ISI
Cairo

A Constraint-Oriented Programming Model with Application to Statistical Graphics.

R.W. Oldford *
Department of Statistics & Actuarial Science
University of Waterloo
Waterloo, Ontario N2L 3G1
Canada.

1 Introduction.

Computational processes are traditionally thought of as uni-directional. One begins with a collection of inputs, performs some calculations on these inputs, and produces one or more outputs. Here we consider a different computational model where the direction of computation is inferred from constraints imposed on the variables involved. Under this model variables are connected by a relation, or *constraint*, such that once values of some number of variables are known, the values of the remaining are determined by the relation. What the relation is between variable values is the reponsibility of the user; how these values are determined is the responsibility of the program object representing the constraint. Moreover, the constraint is active at all times. That is, as the values of some variables change, the constraint is responsible for updating the values of others appropriately. A software system that allows the user to specify relations of interest, and which then determines the calculation paths as needed, we describe as *constraint-oriented.* In this

*Research supported by grants from the Natural Sciences and Engineering Research Council of Canada.

paper some applications of this constraint-oriented programming methodology to statistical analysis are explored. In particular, we present a software representation for a constraint-oriented methodology and through examples explore a number of constraints that would be useful in a statistical context.

In the next section, a brief introduction to constraint-oriented programming is given. Examples give the basic ideas. Section 3 explores a variety of examples that are of interest to the statistical community. Section 4 is a technical section where the software representation for a general constraint is proposed and some technical issues discussed. Section 5 raises some known concerns.

Throughout the paper the parlance of object-oriented programming will be assumed. For precise definitions and further elaboration on object-oriented programming see the books by Keene [Keene 89] and Steele [Steele 90].

2 Constraint-oriented programming.

The fundamental building blocks of this computational model are constraints. These are small program objects that represent a relationship between variables which must be satisfied by the values the variables obtain. Consequently both constraints and variables are represented as objects which are instances of a class called *constraint*, and of a class called *variable*, respectively. Minimally, the class variable has a slot (or field) called *value*, say, where its current value is stored, and a slot called *constraints* which contains the list of constraints in which it participates. Similarly, a constraint has a slot called *constrained-objects* which identifies the collection of variables whose values are related according to the constraint. Other attributes and behaviours of these classes, and related subclasses, will be defined as needed in the course of the discussion.

In this section three simple examples are used to introduce the basic elements of constraint-oriented programming. A more complete introduction can be found in Leler [Leler 88].

2.1 A simple two constraint example.

The equation

$$x + y = z \tag{1}$$

specifies a relationship between three variables - x, y, and z. Given the values of any two variables, the value of the third is completely determined. The equation 1 can be represented by four objects: one for each variable x, y, and z, and one for the constraint 1. The constraint object is responsible for setting the value of the third variable given the values of the other two.

The constraint object here is a simple *adder* constraint. Amongst other things it contains one function for each variable which will set the value of that variable given the values of the other two. Consequently, the user need only specify the relationship between x, y, and z as an instance of the adder constraint class with x and y as addends and z as their sum. Then once any two of the three variables are given values the instantiated adder object sets the value of the third variable by calling the appropriate function.

Since each variable is represented as a data structure, we can easily imagine that any variable could participate in more than one constraint. For example,

$$u * v = x \tag{2}$$

is another constraint, say a *multiplier constraint*, involving x as well as two new variables u and v. Now given the values of any three independent variables (e.g. u, x, and z; not x, y, and z), the values of the remaining two are determined by the operation of the two constraints together (one for each of equations 1 and 2). If many constraints relate a number of variables then the collection of constraints and variables form a *constraint network.*

The network in this example is shown in Figure 1 where each constraint is a black-box calculator and each variable is a circle connected to the appropriate constraints. When the value of a variable is set in the network, it informs all constraints in which it participates that it has a new value to be processed. Each of these constraints updates the values of the variables they constrain. These variables in turn inform the other constraints in which they are involved, and in turn these constraints set yet other variables, and so on. This continues until the effect of the original variable set has propagated as far as possible through the network.

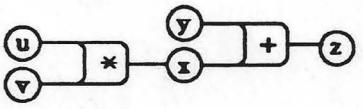

Figure 1: A simple constraint network.

To illustrate, suppose that in the example of Figure 1 we begin with no variables having values. Local propagation proceeds as follows. Setting $y = 3$ causes the adder constraint to process this information. Because neither of x or z have values at this point the adder constraint can set no further values. Now suppose the user sets $u = 5$. Then the multiplier constraint is informed and it attempts to process the new value of u. As before, there is insufficient information to determine the value of either v or x and, because u is not involved in any other constraint, the network propagation stops. Not until the user gives the value of a third variable, say z, is the network solved. Suppose then that the user assigns $z = 10$. The adder constraint is informed of this new value from z and sets $x = z - y = 7$. Since x now has a value, it informs all constraints in which it is involved *except* the one which set its value. Hence, the multiplier constraint is now informed of a value to process from x and it sets the value of $v = 1.4$. The variable v does not participate in any further constraints and the propagation ends. All constraints are satisfied and the network is solved.

