-append "0;<" size ",0cm>:"))
         (apply draw-polyhedra-cmds (cons rotilt polys))
         (list "\\end{xy}%") ))))
```

4 The Regular Polyhedra

4.1 Tetrahedron

The easiest way to make a tetrahedron is just to draw diagonal edges between every other pair of vertices of a cube.

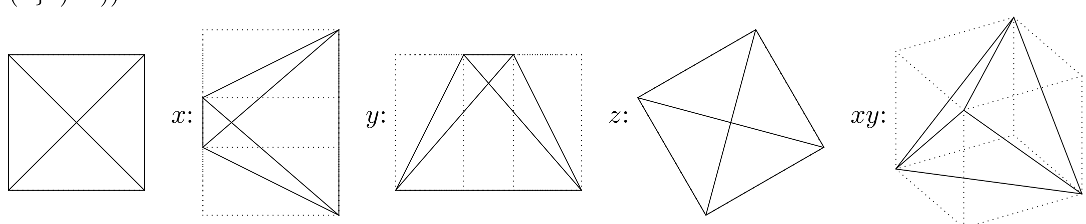
```
(define tetrahdn-edges '((0 3)(0 6)(0 5)(3 6)(6 5)(5 3)))
```

Let's make the drawing small so there is room to print it several times.

```
(define draw-small-polyhedra (drawing-proc "0.9cm" #f))
(define xy (*mm (x-rot 1/6) (y-rot 1/6)))
(define frame (list cube-verts cube-edges "."))
(define tetrahdn (list cube-verts tetrahdn-edges "-"))
```

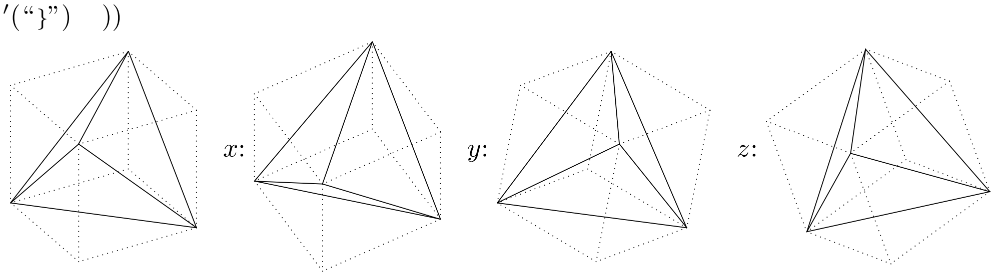
The `\mbox` TEX command makes all the little drawings stay together without the line breaks that might otherwise be inserted between them.

```
(show-TeX
  (append
  ı '("\\mbox{") (draw-small-polyhedra #f frame tetrahdn)
  ı '("\\quad␣$x$:\\␣") (draw-small-polyhedra (x-rot 1/6) frame tetrahdn)
  ı '("\\quad␣$y$:\\␣") (draw-small-polyhedra (y-rot 1/6) frame tetrahdn)
  ı '("\\quad␣$z$:\\␣") (draw-small-polyhedra (z-rot 1/6) frame tetrahdn)
  ı '("\\quad␣$xy$:\\␣") (draw-small-polyhedra xy frame tetrahdn)
  ı '("}")   ))
```

$\longrightarrow$ 

That last one is the only one that actually looks right, so let's make that the starting point of three rotations.

```
(define draw-tilted-polyhedra (drawing-proc "0.9cm" xy))
(show-TeX
  (append
  ı '("\\mbox{") (draw-tilted-polyhedra #f frame tetrahdn)
  ı '("\\quad␣$x$:\\␣") (draw-tilted-polyhedra (x-rot 1/9) frame tetrahdn)
  ı '("\\quad␣$y$:\\␣") (draw-tilted-polyhedra (y-rot 1/9) frame tetrahdn)
  ı '("\\quad␣$z$:\\␣") (draw-tilted-polyhedra (z-rot 1/9) frame tetrahdn)
  ı '("}")   ))
```

$\longrightarrow$ 

4.2 Six Edges

Both the dodecahedron and the icosahedron can be inscribed in a cube with an edge lying within each face of the cube. The edges are centered on the face of the cube. One pair is at the intersection of the plane $x = 0$ with the faces $y = \pm 1$, the other pairs are at the intersection of $y = 0$ with $z = \pm 1$ and $z = 0$ with $x = \pm 1$. Since we don't know yet how long we want the edges to be (they will differ in the icosahedron and dodecahedron) we make a procedure that will determine the twelve vertices for any given length.

