7

Experiments in EdenLisp

This chapter deals with experiments in EdenLisp that test the features of the implementation and illustrate its potential for design. A reasonable criticism of EdenLisp might be that it does not add anything to existing methods for problem-solving. The potential of EdenLisp is particularly in terms of how it uncovers relationships that are based on observation and state. However it would be a poor tool if it did not do at least what other methods can do. So we begin with some experiments with the language that illustrate straightforward design problems. We then compare approaches that use conventional programming tools and show that both economy of programming and exposure of design structures are also benefits of EdenLisp

7.1 Parametric Studies

7.11 Tumbler-Mixer Machine

Possibly the simplest way to use EdenLisp is to exploit its spreadsheet analogy to carry out design calculations. Because equations can be set out in conventional form without recourse to row-column tabular formulae peculiar to spreadsheets, an EdenLisp script reads more naturally as a design report of the design calculations. The ability to change individual definitions means that the sensitivity of particular variables may be examined interactively by a process of inspection as variables are redefined in a particular range. Such parametric studies are common in design as an interactive way of checking sensitivity, rather than using optimisation programs where the constraints have to be preconceived or where the effect of constraint relaxation is difficult to predict *a priori.*

As an example of the advantages of parametric studies of that kind, the following is one where I used EdenLisp to perform calculations for a local firm. The Company required a partial redesign of a tumbler-mixer machine that they use for mixing fine powders such as those used in the food industry.

The arrangement is shown in *fig. 7.1.* The flask contains up to 4 tonnes of powder and is inserted into a cage that is rotated about a horizontal axis. The drive shaft is attached to the cage at a position such that the cage is rotated about its vertical axis by 30°. That means that the powder is thrown from left to right as it rotates. The requirement was for the shaft to be of the correct diameter for the steady and impact loading as the powder falls about during the mixing process.

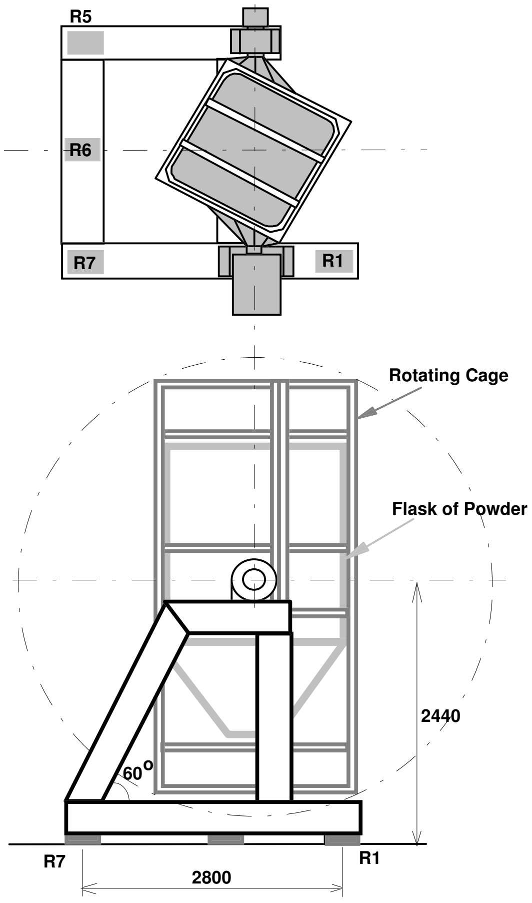


Fig. 7.1 Powder Tumbler- Mixer Machine

The EdenLisp script shown in *Appendix B* contains the formulae needed to calculate the shaft diameter. Its form is very analogous to a spreadsheet as no graphics are required for this computation. It is simply used for the parametric study. From that study it was found that, for example the impact loading was the most significant factor in the shaft design, whereas a hollow shaft could be specified with very little increase in the outer diameter. Different ways were explored to determine the machine stiffness and the effective load due to impact from falling powder.

7.12 Four-Bar Linkage

A reasonable extension of the spreadsheet analogy is to make a "graphical spreadsheet" in the form of calls to AutoCAD. We show that we can make a system that automatically up-dates an AutoCAD drawing as the user changes any of the declared parameters. The example used is the design of a 4-bar linkage. The EdenLisp script for this (*Appendix B*) is similar in structure to the tumbler design in the previous section, but with the addition of constructions for creating geometrical objects by the method described in earlier chapters. The geometry of a 4-bar linkage appears simple in terms of the drawing but the relationships are non-trivial. For example in *Fig 7.2* given the co-ordinates for the points `J0`, `J1` and `J3`, the point `J2` is difficult to determine. The formulae for finding `J2` are as follows

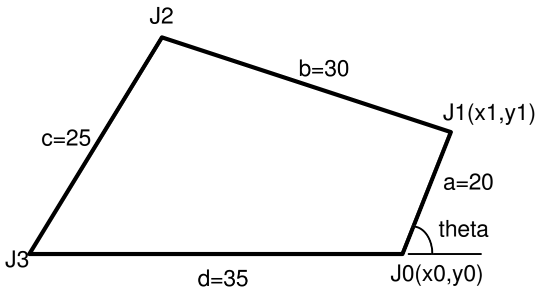


Fig 7.2 4-Bar linkage design in EdenLisp

```
L    =   - sin(theta)                 ;Computations to establish angle delta
M    =  (d/a) + cos(theta)
S    =  a*a + c*c + d*d - b*b
K    =  d*cos(theta)/c + S/(2*a*c)
sq   =  sqrt (L*L + M*M - K*K)
delta=  2*atan((L + sq)/(M - K))      ; Compute angle delta from given data

x    =  -(d + c*cos(delta2))          ; hence find co-ords of point J2
y    =  c*sin(delta2)
```