2.2 Temperature conversion

Consider a classic example in the constraint-programming literature. Suppose we have a single equation that relates the values of two variable quantities C and F representing the temperature measured in degrees Celsius and degrees Fahrenheit, respectively:

$$9 * C = 5 * (F - 32). \tag{3}$$

For any given value of C we can solve this equation for the value of $F = 1.8 * C + 32$ and for any given value of F we can solve this equation for the value of $C = 5 * (F - 32)/9$. Consequently this relationship could be expressed as a single constraint. However, we are not likely going to want

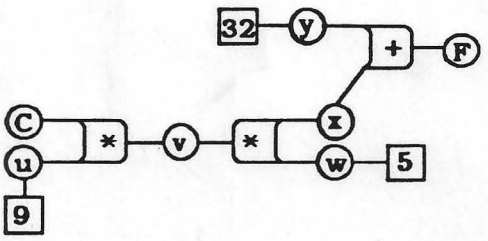

Figure 2: A Celsius-Fahrenheit converter.

more than a single realization of this relationship so defining a new class of constraint to model it seems more than it merits. Instead, by using the more primitive constraints of the previous example we can represent the relation 3 by the constraint network shown in Figure 2.

A new kind of constraint is introduced in Figure 2. The square boxes represent instances of a *constant* constraint – the values in these boxes cannot be changed and are passed on to any variable attached to them. Further, it was necessary to introduce five new *internal* variables (u through y) to hold the results of intermediate calculations. These are of no intrinsic interest to the user. As with the previous example, a value of either variable, F or C, will propagate through the network to update the value of the other variable, C or F as appropriate.

One of the most attractive features of constraint-oriented programming is the ability to add new constraints to an existing network *with minimal fuss*. The user of the Celsius-Fahrenheit converter need only know that the two variables, F and C , are constrained to obey the relationship between degrees Fahrenheit and degrees Celsius – how this is achieved is immaterial. To express temperatures in Kelvin degrees, he or she need only introduce those constraints which relate Kelvin degrees, K say, to Celsius degrees, C (as in $C = K+273.15$). This could be achieved by a *single program statement* like

make-adder(K, make-constant(273.15), C).

Figure 3 highlights the addition this single statement makes to the constraint network. Because the variable C now participates in two constraints in the redefined network, the user is now able to express a Kelvin measurement on the Celsius *and* on the Fahrenheit scale.

This interactive incremental definition of constraint networks makes them very attractive for exploratory analysis. The user need not be concerned with

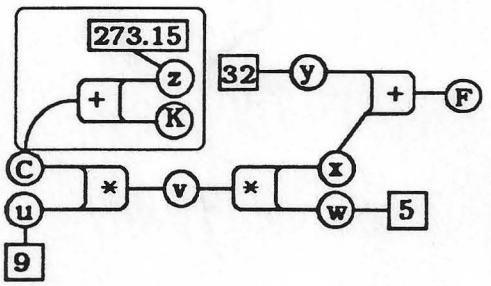

Figure 3: A Kelvin-Celsius-Fahrenheit converter.

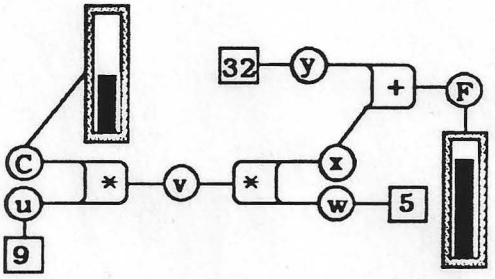

Figure 4: Temperature conversion with thermometer displays.

the action, or even the definition, of the complete network. Rather, interest will focus on small pieces of the network at any given time.

In the present example, one is probably interested only in how the measurement of temperature on one scale appears on another scale. How these changes are effected is of no interest. A familiar means of displaying this information is to have a simple bar chart behave as a thermometer would on each of the different scales. For the Celsius-Fahrenheit converter this is shown in Figure 4. Each thermometer is a graphic whose bar (mercury) height is proportional to the reading on the corresponding variable C or F. As the Figure indicates, this can be implemented as a *monitor* constraint. Whenever the value of C changes then this information is passed on to the thermometer since it is a constraint like any other. Indeed, the thermometers might also accept input. That is, the user could point to the graphic on the display and move the bar of the thermometer up or down. This would change the value of C which would in turn be propagated to the other thermometer as well. (Clearly, two identical true thermometers would show the same height whatever the scale. The difference here is that each thermometer could be

displaying a different range of possible results.)