```
(define (twelve-vertices a)
  (list→vector
    `((1 ,(+ a) 0) (1 ,(- a) 0)
      (-1 ,(+ a) 0) (-1 ,(- a) 0)
      (,(+ a) 0 1) (,(- a) 0 1)
      (,(+ a) 0 -1) (,(- a) 0 -1)
      (0 1 ,(+ a)) (0 1 ,(- a))
      (0 -1 ,(+ a)) (0 -1 ,(- a)) )))
(define six-edges '((0 1)(2 3)(4 5)(6 7)(8 9)(10 11)))
```

Make a set of edges that extend all the way to the edges of the cube. Number them to make it easier to connect them properly. (I look at a numbered diagram and type in the definition of the edge list. It would be more elegant to generate these lists automatically, but I don't know how.)

```
(define long-six-edges-numbered (list (twelve-vertices 1) six-edges "-" 'N))
```

Make another version with edges half as long drawn with double lines.

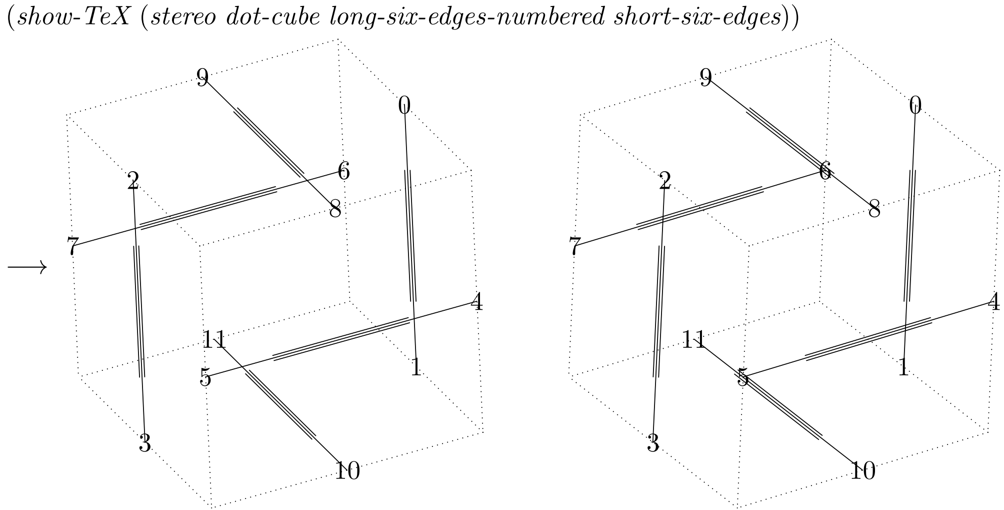
```
(define short-six-edges (list (twelve-vertices 1/2) six-edges "="))
```

Draw the cube with dotted lines.

```
(define dot-cube (list cube-verts cube-edges "."))
```

Now lets draw them. By drawing two copies next to each other, each rotated a little along the vertical axis in opposite directions, we get a stereographic pair. If you cross your eyes until the two copies merge into one, it will appear three dimensional (sort of).