In the script nodes `J0` to `J3` are written as strings within a variable `nodes1` of type `LLstr` (list of string) as follows.

```
nodes1 = ["J0", "J1", "J2", "J3"]
nodes2 = cedge(nodes1)
```

The function `cedge` creates connections between the nodes and can set up a number of instantiations that have the same connectivities but different realisations. Function `cedge{nodes1)` on the four nodes yields the string connections:

```
(("J0" "J1") ("J1" "J2") ("J2" "J3") ("J3" "J0")).
```

Using these topological relationships the coordinates for the nodes are assigned and the strings made into variables with corresponding edge connections via the *`complex`* operator.

```
barb = complex (nodes2)            ; create variables making up the 4-bar
```

The variables `J0` to `J3` now acquire their values from the definitions as follows

```
J0 = [0.0,0.0]                     ; coords of fixed pivot of driver bar a
J1 = [a*cos(theta), a*sin(theta)]  ; coords of  moving end of driver bar a
```

```
J2 = [x, y]                           ; coords of  driven end of bar c
J3 = [-d,0.0]                         ; coords of  fixed pivot of driven bar c
```

The `object` operator computes the set of coordinates resulting from moving the nodes to be about a given origin and with given scaling factor on each dimension. The object can be realised from its skeletal structure, and displayed as single straight line segments or bars of complex geometry. The definitions are:

```
barf = object(barb,[300, 150],[5,5])  ; create 4-bar at (300,150) to scale*5
bard = Wireframe (barf, "pline")      ; draw the 4-bar as a polyline
```

Given the geometry in *fig.7.2* we can change the definition or value of any declared variables. The corresponding coordinates are recalculated and the values of draw functions representing the lines get changed too. The effect of a draw-function change is to delete the AutoCAD handle, the address where the display information is found. The deletion actually removes the displayed entity and enables the new drawing function to display the new position without retaining the previous display. That means it is possible to have an animation by simply changing a variable sequentially. In the code that is done by having a set of values explicitly defined in turn.

An interesting feature of the display is the labelling. The EdenLisp operator `label` is designed to deal with labels so that the considerable flexibility built into AutoCAD regarding font type, shape and size can be used. At the same time the power of EdenLisp is added, in that labels can be attached to objects in such a way that their attributes may be defined in relation to that object. The code for label "BarA" is

```
Str    : labela
Lreal  : Lpta

J4     = midpt(J0,J1)
Lpta   = locate(j4,origin,size)
labela = label(Lpta, 5.0, "barA")    ; label it
```

The `label` function takes three arguments, the location of the text in object space, a list of font attributes such as height, width and justification, and the text string itself or a variable that generates a text string. With that operator, labels can be moved with the object, the text can be changed to report something about the object, the position relative to the object can be moved such that it not obscured by graphic entities. In *fig 7.3* labels `BarA`, `BarB` and so on get moved in relation to their appropriate bars.

Fig.7.3 shows six possible arrangements of the 4-bar linkage that one might expect by varying the angle `theta.` We should be able to produce all six pictures on the display by creating different instantiations of the frame with different origins and scales. These would be distinct realisations with independent coordinate sets. However all instantiations made from the same frame will be geometrically identical since all the geometrical descriptions are based on the same definitions. Thus the picture with six objects cannot be realised unless there is an identical set of definitions for each object but on a different set of variables, *e.g.* the angle theta is a *different variable* on each object.

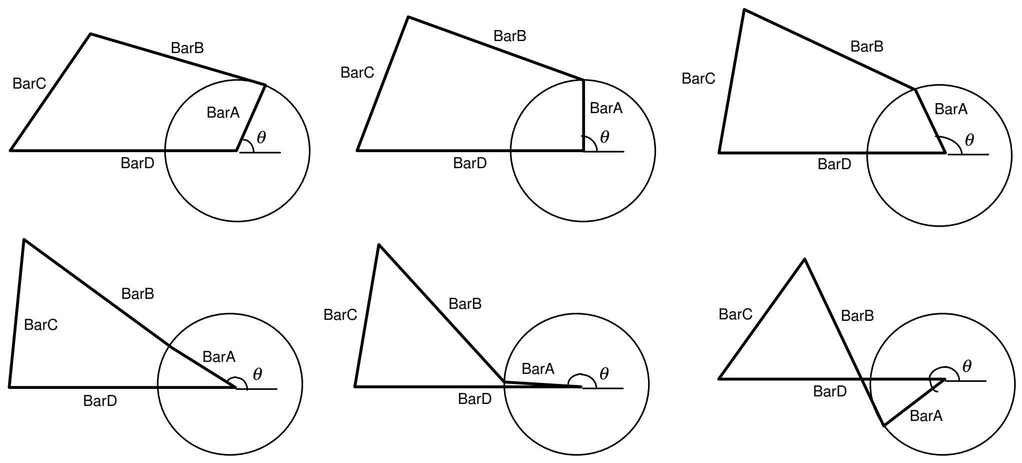


Fig.7.3 Arrangements of a 4-bar Linkage for different theta values

The problem raised here is a fundamental one. If we wish to show simultaneous pictures then we need to have different 'windows' for holding the different values. Alternatively we can regard each Intantiation as a *suspended state*. A picture is constructed and displayed with one script of definitions and then left in that state. A separate copy of those definitions is then created as a different script and that script becomes the current state to be changed at will. This scenario is similar to that used in windowed systems such as MS Windows where a window is only active while made so explicitly.

