SketchFlat, A Constraint-Based Drawing Tool

Jonathan Westhues

jwesthues at cq.cx

June 26, 2007

1 Introduction

SketchFlat is a two-dimensional technical drawing program. It is designed primarily to generate CAM output data, for manufacturing on a laser cutter, waterjet machine, vinyl cutter, 3-axis mill, or other machine tool.

In a typical drawing program, the geometry of lines and curves might be specified by dragging points with a mouse. If it is necessary to specify a point's position exactly, then it might be possible to specify that point's coordinates (x, y), as numbers typed in to a text box.

SketchFlat is different; here a drawing is specified in terms of distances, angles, and geometric constraints. A line segment might be specified as 30 mm long, rotated 15° with respect to the y-axis, and translated such that its two endpoints lie on the x- and y-axes. This is often called 'parametric dimensioning,' and its advantages are obvious. The sketch looks like a dimensioned mechanical drawing; the user can enter the specifications of the part in whatever format corresponds most naturally to the specifications that he is given. If the specifications of the part change, then the CAD tool, and not the user, will recalculate the geometry.

Such a tool is an old idea. Ivan Sutherland's famous Sketchpad introduced the concept, in 1963. By now, many commercial programs are available. Most 3-d CAD packages provide some form of parametric dimensioning. Pro/E uses a full numerical-solution sketcher. Autodesk Inventor provides dimensioning, but as best I can tell, they use a spreadsheet-type (acyclic dependency graph) solver, which isn't nearly as powerful.

Different constraint solver drawing tools might be compared based on several criteria, including:

Choice of geometric primitives. Consider an arc of a circle. We might describe it in terms of its center (x_c, y_c), its radius r, and its beginning and ending angles θ_i and θ_f. This representation translates nicely into parametric equations, which might be useful. On the other hand, we are unlikely to be interested in θ_i and θ_f directly. The user is more likely to wish to constrain, for example, the positions of the endpoints of the arc. This means that our constraint equations will be more complex. They will have to represent not just the desired concept (i.e, the desired position of the arc's endpoints), but additionally the mapping from our choice of primitive to the positions of the endpoints of the arc. They will also be specific to just arcs; we can't reuse our general 'point' constraints.

So we might choose a different set of parameters. We could choose the endpoints of the arc, $P0$ at (x_0, y_0) and $P1$ at (x_1, y_1), and the radius r. This representation is also flawed. If the chord length l_c from $P0$ to $P1$ is greater than the diameter $2r$, then the variables do not describe a circle. If l_c is less than $2r$, then the variables describe two different possible circles; if the arc's endpoints lie on a vertical line, then one circle's center lies to their right, and the other to their left. We could pick between the circles according to the sign of r, for example, but that gets complicated, and we've done nothing to fix the problem when $d > 2r$.

Our chosen primitive should have geometric meaning for any possible value of the variables. This is a generally good idea, but it's particularly important when we are solving for these variables. If certain values of the variables are meaningless, then our constraint equations are very likely to have numerical problems as we approach the impossible configurations.

We might improve our $\{(x_0, y_0), (x_1, y_1), r\}$ representation by replacing the radius r with some better-behaved derived value. Consider, for example, the parameter

$$a = \sqrt{\frac{[(x_0 + x_1)/2 - x_c]^2 - [(y_0 + y_1)/2 - y_c]^2}{(x_0 - x_1)^2 + (y_0 - y_1)^2}}$$

This is the ratio of two lengths: that of the chord, and that of the shortest line from the chord to the center of the circle. For a semi-circle, it is equal to zero. As the angle of the arc goes to zero (or 360) degrees, a goes to negative or positive infinity. The left-or-right ambiguity is naturally resolved by the sign of a. In combination with the endpoints, this parameter a has geometric meaning for any value of a. (This choice of parameter is a semi-popular trick in computational geometry, as far as I know proposed by Sabin in his Ph.D thesis.)

But, the parameter a by itself is not very meaningful. It's unlikely that anyone would be interested in a constraint directly on a. This means that our constraint equations will be complicated.

It might seem attractive to represent the arc in terms of three points: its two endpoints, and its center. This will allow us to reuse all of our general 'point' constraints, and it is always unambiguous. (We still have to describe the direction of the arc, but we might, for example, say that it runs counter-clockwise from $P0$ to $P1$.) The problem is that we gain an extra free variable, which once again means that the user can construct impossible arcs, if the radius from $P0$ to the center is not equal to the radius from $P1$ to the center. This free parameter will also cause trouble for the solver; we will need six constraint equations to describe the arc, when we should need only five.