Because the thermometer graphic is itself a constraint the user could in fact probe the value of any variable with it. Regardless of how complex the underlying constraint network, the probe on C would be installed with a simple command like

bar-monitor(C).

3 Applications to statistical systems.

The potential applications of this programming methodology to statistical analysis systems is vast and largely unexplored. In this section some of the more obvious uses are presented.

3.1 Statistical Graphics.

The earliest constraint-oriented systems were developed to display interactive graphics [Suth 63,Borning 81,Gosling 83,BornDuis 86], [1] so it should come as no surprise that constraints would have a variety of uses in a statistical graphics system. In this section we survey some of these uses.

Throughout this section we will assume that the model for the statistical graphics involved is a hierarchical one where all elements of the display are addressable as in the 1988 model by Hurley and Oldford [HurOld 88a, HurOld 88b,HurOld 91]. This allows us to constrain any piece of the graphic.

3.1.1 Graphics Probes.

Perhaps the most obvious collection of useful graphical constraints are those which probe the data and display their values in a graphic. As suggested in the temperature conversion example of Section 2, these could be graphics which both display the data and allow the data to be changed as a result of changes in the display. It is easy to imagine using these probes to interact with parameters of any program object – tuning constants, parameter estimates, iteration values, convergence tolerances, and so on.

[1] Brief reviews of these and other constraint systems can be found in Leler's 1988 book [Leler 88].

The underlying constraint, a *graphics-probe*, is relatively simple: the value of a variable is constrained to equal the value of some variable in the graphic – which variable depends on the graphic involved. Whenever this variable changes, the graphic is told by the constraint to redraw itself with the new information. If the graphic variable of interest is not identifiable as a data structure then the constraint is a little more complicated (but not much) and may require more specializations for different graphics.

The range of possible graphics is immense but the number of distinct archetypes is few (despite lawyers' arguments to the contrary). A minimal collection for our purposes would include *text-boxes*, *slider-bars*, *bar-monitors*, and *needle-meters* or *dials*. Vector-valued data can be monitored as a single unit by displaying an array of any one of these graphics. This would require only a slightly more complex version of the simple graphics-probe constraint.

Peculiar to statistical analysis are graphics-probes that display some statistical feature of the data. Many of these will be one-way probes – that is, the graphic will operate as a monitor of the data rather than as an input device to effect changes in the data. Some examples are histograms and other density plots, index-plots, qqplots, and fitted curves and lines. For instance, suppose that the data being monitored are residuals from some fit, then we would like to see the change in these various plots as the some aspect of the fit was changed. Other graphics-probes might be bi-directional as in the display of a fitted line where moving the line on the display changes the estimates of its slope and intercept (and possibly attributes of its fit to the data).

On occasion the probe will be monitoring changes over time: either time defined by the user repeatedly changing variable values in the network, or as defined by the value of a variable which increases with each "clock tick". Monitoring a simulation as it proceeds [Borning 81] and animating the behaviour of an algorithm [BornDuis 86] are two examples of the latter notion of time. In either case, it is sometimes of interest to record the entire record of the changes over time. The graphic would then display the entire trajectory of the data being monitored; each new value would *add* information to the display rather than replace it. A classic example is the simple time-series plot where a new line segment is added to the right-most end of the plot as the data value is updated by the network.

The simple graphics-probe is intended to be used with any graphic capable of redrawing itself given new information. Consequently, any arbitrary glyph

could be displayed and updated – weather-vanes, stars, chernoff faces, classification trees, and so on. It would thus admit to a wide range of application – from pure pedagogic demonstration, to data analysis, to methodological research in Statistics.

3.1.2 Linked Displays.

The first use of constraints in statistical computing was by McDonald in his 1986 "Arizona" system to effect linking of points in two or more graphical displays [McDonald 86]. Here as one point changes colour, points that are linked to it change colour as well (or shape, or size, or whatever). McDonald did this by introducing a simple constraint he called a *leader-follower* constraint. Since each point in the plot represented a single data case, the data case was the "leader" and the various point-symbols representing that case were the "followers". Attributes like colour were defined as part of the case so that whenever the case changed its colour all the point-symbols "followed" in suit redrawing themselves with the new colour.