7.13 Drawing Frame

A comparison of EdenLisp with AutoLisp is instructive. AutoLisp may be used to construct parametric drawings, an example of which is a parametric that draws a standard drawing frame and title block on a specified A-size paper. A 10 mm border is to be drawn around the paper with the title block in the bottom right-hand corner. The paper may be landscape or portrait. A difficulty is that the title block should not be directly proportional to the paper size, otherwise what is reasonable on A0 will be far too small on A6 and what suits A6 will be too large on A0. Comparing the parametric written directly in AutoLisp with the EdenLisp program shown in Appendix B (that incidently does the task in a very different way), the AutoLisp code ran to 4 pages compared with half that for the EdenLisp. The EdenLisp script is also more flexible. For example, a single change of variable will switch landscape to portrait and the labels may be changed at will. That means that if the drawing needs to be retitled with a much longer name midway through the exercise the whole block can be rearranged to enable the longer text string to be accommodated. It may be said, quite rightly, that it is easy to include that possibility in the AutoLisp parametric. But the number of possibilities extends to <u>all</u> variables in EdenLisp, not just those preconceived in setting up a parametric.

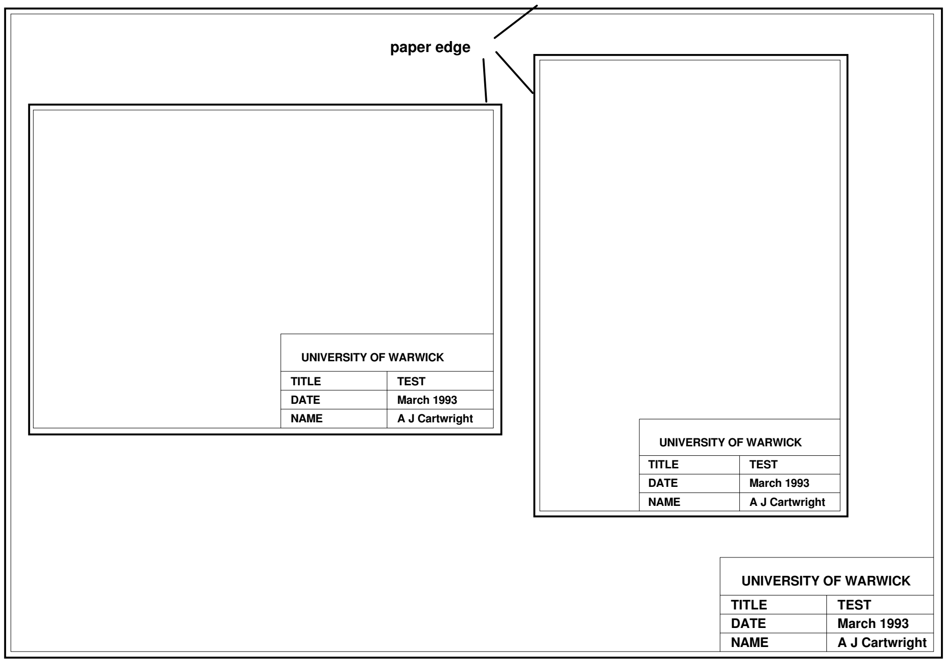


Fig 7.4 Sample outputs from Drawing Frame Program, superposed. Diagrams show A0, A6 landscape and A6 portrait (not to the same scale)

The problem of making the title block in reasonable proportion to the frame is a case in point. Realising the problem, it is possible to experiment with one or two basic definitions to arrive at a suitable formula that allows the title block to increase rather more slowly than the actual sheet size. (In fact a logarithmic relation was found to do the job!) This "sensitivity analysis" is precisely the activity that the spreadsheet is good for and so is natural for EdenLisp. *Fig. 7.4* shows outputs from the code, annotated and with paper outlines inserted for clarity.

7.14 Precision Balance

The final parametric design example explored in this section is the extension of the "wireframe" into surface modelling. Nothing new is added conceptually to the Definitive method by such extensions. Solids and surfaces are defined in terms of their generators and will be displayed by AutoCAD by simply calling the appropriate commands. Also properties of the resulting solids may be calculated by AutoCAD or by accessing information given in the definitions.

The concept used in this design is the so-called elastic hinge. [Smith ST & Chetwynd, 1992]. A sketch is shown in *fig. 7.5* of a typical notch type hinge. Because the thickness t is so small the top of the bar can be rotated forming a short-travel hinge. A monolithic construction is one in which hinges are machined from the solid by drilling holes that are 0.3 mm apart or less at their circumferences, and then joining such holes together with slots or saw cuts. An example of that construction is shown in *fig 7.6*

Calculations are based on the following notation

t	=	Thinnest section thickness, (mm)
R	=	Radius of curvature of the notch, (mm)
b	=	Width of material at the thinnest section, (mm)
F	=	Load, (N)
λ	=	Stiffness of one hinge, (N/mm)
E	=	Modulus of elasticity, (MPa)
θ	=	Maximum angle of rotation of hinge
L	=	Length of lever, (mm)
K	=	Correction factor for the notch stress concentration factor
Q_{max}	=	Maximum displacement of lever
T_{max}	=	Shear stress on hinge, (MPa)