```
(define draw-polyhedra (drawing-proc "2cm" xy))
(define (stereo . polyhedra)
  (append '("\\mbox{")
    (apply draw-polyhedra (cons (y-rot -1/40) polyhedra)) '("\\hspace{1cm}")
    (apply draw-polyhedra (cons (y-rot 1/40) polyhedra)) '("}") ))
(show-TeX (stereo dot-cube long-six-edges-numbered short-six-edges))
```

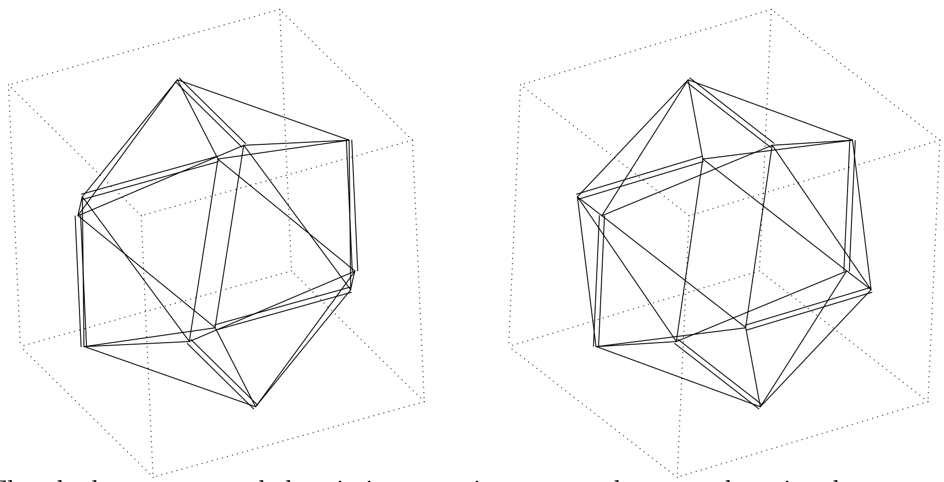


4.3 Icosahedron

Given the above picture with the vertices numbered, it is easy to define a list of other edges needed to draw the icosahedron.

(**define** *ico-other-edges*
′((0 8) (0 9) (1 10) (1 11)
ı (2 8) (2 9) (3 10) (3 11)
ı (8 4) (8 5) (10 4) (10 5)
ı (9 6) (9 7) (11 6) (11 7)
ı (4 0) (4 1) (5 2) (5 3)
ı (6 0) (6 1) (7 2) (7 3)))
(**define** *other* (*list* (*twelve-vertices* 1/2) *ico-other-edges* “-”))
(*show-TeX* (*stereo dot-cube short-six-edges other*))

$\longrightarrow$

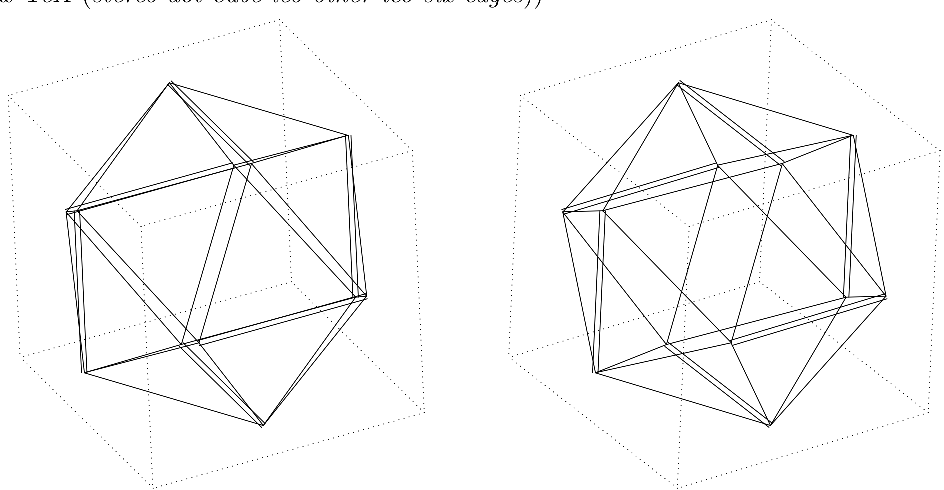


That looks pretty good, but it is not quite correct, because the triangles are not equilateral. Look at the pentagon formed by the bases of the five triangles around a single point. To make the whole thing symmetric the length of the six edges that lie on the faces of the cube must be exactly as long as the other edges.

Now the length of the six edges is just $2a$; the length of the other edges is $\sqrt{1^2 + (1-a)^2 + a^2}$. If these are to be equal, we must have $4a^2 = 1^2 + (1-a)^2 + a^2$ or $1 + 1 - 2a - 2a^2 = 0$, that is $1 - a - a^2 = 0$, or $a = (1 \pm \sqrt{1+4})/(-2)$.

(**define** *a* (/ (- 1 (sqrt 5)) -2))
(**define** *ico-six-edges* (*list* (*twelve-vertices a*) *six-edges* “=”))
(**define** *ico-other* (*list* (*twelve-vertices a*) *ico-other-edges* “-”))
(*show-TeX* (*stereo dot-cube ico-other ico-six-edges*))

$\longrightarrow$

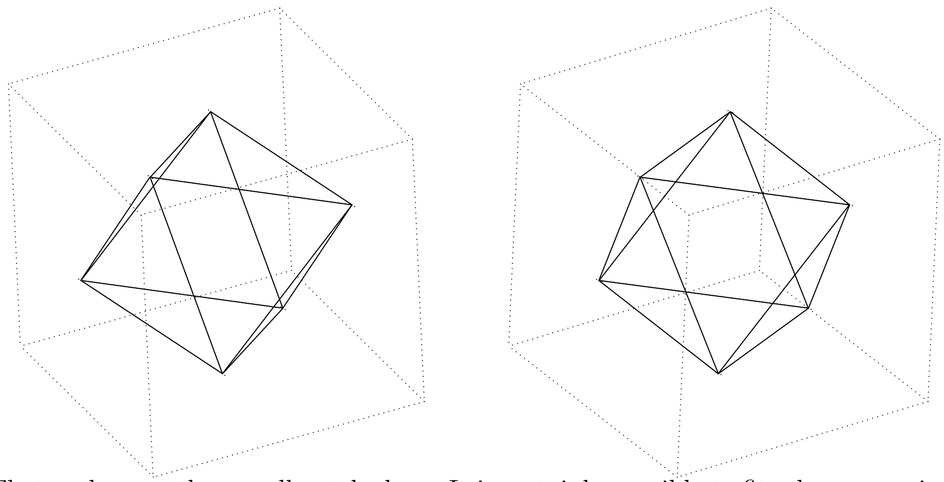


4.4 Octahedron

By the way, we can easily draw an octahedron using exactly the same procedures, by setting the length of the six edges to zero, so that they become single points in the center of the cube faces.

(**define** *octa-six-edges* (*list* (*twelve-vertices* 0) *six-edges* "="))
(**define** *octa-other* (*list* (*twelve-vertices* 0) *ico-other-edges* "-"))
(*show-TeX* (*stereo dot-cube octa-other octa-six-edges*))

$\longrightarrow$



That makes a rather small octahedron. It is certainly possible to fit a larger one inside the cube. Is it possible to inscribe one with all the vertices on the faces of the cube? If not, how many vertices can be on the faces? Can some edges of the 8-hedron lie in the faces of the cube? What are the symmetries of the solution?

4.5 Dodecahedron

To draw a dodecahedron, we need eight more vertices, for a total of $12 + 8 = 20$. These will lie in the long diagonals that run from the origin at the center to the eight corners of the cube. Thus they will be the eight corners of a smaller cube. Call the length of an edge of the small cube "2*b*".

Compute the coordinates of the eight vertices.

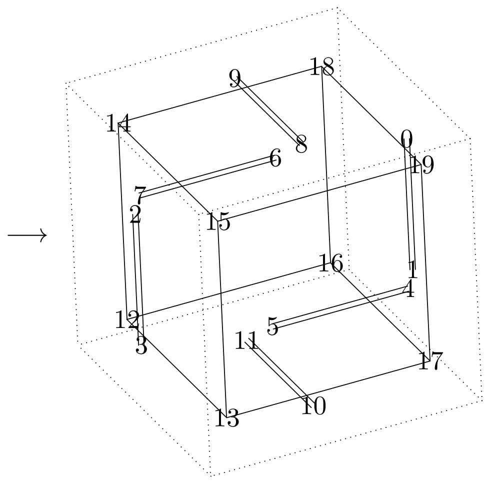
(**define** (*eight-vertices b*)
    (*vmap* (**lambda** (*p*) (*map* (**lambda** (*c*) (* *b c*)) *p*)) *cube-verts*))

These eight extra vertices are the vertices of a smaller cube.

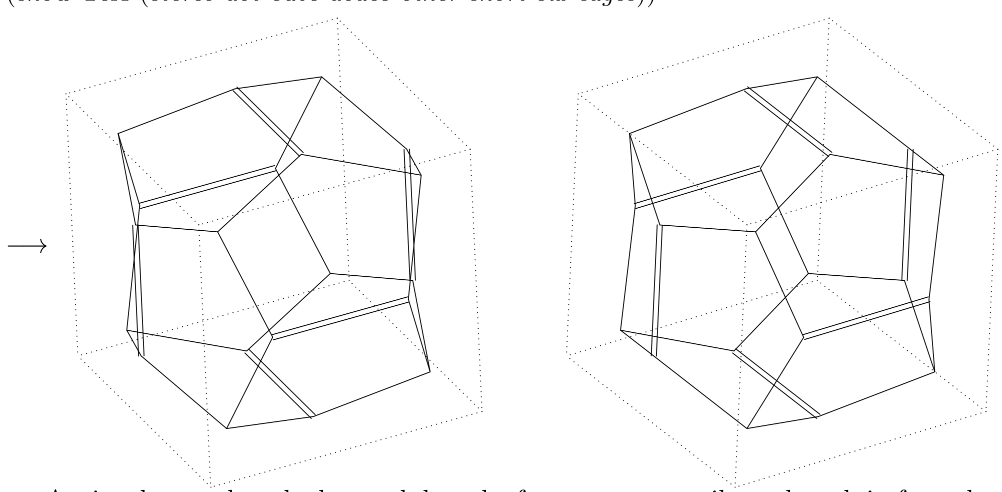
(**define** *inside-cube-edges*
    '((0 1)(0 2)(0 4)(1 3)(1 5)(2 3)(2 6)(4 5)(4 6)(3 7)(5 7)(6 7)))

We need to put the new vertices into the same vector of points with the "*six-edges*" so that edges can be drawn between points in the two sets.

(**define** (*vector-append v w*)
   (list→vector (*append* (vector→list *v*) (vector→list *w*))))
(**define** (*renumber-points n edges*)
   (*map* (**lambda** (*edge*) (*map* (**lambda** (*c*) (+ *n c*)) *edge*)) *edges*))
(**define** *twenty-verts* (*vector-append* (*twelve-vertices* 1/2) (*eight-vertices* 3/4)))
(**define** *inside-cube-renumbered* (*renumber-points* 12 *inside-cube-edges*))
(**define** *inside-cube* (*list twenty-verts inside-cube-renumbered* "-" 'N))
(*show-TeX* (*stereo dot-cube inside-cube short-six-edges*))