In the Views system of Hurley and Oldford [HurOld 88a,HurOld 88b, HurOld 91] no such drawing information was attached to the case. It resided only on the point-symbol object which was a "view" of that case. Individual point-symbols could appear in more than one plot so that linking was achieved because it was the *same* point-symbol in each plot. However, on occasion two (or more) plots would be constructed that displayed different variables of the same cases but which did not share the same point-symbols. Consequently the plots were not linked. To link them, each pair of point-symbols that displayed the same case would need to be merged into a single point-symbol. Once linked, if it was decided that the linking should be broken then in one of the two plots the point-symbols would need to be replaced by new point-symbols.

As in the Arizona system, in Views we could constrain the point-symbols that share the same case. However, having display attributes like colour on the case unnecessarily clutters the case object and muddies the distinction between the case object and its display. To maintain this distinction, we forego placing display information on data objects and instead constrain only the point-symbols involved. Any constraint on the point-symbols is now "leader-less" so that a leader-follower constraint becomes inoperative.

A more appropriate constraint is one we call a *broadcast -constraint.* In its simplest form, it connects an arbitrary number of variables and ensures that each has the same value. In particular, the value of the last variable to alert the constraint is broadcast to all other variables whose the values are changed accordingly. Thus point-symbols connected by a broadcast-constraint all have the same drawing-style.

Often scatterplots are linked to other statistical plots. For example, a bar of a histogram could be linked to a point-symbol for each case included in the count of the histogram bar. Then colouring the histogram bar red would cause all points to which it is linked to be coloured red as well. But what about the other direction – from the point-symbols to the histogram bar? Could we mark some points in a scatterplot with a single colour and watch where they appear in the histogram? Neither a leader-follower nor a broadcast-constraint would achieve this behaviour.

One way around this is to display a histogram, not as a collection of bars but rather, as a collection of bins where point-symbols are piled up to give the count in that bin as in [Stuetzle 87]. Then the display of each point-symbol in the scatterplot could be constrained to match the display of a point-symbol in the histogram (using the views system, they could in fact be the *same* point-symbol). Now colouring the point-symbols in one plot would be reflected in the other. Of course, in the histogram, there will be some sorting of the point-symbols in each bar so that point-symbols of the same colour are together within the bar. Unfortunately this has changed the display of the histogram.

A similar solution, but one more in keeping with traditional display, is for a histogram bar that bins n points together, to be composed of n smaller bars one atop the other, each one corresponding to a single point-symbol in the scatterplot. Again like colours are grouped together within the bar.

An alternative to both that uses fewer data structures (display objects and constraints) is to employ a many to one constraint connecting the n point-symbols to the single bar of the histogram. In particular, the constraint to be employed is one we call a *polling-constraint.* Its application is illustrated in Figure 5.

A polling constraint connects an arbitrary number of variables, say n, whose values are to be polled and the results reported to an $(n+1)$st variable. That is, the values of the n objects are sorted and the number of times each

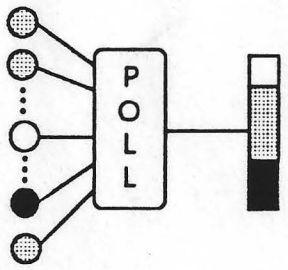

Figure 5: A polling-constraint between point-symbols and a histogram bar.

value appears is counted. Then, the total number n, the different values that were obtained, and the number of times each one was attained, are passed onto the $(n+1)$st variable. Information flowing in the reverse direction forces unaminity from the value of the $(n+1)$st variable to each of the n remaining variables, exactly as would a broadcast-constraint.

In the example of Figure 5, n point-symbols of different shades are connected to a single bar of a histogram. The polling-constraint polls the point-symbols and gives the information on numbers of each shade to the histogram bar which in turn draws itself with fractions of its bar of each shade according to the information it received. If the histogram bar is highlighted directly, then the information flow would be in the reverse direction and the point-symbols which are connected with that bar would all become highlighted.

As different kinds of statistical graphics are linked, no doubt more kinds of constraints will be developed.

3.1.3 Graphical Layout.

More in keeping with the work by Sutherland [Suth 63] and by Borning [Borning 81] are constraints on the layout of the components of a graphic. This early work was concerned with the layout of geometric figures. In the Views system [HurOld 91], constraints are used to maintain the proper relationship between regions of a plot.

For example, suppose we draw a scatterplot having two axes and a point-cloud. In the Views system, these three different graphics are put together in a single plot by assigning each one to a different rectangular region of the display. Figure 6 shows these three regions. They are drawn to fit in their respective regions.

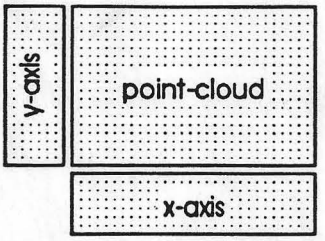

Figure 6: Three regions of a scatterplot.