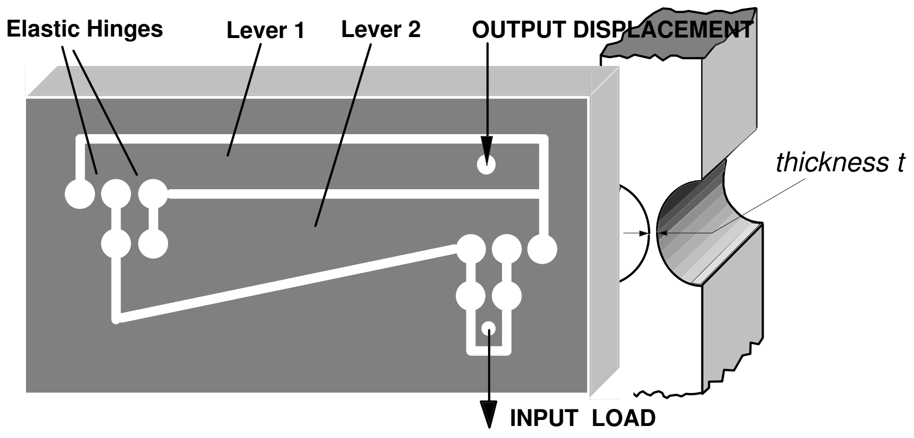


Fig. 7.5 Detail of Notch type Hinge

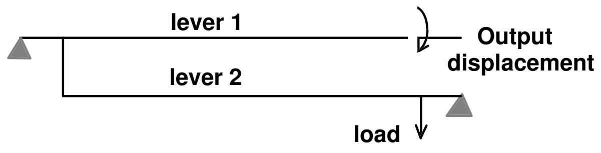


Fig.7.6. Monolith and equivalent Lever System

The following relationships obtain.

$\lambda \;=\; E\, b\, t^3 / (24\, K\, R\, L^2)$

$K \;=\; 0.565\, t / R \;+\; 0.166$

$M_{max} \;=\; b\, t^2\, \sigma_{\max} / (6\, Kt)$

$Kt \;=\; 0.325 + (2.7t + 5.4R\,) / (\, t + 8R)$

$\theta \;=\; 4K\, R\, \sigma_{max} / (Kt\, E\, t^2)$

$Q_{max} \;=\; L\, \theta$

$T_{max} \;=\; \sigma_{max}\, b\, t$

$B_{max} \;=\; 6\, M_{max}\, Kt / (t\, b^2)$

For aluminium we can insert some design figures as follows.

$E \;= 72000.0$

$b \;= 5.0$

$t \;= 0.3$

$R \;= 5.0$

$L \;= 55.0$

$\sigma_{max} \;= 100.0$

$F \;= \; 400.0$

$K \;=\; 0.565\, t / R \;+\; 0.166 = \; 0.565 \,.\, 0.3 / 5.0 \;+\; 0.166 \;=\; \underline{0.2}$

$\lambda \;=\; 72000 \,.\, 5 \,.\, 0.3^3 / 6 \,.\; 0.2 \,.\, 5.0 \,.\, 55^2 \;=\; \underline{0.54 \text{ N/mm}}$

Because of the complexity of the formulae it is difficult to decide what the important parameters are in the design of an individual notch. Numerous arrangements of the monolith construction can be made forming, complex lever systems of which one is shown in *fig.7.6*. The design of these lever systems is very much an interactive iteration, trying out many different arrangements of levers to get the most compact one that has the maximum lever ratio. The aim is to get the maximum lever ratio into the smallest space, and EdenLisp provided a neat tool to inspect different layouts. In the practical case different materials and arrangement were attempted, some of which were manufactured and tested with a view to being used in a precision weighing machine.

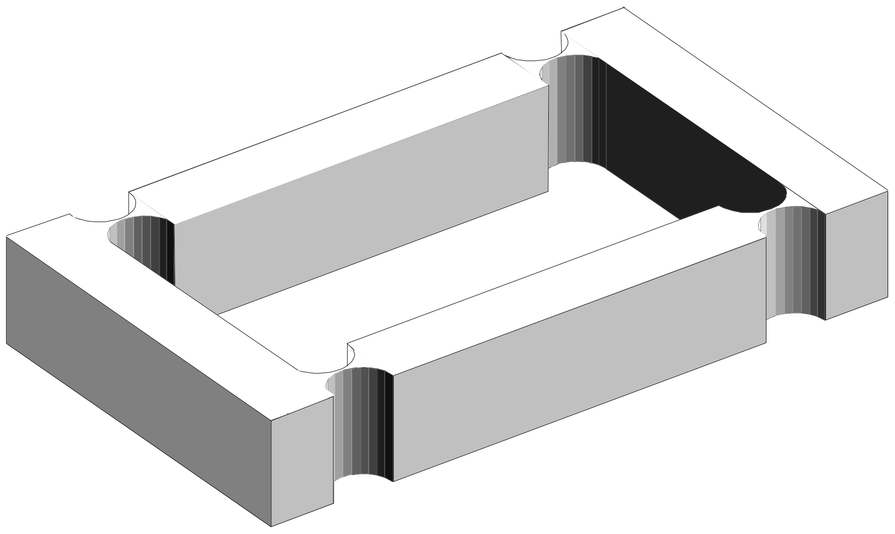


Fig. 7.7 Monolith construction in Three dimensions (with added shading)

The geometric construction of the notch was straightforward, but the display proved to be quite difficult because of rather arcane methods used by the AutoCAD system for accessing information already on the screen. The notch was constructed by using arcs and polylines, the only legal method for then "extruding" the plan shape into the third dimension. When the base shape has been constructed it is necessary to construct the extrusion vector and to refer to both the shape and the vector before the system obliges with the extrusion. Unfortunately the act of creating the extrusion vector renders the base shape inaccessible. It appears that the extrusion vector becomes attached to the base shape and so the latter cannot be addressed, even by its handle. The somewhat eccentric solution is to create the extrusion vector, point to it and then "delete" it before pointing to the base shape. The extrusion vector is then reinstated to allow the system to use it to create the extrusion itself. Because of that the shape shown in *fig.7.7* proved very difficult to obtain, even though it contains no new EdenLisp concepts. Some artistic licence has therefore been exercised on the shading in the figure.