```
(define dodec-other-edges
   '((0 19)(0 18)(1 16)(1 17)   (2 14)(2 15)(3 12)(3 13)
     (4 17)(4 19)(5 13)(5 15)   (6 16)(6 18)(7 12)(7 14)
     (8 15)(8 19)(9 14)(9 18)   (10 13)(10 17)(11 12)(11 16)) )
(define dodec-other (list twenty-verts dodec-other-edges "–"))
(show-TeX (stereo dot-cube dodec-other short-six-edges))
```



Again, the topology looks good, but the faces are not equilateral, and, in fact, they are not even flat.

We must make the edges equal in length. The six edges in the cube faces still have length $2a$, the other edges have length $\sqrt{(1-b)^2+b^2+(b-a)^2}$, so $4a^2 = (1-b)^2+b^2+(b-a)^2 = 1-2b-2ab+3b^2+a^2$. In addition we must ensure that the vertices are all the same distance from the center. The twelve vertices are at a distance $\sqrt{1^2+a^2}$ from the origin, while the eight other vertices are at a distance $\sqrt{b^2+b^2+b^2}$, so $3b^2 = 1+a^2$, thus $b = \frac{1}{\sqrt{3}}\sqrt{1+a^2}$

Those equations are difficult to solve in closed form. Instead we use a method called "false position", or *"regula falsi"* [7, p.354].

Procedure *find-zero* has four parameters: *fn* a function for which we want a zero, *a* and *b* an upper and lower bound on an interval in which we will search for the zero, *delta* the tolerance.