The three regions are data structures that are constrained to stay aligned. In particular, the vertical coordinates of the y-axis must equal those of the point-cloud region, and the horizontal coordinates of the x-axis must equal those of the point-cloud region. Then as any one of the regions changes – stretches or relocates – the other regions (and the graphics inside them) follow suit. Note that some dimensions are unrestricted so there is nothing preventing the user from moving the y-axis left and right or the horizontal axis up and down.

In the layout design, our objective was to provide enough restrictions that the plot is still sensible yet leave the user enough degrees of freedom to rearrange the display to suit the problem. Constraints provide a simple mechanism for just that.

3.2 Constraints on Calculations.

One can also organize purely calculational results via constraints. For example, consider the simple family of order-preserving power transformations of the variable $x > 0$ given by

$$y = \begin{cases} x^{\lambda} & \text{if } \lambda > 0 \\ ln(x) & \text{if } \lambda = 0 \\ -(x^{\lambda}) & \text{if } \lambda < 0. \end{cases} \tag{4}$$

This equation can be modelled as a single constraint on the three variables x, y, and λ. Note that it is completely reversible – given any two variable values, the constraint can determine the third.

In exploratory analysis one is often interested in the shape of the sample density. By coupling the constraint defined by equation (4) with two

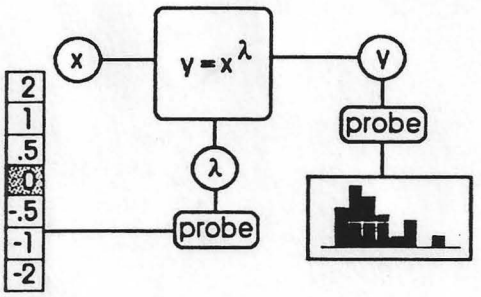

Figure 7: Monitoring the effect of transformation of the data.

graphics-probes, we can interactively investigate the effect of various transformations on the shape of the sample density. In Figure 7, a set of values for λ are arranged in Tukey's ladder of re-expressions as a graphics-probe on λ, and a histogram is attached to as a probe on the transformed data y. The user would see only the ladder and the histogram. When a different value of λ is selected, the information proagates through the network to cause a new histogram to result. Other graphics-probes like a qqplot or an index-plot could be added simply by attaching further probes to the transformed data variable.

With just this simple constraint and a few graphics-probes many interesting possibilities emerge. For example, two power-transform constraints could be hooked in parallel allowing transformations on two data vectors x and y to yield x^{λ_1} and y^{λ_2} which could in turn be constrained to be displayed together in a scatterplot. Ladders on each of λ_1 and λ_2 will allow the user to select values which straighten the plot. Similarly, the transformed data could be used as the input to virtually any statistical procedure.

Consider another example. Suppose we have three variables x, y, and w, and we wish to perform a weighted least-squares fit of y on x using weights w. We could represent this as a single constraint between these variables and two new variables *yhat* and e, being the estimated means and residuals which result from the fit. Now as x, y, and w change, the fitted values and residuals will change. Again using probes of various sorts, the user could easily experiment with new weighting schemes (downweighting, deleting selected points, introducing new variance functions to produce weights, and so on), different transformations of either x or y (simply by placing a power-transform constraint betwen each variable and the weighted-least-squares fit constraint), and even with different values of x or y.

While simple and relatively unimportant in themselves, these examples do illustrate the potential power of this approach. Namely that many small network fragments can be linked together via constraints to provide powerful new interdependent tools for data analysis. Note that we are not suggesting that all algorithms be rewritten in terms of contraints. However, it would be very useful to have algorithm fragments written this way to encourage interactive exploration of both the data and the algorithms.

4 The software model.

There are many ways that one could represent constraints in software (e.g. see [Gosling 83], [SusSteel 80], and [Borning 81] for three different representations). Historically, the choice of representation used by various authors has depended upon the domain of problems they are considering and on the implementation language they use. More generally, Leler shows that all existing *numeric* constraint systems (pre 1988 at least) can be specified in terms of his constraint-programming language called Bertrand [Leler 88]. Bertrand is designed to be a computationally complete constraint-programming language in which to specify any numeric constraint-system. The resulting system will be extensible and its constraint networks solved efficiently. The intention is to allow constraint-programmming languages to be used as general purpose languages without resort to escaping to some underlying host language like LISP or Smalltalk.

While Leler's work is important, the software representation that we adopt for constraints is an extension of Steele's model for constraints [SusSteel 80]. There are a number of reasons for this. First, the Steele's approach is simple to implement. Second, unlike Leler we prefer to work in a host language that already has many attractive programming features, namely Common Lisp [Steele 90], so that "escape" to the implementation language is an attractive feature. Constraints are just another programming approach that seems to simplify certain problems. Consequently, we hope to introduce constraints with a minimal amount of disruption to existing software. Third, as is evident from considering the application of constraint-oriented programming to statistical graphics, it is not clear that we would like to restrict consideration to *numeric* constraints alone. Occasionally the

"values" we will be constraining will be of arbitrary type.