7.2 Patterns in Design

It frequently happens in design that patterns can be detected as underlying the main problem and these provide families of solutions to different design problems. Examples abound: parametric design yields products of a similar shape topology; designs may be built from standard elements such as gear boxes and pumps, or from geometrically similar components such as turbine blades. Often functions have to be applied repetitively to different or similar components, or different functions affect the same set of components in different ways. The same principles may therefore be called for very different purposes. One of the benefits of the Definitive method of tackling design is that it forces one to consider those patterns, or alternatively patterns emerge from having to express the design definitively. We have already seen in the consideration of the shaft design in EDEN how the hierarchies identified as typifying design generally are exposed by the Definitive method. Now we illustrate how problems of this kind have been attempted in EdenLisp.

7.21 Analytical Graph Plotting

Arrays of entities are mentioned as pattern in design. Often the entities are identical, as in groups of standard fasteners holding the top of a machine element; sometimes there are trivial differences in adjacent entities such as in gate arrays for VLSI design. It is therefore desirable to have tools that generate dummy arrays that can be transformed into local entities as required.

The plotting of graphs provide an area for exploring arrays in EdenLisp. Arrays in this case consist of Cartesian pairs or triples, but even with such a simple array a problem of definition was identified in DoNaLD. If each point in a graph is significant then it should be defined in such a way that it does not depend upon other elements in the graph, nor should the number of graph elements need to be pre-declared, else the redefinition becomes clumsy. Giving each point a unique definition has the advantage that any point can be referenced or redefined. However the penalty is likely to be an information explosion that may not be necessary, or a tedious repetition of definitions that are virtually identical for each point. Action definitions provide a way to produce any type of definition; in particular lists can be created recursively to any size to create the required arrays to represent graphical data. As an example `Graph.Lsp` was implemented. The listing is remarkably short, given the amount of data created with unique access labels.

To make and plot an analytic function as a Cartesian graph the following sequence is adopted.

1. Create a list of length Npts
2. Write the analytic function to be plotted as a string, e.g. fn = "sin(1/x)"
3. Use mkgraf function to create the coordinates of the curve with input arguments Npts, range, varname (as string), funct (as string)
4. Draw standard axes, using range to scale the x-axis and the maximum value of the function to scale the y-axis
5. Draw the graph

EdenLisp functions and definitions that create and plot the analytic function graph are in *Appendix B.* The function at the commencement of the listing is used in the creation the array of coordinates. It illustrates the complexity of functions that underlie EdenLisp. It is frequently necessary to create such functions but the library that accompanies the EdenLisp code has been created over a period of time and covers many common applications. It is comparatively straightforward to add more functionality although it does have to be written in AutoLisp. Once the definitions are compete it is straightforward to annotate the graph with axes and labels as desired. The following figures and function were used to create the carpet graph shown in the output below. The axes are omitted for clarity.

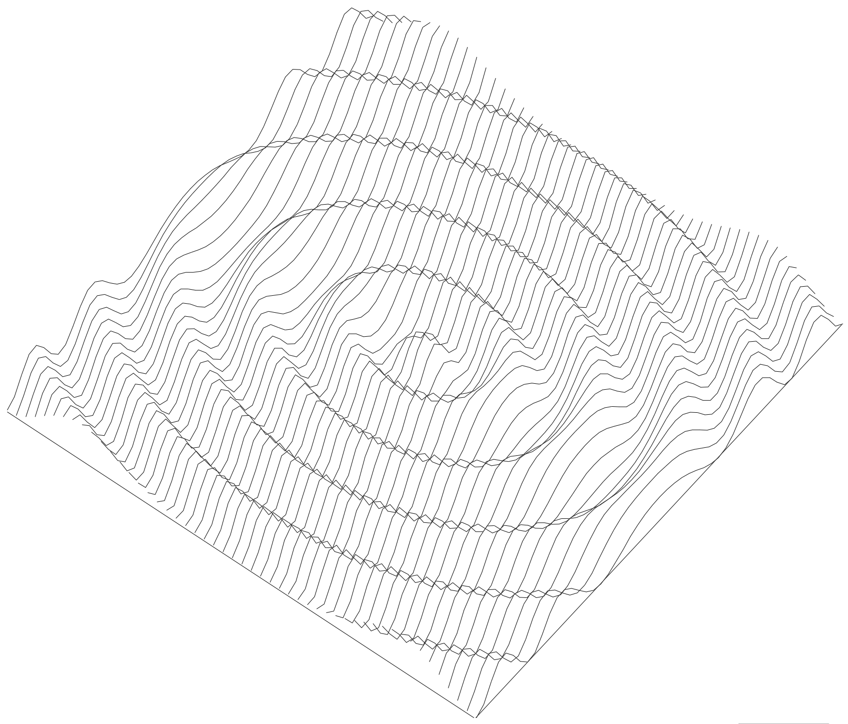


Fig 7.8 Output from Graph.Lsp for the function $z = \cos\sqrt{x^2 + y^2}$

```
Npts   = 50
xrange = 40.0
yrange = 40.0
func   = "cos(sqrt(x^2 + y^2))"        ; func = graph of analytic curve

flist  = mkgraf (Npts, xrange, yrange, func)
origin = [100.0,100.0,100.0]
gpts   = object(flist, origin, [1,1,1])
gline  = wireframe (gpts, "carpet")
```

The program generates 50 points on a single value of `y`, varying `x` by the amount `xrange/Npts` and then repeats that for `yrange/Npoints`, so producing 50x50 points and 50 lines and the shape shown in *fig 7.8*, like a series of ripples on a liquid. Because the result is a single array there are no labels for each point or line. That makes the method suitable for conditions where the individual points are not of interest. If the points are interesting then we need to use actions that generate point labels. We examine methods of doing that in the next section.