```
(define (find-zero fn a b delta)
  (define (find0 low hi)
    (let* ((mid (/ (+ low hi) 2))
           (midval (fn mid)))
      (if (< (abs (- hi low )) delta)
          mid
          (if (< midval 0)
```

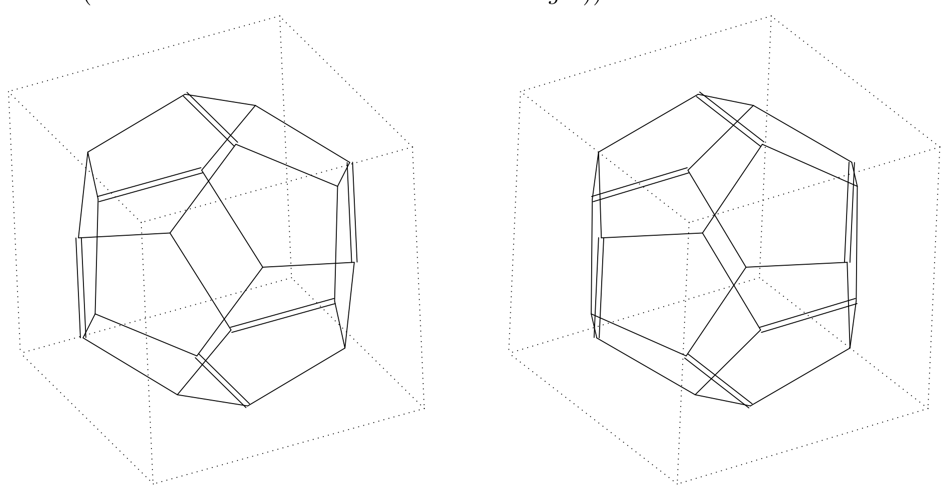
```
              (find0 mid hi)
              (find0 low mid)))))
  (let ( (aval (fn a))
         (bval (fn b)) )
    (or (and (< aval 0 bval)(find0 a b))
        (and (< bval 0 aval)(find0 b a)) )))
```

we can test that by finding the zero of the sin function that is between 1 and 4, which should be π.