4.1 Steele's model.

Steele's model is the one we used in the examples of the introductory section on constraints. There are two kinds of general data structures: variables and constraints. Variables are used to connect constraints to one another and constraints restrict variables - each maintains a pointer to the other. Each variable has a value (potentially) which can be set by the user or by any constraint in which it participates. Once some variables have known values, the network determines the values of other variables by firing the connected constraints. This means of constraint-satisfaction is described in Section 2 and is known as *local propagation of known states*, or simply *local propagation.*

In addition to setting the value of a variable, Steele also records on the variable its source of information for the value. So if the user set the value, then perhaps the string "user" is attached as the informant of the variable; if the value is set by a constraint, then the identity of the constraint is the informant. This has two principal uses. First, should anyone be interested in the reason some variable has a particular value, an explanation can be constructed by following the informants back through the network one step at a time. Second, it provides a simple mechanism to avoid conflict and to propagate information throught the network. If the variable already has a value, then only the informant of that variable is allowed to change the value of the variable. Should another informant attempt to change the value then an error is flagged. Propagation is carried out by having each variable inform every constraint in which it participates, *except the one that just set its value*, that it has a new value. Each constraint updates its other variables' values as appropriate.

Once enough variable values are set by the user, the remaining are determined by the network. Should new-values be desired for some variables which were set by the user, the values of one or more of the original variables the user set must be retracted. This is most simply done by beginning with one of the original variables, one that we are willing to allow to be changed, and forcing it to lose its value. Like a numerical value, this loss of information is propagated throught the network so that the variable of interest no

longer has a value and can now be set by the user.

4.2 Constraints and kaleidostates.

As can be seen from the examples on statistical graphics, it is not clear that the participants in a constraint should be simple variables which take on a single value. For a point-symbol, we might constrain its colour, shape, and point-size to be the same as another point-symbol but only its colour to be the same as a third point-symbol. This could be implemented as four constraints: one for each of the attributes constrained to be identical between the first two point-symbols and a fourth for the colour constraint between the first and third point-symbols. Alternatively, we might prefer a single constraint between the first two point-symbols that simultaneously constrains all three attributes at once. Other examples are easily constructed.

In our application, neither a variable nor its value are simple. A constraint "variable" is an arbitrary object and its "value", or what is being constrained, is also arbitrary. More generally, what aspect of the object is being constrained and what state that aspect is in depend on both the object *and* the constraint. So for some constraints the colour of a point-symbol is the state of interest, for others it is the entire collection of attributes one might call the drawing-style of the point-symbols, and for still others it could be the data case to which the point-symbol refers.

To emphasize this polymorphic view of the variables of a constraint our variables are called *kaleidostates.* In our implementation, *kaleidostate* is a class that can be mixed into any class as appropriate. It defines the slots and methods that are needed for any object to participate in any constraint. In this way, the user can build and use constraints without needing to understand the details of implementation and satisfaction of constraints.

Each kaleidostate will have a different state depending upon how it is viewed – some constraints will see one state (e.g. colour = black), others will see another state (e.g. case = "John Doe"). This difference is captured by another class of objects we call a *state-lens.* A constraint interacts with a kaleidostate entirely through a state-lens.

The kaleidostate is to be mixed into whatever class of objects are of interest to be constrained. Consequently its structure is kept simple and is set out in Figure 8. The *constraints* slot contains a list of all the constraints

kaleidostate		
	Supers:	None
	constraints:	A list of constraint objects.
	state-lenses:	A list of state-lens objects.

Figure 8: The kaleidostate class.

constraint		
	Supers:	None
	participants:	A list of objects which participate in the constraint.
	state-types:	A list containing the state-type of each participant.

Figure 9: The constraint class.

that the kaleidostate is currently participating in and the *state-lenses* slot is a list containing the state-lens used to view that kaleidostate – these lists are not necessarily of the same length. A number of functions are defined to operate on these two lists.

Similarly, constraints are also rather straight-forward objects as outlined in Figure 9. The rôle of each participant will depend entirely on the constraint. The state of the participant is determined by the state-lens through which it is being viewed by the constraint. The state-type in the same position of the state-types list as the participant in the participants list is used to determine the state-lens.

The class constraint is an abstract class which is never instantiated. It is simply a mixin class representing the most general features of a constraint. For a given application, specialized subclasses of this class like *graphics-probe* or *adder* are constructed and these are instantiated.