7.22 A Denture Design Aid

This problem came to my notice at a seminar [Randell, 1993] where David Randell presented the software package that his team had been doing in order to help the dental profession. The software, written in Prolog, was intended to enable a dental technician to design dentures. On starting the program the user was presented with a two dimensional diagram of a standard set of teeth, such as that shown in *fig.7.9*.

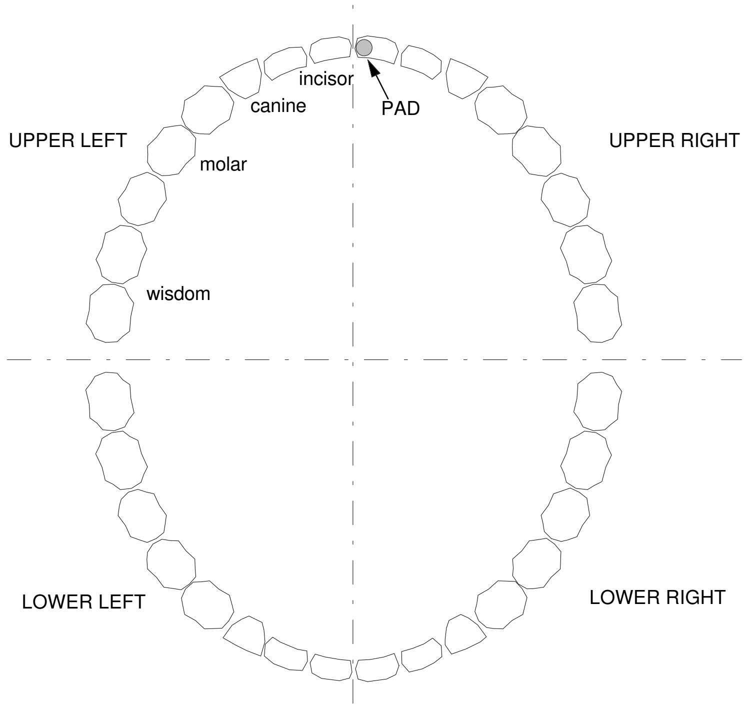


Fig 7.9 Output from the Dental Scripts

The user could then modify the diagram interactively from a menu of on-screen icons: teeth could be selected, removed, geometrically modified, moved a small distance, rotated slightly, or made artificial. The idea was that the dental technician could identify a patient' s teeth profile from a pressing and then take and modify the standard diagram to enable a denture to be designed around teeth that were missing or to be extracted. Dentures are designed using standard techniques: to hold the artificial teeth, to shape the body such that the denture is big enough not to be accidentally swallowed, to fit it comfortably in the mouth, and to do the job of biting and chewing. The rules for doing that were embodied in the Prolog program. For example, the user needs to shape the denture in such a way that forces on the artificial teeth from eating are transmitted to adjacent natural teeth, as the gums beneath the artificial teeth are unable to cope with such loads. That is achieved by bracing the denture against pads that fit into shaped orifices in the natural teeth, or by tension members hooked around the lowest point of the natural teeth next to the gums. All these members could be placed on the 2D diagram to show the design.

The problem with the software was identified during the seminar. The program was very long (described as over 20 mm thickness of A4 printout!) and although it was well structured it took considerable skill to modify the program to add new features. Constraints on the designer were built into the package to cope with various levels of user skill, so that warnings, helpful advice and prevention of impractical designs were conditional on the user' s level of understanding*i.e.* the constraints could be circumvented or removed by the expert user to reduce design time. Although the constraints were at various levels of severity they were preconceived. If new constraints were identified as desirable they could only be added by the programmer and further development was limited by the same requirement for a programmer.

The standard diagram is straightforward and can be generated quickly by any graphics system. The difficulties arise when it is desired to modify the tooth forms. The interactions to do that would be mechanical, leaving the basic reasoning and interpretation of the actions with the user. That would make the generation of constraining rules impossible. On examining the specification for constraining a design, the underlying pattern that emerged was that basic elements such as tooth, pad, hinge and plate could be represented as instantiations of particular sets of abstract definitions that give shape, position and orientation. By means of such archetypes, teeth could be placed in standard positions for quickly generating the initial set-up, but could be subsequently modified locally if necessary. Any tooth could also be formed with its own unique coordinate set but having its topology and family attributes editable in a global sense, *i.e.* wholesale changes to all the molars can be done, for example to model female, male or children' s teeth with very little effort. To carry out that approach the following analysis of the abstract program was made.

<u>Underlying pattern</u>

level 1 Abstractions

- topology of tooth forms
- topological relationships of teeth
- definitions of components for tooth form, pad, saddle, hook, plate for denture

level 2 Archetypes

- geometry of archetype tooth forms, pad, saddle, hook, plate for denture
- positions of teeth around mouth and of pads, saddles, hooks on teeth
- number and size of teeth for different people: male, female, child

level 3 Instantiations

- instantiation of archetypes in given position and with particular rotation
- copies of instances for standard repeats such as molars
- conversion of archetype into local edition of a form

level 4 Editing

- instantiation of variant editions for odd shaped teeth, local variations, positions of caries, pads, fillings, etc.
- extracted and artificial teeth
- editing of archetypal plate for desired shape

level 5 Constraints

- Functional restrictions on editing, e.g. teeth cannot be transplanted to other quarters, hooks are tension members, pads compression members, plates must cross quarters