```
(show (find-zero sin 1.0 4.0 1.0e-6))
⇒3.1415926218032837
```

Close enough!

```
(define (b-of a) (sqrt (/ (+ 1 (* a a)) 3)))
(define a
  (find-zero (lambda (a)
               (let ((b (b-of a)))
                 (- (* 4 a a) (+ 1 (* -2 b)(* -2 a b)(* 3 b b)(* a a))) ))
             0.0 1.0 1.0e-6))
(define b (b-of a))
(define dodec-verts (vector-append (twelve-vertices a) (eight-vertices b)))
(define dodec-six-edges (list (twelve-vertices a) six-edges "="))
(define dodec-other (list dodec-verts dodec-other-edges "-"))
(show-TeX (stereo dot-cube dodec-other dodec-six-edges))
```

⟶ 

5 Slightly Irregular Polyhedra

By relaxing the regularity requirements a bit it is possible to go beyond the Platonic solids to make several kinds of almost regular polyheda. For example, a prism with all vertices congruent can be made from two trianges and three squares. Another generalization is to allow the faces to be non-convex. There are several almost regular polyhedra with pentagrams for faces. Drawing them is left as an exercise.

References

[1] Richard H. Battin, *An Introduction to the Mathematics and Methods of Astrodynamics*, AIAA Education Series(1987)

[2] Helmut Kopka and Patrick W. Daly, *A Guide to LaTeX, third edition*, Addison Wesley (1999)

[3] Donald E. Knuth, *Literate Programming*, Computer Journal 27(1):97–111, 1984.

[4] Donald E. Knuth, *Literate Programming*, CSLI, 1992. CSLI Lecture notes no. 27.

[5] Donald E. Knuth, *TEX: The Program (Volume B of Computers and Typesetting)*, Addison-Wesley, 1986, ISBN 0-201-13437-3

[6] Donald E. Knuth, *Metafont: The Program (Volume D of Computers and Typesetting)*, Addison-Wesley, 1986, ISBN 0-201-13438-1

[7] William H. Press, Saul A. Teukolsky, William T. Vetterling and Brian P. Flannery, *Numerical Recipes in **C** (Second Edition)*, Cambridge University Press (1992)

[8] Norman Ramsey, *Literate programming: Weaving a language-independent web*, Communications of the ACM, 32(9):1051–1055, 1989.

[9] Norman Ramsey *The noweb Hacker's Guide*, Princeton University, 1992. Revised 08/1994.

[10] John D. Ramsdell *SchemeWEB – WEB for Lisp. Simple support for literate programming in Lisp.* The MITRE Corporation, 1994

[11] David F. Rogers and J. Alan Adams, *Mathematical Elements for Computer Graphics*, McGraw-Hill (1976)

[12] Kristoffer H. Rose and Ross Moore, *XY-pic Reference Msnual*, BRICS Computer Science, Aarhus University, Denmark (1998)

[13] Will Clinger, et. al. *Revised^7 Report on the Algorithmic Language Scheme*,

[14] Michael Sperber, et. al. *Revised^6 Report on the Algorithmic Language Scheme*, Cambridge University Press ISBN: 9780521193993 (2010) or http://www.r6rs.org/

[15] Richard Kelsey, et. al. *Revised^5 Report on the Algorithmic Language Scheme*,

[16] Aaron Hsu, *ChezWEB User's Guide*, gopher://gopher.sacrideo.us/1chezweb

[17] Dorai Sitaram, *SLaTeX*, 1991, 1999
`http://www.ccs.neu.edu/ dorai/slatex/slatex.tar.gz`

[18] Alfred North Whitehead, *Process and Reality*, Macmillan Company (1929)