4.3 State-lenses.

A state-lens is meant to determine the definition of the state of a kaleidostate. Consequently, it must contain all the information necessary to interact with a kaleidostate in accessing and updating the state. The slots of a state-lens

state-lens		
	Supers:	None
	state-type:	Unique name for the state-type.
	update-info:	Value of last piece of information passed through this lens.
	setter:	Identity of source of update-info.
	update-function:	$\lambda(ks, l, i)$
	get-function:	$\lambda(ks, l)$
	act-on-update-function:	$\lambda(ks, l, s)$

Figure 10: The state-lens class.

are given in Figure 10.

The slot *state-type* is a "hook" to allow the state-lens to be easily referred to by the user and by constraint objects alike. Its value is just a unique but meaningful label to be of use in looking up the appropriate state-lens on the object – no two state-lenses of the same state-type ever appear on the same kaleidostate. The slot *update-info* stores the information that was last passed through this state-lens to update the kaleidostate.

In our model, the information passed from one node in a constraint network to another is not necessarily the value of any state of any kaleidostate in the network. Rather it is information to be processed by kaleidostate and constraint alike. Consequently, there is a function stored on the *update-function* slot of the state-lens that takes three arguments – the kaleidostate ks, the update information i, and the state-lens l. This function is called to update the state of the kaleidostate ks. Similarly, the function which returns the state of the kaleidostate ks as seen through the state-lens l is stored as the value of the slot *get-function*.

Finally, on the slot *act-on-update-function* is stored a function of three variables which is called *after* the update of the state. This is a simple "hook" to allow an action to take place that depends on the kaleidostate ks, the state-lens l, and the setter s. In a *graphics-probe* for example, the act-on-update-function always has the graphic (a kaleidostate) redraw itself. Similarly, objects like *least-squares-fit* which are constrained could be asked to recalculate themselves once the update has taken place.

Kaleidostates are updated by calling a generic function *update-state*. For

slot-lens		
	Supers:	state-lens
	slot-name:	Name of constrained slot.
	reader:	Function used to read the value of the slot.
	writer:	Function used to set the value of the slot.
	update-function:	Uses writer function or slot-value.
	get-function:	Uses reader function or slot-value.

Figure 11: The slot-lens class.

an update to be performed, the pass of information must be considered a legal one. This is determined by the truth value of a predicate function called "legal-pass-p" which has three arguments: the kaleidostate, the informant of the update, and the state-type. For example, in Steele's model this function would evaluate to TRUE if there was no previous setter of the state or if that setter was identical to the present informant. Here we have more flexibility in exploring what are appropriately "legal" passes of information. If the pass is legal, then the update is performed by calling the update-function stored on the state-lens. This done, the action-on-update is called. Finally, all constraints that the kaleidostate participates in, with the exception of the one which just called for the update, are informed that the kaleidostate has a new state. Had the update-info been the same as before, then no update would be performed (legal or not), and no further constraints would be contacted. A similar process is followed for "lose-state".

Often interest lies only in constraining the value of a single slot. A specialization of state-lens is built to accommodate this pattern of use. Figure 11 shows the new slots that are added to the class *slot-lens*. Slots not mentioned in Figure 11 are as defined for state-lens.

5 Other considerations.

The performance of local propagation is simple and fast but it has its limitations. First it may not be fast enough for highly interactive graphics. Luckily,

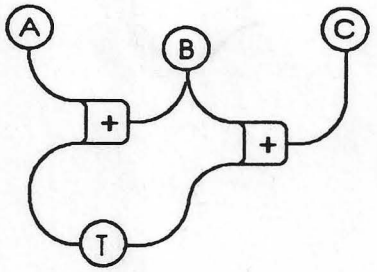

Figure 12: A simple constraint network containing a cycle.

for many of the applications discussed here, *slowly* interactive graphics would be enough. The ease with which one can add an arbitrary graphics link far outstrips the annoyance of waiting seconds (or even minutes!) for the new graphic to be updated. That said, it is still important to be able to compile networks of constraints (without losing its structure so that its topology could be altered at any time and the network recompiled) Principal contributions in this area have been made by the work of Sutherland, Borning, Gosling, and Leler amongst others (again see [Leler 88] for an overview). Second, and worse, local propagation does not always work!

As an example, consider the simple network constraining the values of four variables as displayed in Figure 12. Given the value of A and B the values of C and T can be obtained by local propagation. But if instead we are given the values of A and C, then local propagation cannot determine the value of either B or T. Lest the reader believe this to be a highly artificial example, it is usually cast as A and C being the end-points of a line-segment,AC, and B is the mid-point of that line segment (T is only a temporary variable representing the vector $\vec{AB}$).