7.23 Using Actions

The scripts in *Appendix B* as Program 7 generate the basic tooth arrangement in the mouth and the denture plate design using both definitions and actions. The levels described above correspond to sets of scripts: each set may have actions that rewrite or create new scripts for instantiating or editing purposes. Defining forty teeth separately by their shape coordinates is tedious when an array produced from an archetype tooth form will serve as well. Editing a shape interactively is easier than inputting coordinates separately and the user should be able to do either global or local changes with equal facility. The level 1 script consists of a set of definitions that yield finite sets of labels that can be used to define any shape. The set of action definitions

```
Mkincisor = A_lst ("incisor","i","lreal",10)
Mkcanine  = A_lst ("canine", "c","lreal",10)
Mkmolar   = A_lst ("molar",  "m","lreal",16)
Mkwisdom  = A_lst ("wisdom", "w","lreal",16)
```

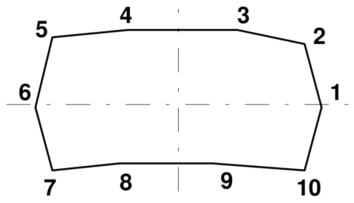
produces sets of definitions that are actioned as sets of labels of type `lreal` (list of real). Definition `Mkincisor` produces the following script.

```
 llreal : incisor
  lreal : i1 i2 i3 i4 i5 i6 i7 i8 i9 i10
incisor = [i1, i2, i3, i4, i5, i6, i7, i8, i9, i10]
```

The function `A_lst` attaches the string `"i"` in turn to each integer from 1 to the number given and then makes it a label, part of a list assigned to the label `incisor` derived from the first string. That label is then declared and the definition actioned. Each abstract action definition enables a new label list to be generated. Actions such as `A_Lst` are AutoLisp functions that return scripts of definitions. They are assigned type `lstr` (list of string).

Level 2 archetypes are easy to define using the pre-declared lists. All we need is to define Cartesian pairs to define a 2D representation of the shape of an arbitrary tooth, *e.g.* the following definitions are for an incisor tooth with shape indicated by the numbered diagram. Consistent numbering of the tooth form means that one can keep track of which is the outer and inner sides of the tooth form.

```
i1 = [ 17,  0]              ;
i2 = [ 15, 10]              ;
i3 = [  7, 12]              ;
i4 = [ -7, 12]              ;
i5 = [-15, 10]              ;
i6 = [-17,  0]              ;
i7 = [-15,-12]              ;
i8 = [ -7,-11]              ;
i9 = [  7,-11]              ;
i10= [ 15,-12]              ;
```



Insisor Tooth Shape

Once a form is ascribed all instantiations at level 3 can have the same form. Changes to any of the `i`-values will cause changes on all the incisors at once. If a local form is required it is simple to invoke another action definition to generate labels and use the same values of `i1, i2, ...,` etc. to initialise the local version.

The locations of teeth are defined by means of a definition of the "general mouth", an ellipse describing the shape of the mouth. The eccentricity of the ellipse is simple to redefine for different mouth shapes being done by scaling as in the following definitions for different mouth types.

```
male   = 1.0                ; scale size of mouth
female = 0.9
child  = 0.7
mouth  = male               ; define current mouth shape
ScM    = evalid(mouth)      ; use its real value
Rx     = 200.0*ScM          ; elliptical shape of mouth minor
                              axis
Ry     = 250.0*ScM          ; elliptical shape of mouth major
                              axis
```

The problem of large numbers of similar definition sets that was identified above applies equally when describing instantiations. One way of dealing with that is to use the technique used in the shaft analysis program implemented in DoNaLD. There new definitions were created using string substitution. For example to make a single instantiation of a tooth one needs to specify the tooth type, where it is to be located in the mouth in terms of which quarter and which position. The following script for example produces the Upper Right incisor at position `UR1pos`.

```
ang    = pi/Nteeth*2.24                    ; position increment round mouth
ang1   = (Nteeth + 0.2)*ang                ; actual position of first tooth

llreal : UR1typ                            ; declare tooth type
lreal  : UR1pos                            ; declare tooth position type
frame  : UR1ins UR1dsp                     ; declare object and display types
UR1pos = [Rx*cos(ang1), Ry*sin(ang1)]      ; middle of tooth profile
UR1typ = rotobj("incisor",pi/2-ang1,1.0)   ; orientation & type of tooth
UR1ins = object(UR1typ, UR1pos, scalea)    ; instance of tooth
UR1dsp = wireframe(UR1ns, tooth)           ; display the profile
```

To produce outlines of the other teeth one would need the last seven definitions to be reproduced with only slight difference in numbering or tooth type. That can be achieved by means of the Action function `A_repl` that replaces `"?n"` (where `n` is an integer) in a list of strings by the element of the replacement list corresponding in position to that integer. For example,

```
tooth_a =["llreal: ?1?2typ
          "lreal : ?11?2pos
          "frame : ?1?2ins ?1?2dsp
          "?1?2pos = [Rx*cos(ang1), Ry*sin(ang1)]",
          "?1?2typ = rotobj(?3, rt-ang1, 1)"'
          "?1?2ins = object(?1?2typ, ?1?2pos, scalea)",
          "?1?2dsp = wireframe(?1?2ins,?4)" ]
LL2  = A_repl(["LL", "3", "canine", "tooth"], tooth_a)
```

uses `A_repl` to replace each `?1` by `"LL"`, `?2` by `"3"`, `?3` by `"canine"` and `?4` by `"artificial"` to produce the following as a list of strings that are actioned as definitions in the usual way.

```
llreal: LL3typ
lreal : LL3pos
frame : LL3ins LR1dsp

LL3pos = [Rx*cos(ang1), Ry*sin(ang1)]
LL3typ = rotobj(canine, rt-ang1, Z)
LL3ins = object(LL3typ, LL3pos, scalea)
LL3dsp = wireframe(LL3ins,tooth)
```

Using that technique all 40 or more teeth may be instantiated from a single definition set, of which an extract follows (for the upper right set).

```
toothp = if ScM=0.7 then "" else tooth
UR1 = A_repl(["UR","1","incisor","tooth"],tooth_a)
UR2 = A_repl(["UR","2","incisor","tooth"],tooth_a)
```

```
UR3 = A_repl(["UR","3","canine", "tooth"],tooth_a)
UR4 = A_repl(["UR","4","molar",  "tooth"],tooth_a)
UR5 = A_repl(["UR","5","molar",  "tooth"],tooth_a)
UR6 = A_repl(["UR","6","molar",  "tooth"],tooth_a)
UR7 = A_repl(["UR","7","wisdom","toothp"],tooth_a)
UR8 = A_repl(["UR","8","wisdom","toothp"],tooth_a)
```

The (cunning!) `if` definition for `toothp` enables the value of `toothp` to be set to `nil` if a child' s mouth is required. In that case the wisdom teeth are missing and the mouth display is rearranged to reflect fewer teeth. The value of `tooth` reflects whether the tooth is artificial or natural. Similarly if `mouth = female` then the mouth shape is scaled by 0.9, whilst keeping the same number of teeth. The appropriate definitions are as follows.

```
ScaleA = if ScM=0.7 then [0.9,0.9]            ; scale Rxy
                     else [ScM,ScM]
Nteeth = if ScM=0.7 then 6.0 else 8.0         ; child has no wisdom teeth
natural  = "cpline"                           ; tooth is shown as a polyline
artificial = "fill"                           ; artificial tooth is shown hatched
tooth    = natural                            ; define current tooth type
```

Amendments to the position are achieved by having tooth position to be a function of `Nteeth`, the number of teeth per quarter. Just as the definition `tooth_a` is the archetypal tooth, similar definition strings may be constructed for archetypal pads, hooks and so on.

To produce the denture plate an abstract plate may be constructed around the design specification. The tasks of the denture are to replace particular teeth, transfer load to adjacent natural teeth, cross the mouth roof or go around the lower set profile and so on. The requirement for positioning is partly defined by the position of the teeth to be replaced. The intersection of the adjacent natural teeth and the nearest artificial tooth may be determined from the centres of those same teeth on the ellipse defining the mouth shape. Thus from a natural input of the names of teeth that need replacing a general shape of the plate can be generated and a sample plate displayed.

The gap between the teeth is determined as the linearly interpolated value between the adjacent natural and the artificial teeth. This gives a positional error as the position is on an ellipse rather than a line, but that will be small. The archetype definition is `t1` as below

```
t1="?1?2?3pl = midpt([?1?2pos, ?1?3pos])"
```

For a plate covering UR3 and 4 we might have definitions that generate the four main points on the edge of the plate as follows

```
mkg12 = A_repl(["UR","2","3"], t1)
mkg34 = A_repl(["UR","4","5"], t1)
mkg23 = A_repl(["UL","2","3"], t1)
mkg67 = A_repl(["UL","4","5"], t1)
```

giving an output such as, for the first definition,

```
UR23 = midpt[UR2pos, UR3pos]
```

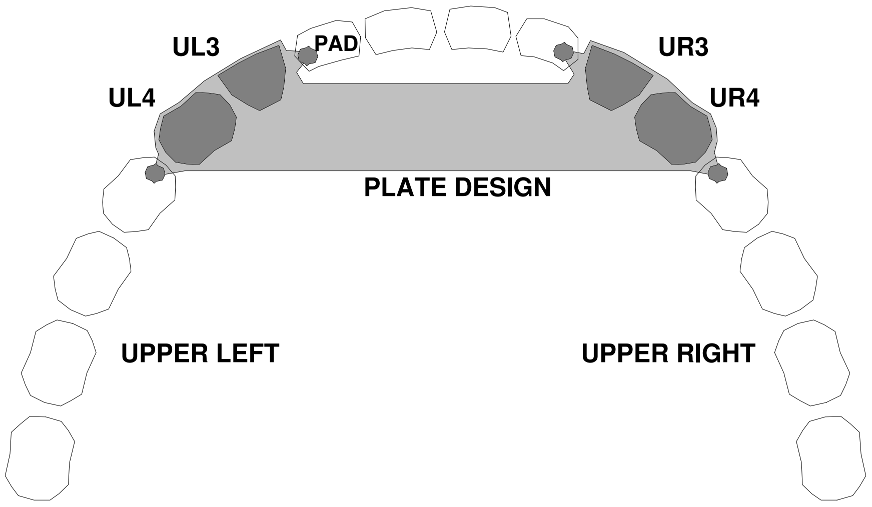


Fig. 7.10 Output of Plate Design defined on four teeth names

Other points demarcating the plate would be generated so as to follow the edges of the missing teeth, obtained from their notional coordinates from the tooth archetypes. The number of points on the plate is as many as required but experiment shows that about seven points are needed for each tooth covered by the plate. A definition to that effect will enable the right number of points to be generated. The final plate design uses the AutoCAD command "`Offset`" that takes a polyline as input and produces another polyline offset to one side or other as requested. A sample plate design is shown in *fig 7.10*. It may be modified to produce the desired shape by appropriate re-definition of the points that make up the definition of `plate`. The pads that enable the plate to be attached easily in the mouth and transfer the load are positioned in standard positions on adjacent natural teeth and assume that the natural teeth can take the extra load.

To make the identification of the tooth positions easier it is best to use labels in the definitions that correspond to the tooth positions, and also to number the label points making up the tooth form in a logical manner to enable the designer to pick off the right points. That process is aided by the fact that dental practice has names for each side and feature of the teeth.

The addition of help and constraint guided functions is an exercise in educational software explored in a later chapter.

Action functions lead not only to an economy of programming but also to the wholesale changing of definitions by a single definition. It is these functions that enable the implementation of agent-oriented programs with the power to make changes in scripts.