The problem is highly related to that of finding efficient constraint satisfaction techniques in order to compile the network. Cycles in the network are at the root of many of the difficulties. Again a number of methods have been suggested for dealing with this and other problems in the topology of the network. These need to be investigated and applied if possible in the present representation. We note only that for many of the applications we envision, the networks appear to be small and acyclic so that the problem is minimal.

6 Concluding remarks.

At best, the present paper is a sketch of some preliminary work that has been carried out on the use of constraints in statistical software. More detail on the software model could be given but that would be beyond the scope of this exploratory presentation. Nevertheless, on the basis of this preliminary investigation constraint-oriented programming seems to hold much promise for applications in statistical analysis.

From the opposite direction, new insight into constraint-oriented language design might be gained from consideration of statistical analysis as an application area. The nature of statistics is such that numerical and non-numerical constraints both come into play. Many recent statistical methods are computationally intensive and are often built from small independent algorithms. Could algorithms, or algorithm fragments, be effectively modelled in a constraint-oriented fashion so that the "new" algorithms are more easily constructed and studied? Can/should the constraint system accommodate calculations with feedback? How, and by how much, can we speed things up in a non-numeric, or partially-numeric, constraint-system?

References

[Borning 81] A. Borning. (1981) The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory. *ACM Trans. on Prog. Lang. and Systems.* 3 pp. 353-387.

[BornDuis 86] A. Borning and R. Duisberg. (1986) Constraint-Based Tools for Building User Interfaces. *ACM Trans. on Graphics.* 5, No. 4, pp. 345-374.

[Gosling 83] J. Gosling (1983) *Algebraic Constraints.* Ph.D. dissertation. Dept. of CS at Carnegie-mellon University. (Also available as a technical report CMU-CS-83-132.) Pittsburgh PA.

[HurOld 88a] C.B. Hurley and R.W. Oldford (1988) *Views: A hierarchical model for statistical graphics.* Technical Report STAT-88-17 from Dept. of Stats. and Act. Sci., University of Waterloo. Waterloo Ontario.

[HurOld 88b] C.B. Hurley and R.W. Oldford (1988) *Views: A hierarchical model for statistical graphics.* Video Technical Report STAT-88-18 (25 mins.) from Dept. of Stats. and Act. Sci., University of Waterloo. Waterloo Ontario.

[HurOld 91] C.B. Hurley and R.W. Oldford (1991) A Software Model for Statistical Graphics. In *Computing and Graphics in Statistics* (edited by A. Bujas and P. Tukey), Volume 36 in IMA Volumes in MAthematics and its Applications. Springer-Verlag New York.

[Keene 89] S.E. Keene. (1989) *Object-Oriented Programming in COMMON LISP: A Programmer's Guide to CLOS.* Addison-Wesley, New York.

[Leler 88] W. Leler (1988) *Constraint Programming Languages: Their Specification and Generation.* Addison-Wesley, New York.

[McDonald 86] J.A. McDonald (1986) *An Outline of Arizona.* presented at the Annual Meetings of the ASA, Chicago IL.

[Steele 80] G.L. Steele, Jr. (1980) *The Definition and Implementation of a Computer Programming Language Based on Constraints.* Ph.D. dissertation. Dept. EECS at M.I.T. (Also available as a technical report MIT-AI TR 595.) Cambridge MA.

[Steele 90] G.L. Steele, Jr. (1990) *COMMON LISP: The Language (2nd Edition).* Digital Press.

[SusSteel 80] G.J. Sussman and G.L. Steele, Jr. (1980) Constraints - A language for expressing almost-hierarchical descriptions. *AI Journal* 14 pp. 1-39.

[Stuetzle 87] W. Stuetzle (1987) Plot Windows. *Journal of the American Statistical Association*, 82(398), pp. 466-475.

[Suth 63] I.E. Sutherland (1963) *Sketchpad: A Man-Machine Graphical Communication System.* Ph.D. disseration, M.I.T., Cambridge MA.

SUMMARY.

Constraint-oriented programming has been a research topic in Computer Science since at least 1963. Advances in computer technology has allowed it to become a more active area in the last decade. In this paper an introduction to constraint-oriented programming is given. A general software model for constraint-oriented programming in an object-oriented system is presented here. The model is an extension of one first proposed by Sussman and Steele [SusSteel 80] and implemented and investigated by Steele [Steele 80].

The key premise in our constraint model is that constraint systems pass *arbitrary pieces of information* around a network, not simply "values". The information can be numeric or non-numeric. Consequently, the state of a constrained variable is determined by the constraint, the variable, and the updated information. This notion is captured in the model by the introduction of a software object which we call a *state-lens.*

Some potential applications in statistical analysis systems are discussed with particular emphasis on the use of constraints in statistical graphics. A number of constraints that are useful in this regard are presented.