The simplicity of the constraints will be determined by the representation of the primitives they are constraining. A careful choice of primitives will simplify the form of our constraint equations, and permit us to apply a single class of constraint to many different primtives.

Handling of underconstrained sketches. We might not always have enough constraints to fully describe the geometry of our sketch. In that case we do not have a unique solution; we have an infinite number of solutions, in some number of parameters. We can still solve, but we will have to choose which of the valid solutions we will report. The choice of what is 'reasonable' is subjective; different assumptions will lead to different valid solutions.

Solution algorithms. Each constraint corresponds to an equation, that describes some property that our sketch ought to satisfy. To determine the geometry of the sketch, we must solve these equations. We might attempt to solve these equations in the same way that a spreadsheet does. In concept, we build a dependency graph, that tells us which unknowns we require in order to solve for some other unknown. If this graph is connected and acyclic, then we can solve for the unknowns one by one. This is what a spreadsheet does.

Like a spreadsheet, this scheme breaks if the user creates a circular dependency. This is unfortunate, because it greatly restricts the set of constraints we can use. A circular dependency corresponds to two or more equations that must be solved simultaneously. This is not impossible, but is much more complex. Simple cases might be soluble in closed form, through a combination of symbolic algebra and pre-solved special cases. Numerical solution is also a possibility; here, we have the significant advantage of a good-quality initial guess. The user must draw the sketch before he constrains it, and he is likely to draw it in some reasonable approximation of the desired final solution.

So we are solving a system of nonlinear equations. Our solver should be fast enough that we can run it as the user drags a point with the mouse; this gives us something on the order of thirty milliseconds. Any general-purpose numerical method (Newton's method, for example) can probably be made to work. If possible, then we would like to take advantage of any structure that the system might have, in order to speed up and stabilize the numerical solver.

The system of equations, once appropriate assumptions have been made, will not have an infinite number of solutions, but it might have more than one. Consider, for example, a five-sided polygon, all of whose edges have the same length, and all of whose vertices lie on a circle. A user might draw this if he wanted a regular pentagon; but if the solver produces a five-pointed star, or an equilateral triangle in which one of the edges is traced three times, then it has still satisfied the constraints. In general, we would like to avoid surprising behavior in the face of these multiple solutions.

2 Data Structures

A sketch consists of entities and constraints.

Internally, an entity is reprsented by a 32-bit unique ID, with type `hEntity`. An entity has associated with it some number of points, some number of extra parameters, and some number of (infinitely long) datum lines. For example, a datum point has one point, zero extra parameters, and zero lines. A line segment has two points, zero extra parameters, and zero lines. A circle has one point, one extra parameter (for the radius), and zero lines.

Points are represented by 32-bit unique IDs as well, with type `hPoint`. The ID for the kth point associated with a given entity may be derived from its `hEntity` and k. Parameters are represented by 32-bit unique IDs, with type `hParam`. The ID for the x or y coordinate of a point may be derived from that point's `hEntity`, and the ID for the kth extra parameter associated with some entity may be derived from its `hEntity` and k.

The current values of each parameter are stored in a table. To describe a line segment, for example, we would start with its `hEntity`. From that, we can compute the `hPoint`s of its two endpoints. For each endpoint, we can compute the `hParam` corresponding to its x and y coordinates, and finally look up the values of those parameters. Only the last step (looking up the parameter values) required a table lookup; the rest is just deriving one unique ID from another according to some arithmetic rule.

This provides a pseudo-object-oriented structure: a constraint that accepts an `hPoint` can accept any point, whether that point is an endpoint of a line segment or the center of a circle. But, since these are just unique IDs, the only real data structure is the table of parameters. This is good. We have simplified the solver by putting everything that it's allowed to change in one table.

A constraint is an equation in terms of some number of parameters. For example, a 'horizontal' constraint might force $p_{y0} - p_{y1} = 0$ for some line segment's parameters p_{y0} and p_{y1}.

To solve, we write an equation for each constraint. These equations are written symbolically, in terms of each parameter's `hParam`. This symbolic form will simplify the solver's work greatly, and permit the application of general solution methods to many different types of constraint. Otherwise we might need a point-on-line solver, and a length solver, and a line-tangent-to-cubic solver, and so on; now we just need an equation solver.

3 Geometric Primtiives

The primitives are represented as follows:

Datum point. By that point.

Datum line. By two parameters: an angle p_θ in $[0, \pi)$, and a parameter p_A. The direction of the line is given by

$$(dx, dy) = (\cos p_\theta, \sin p_\theta)$$

and the line travels through the point

$$(x_0, y_0) = (-p_A \sin p_\theta, p_A \cos p_\theta)$$

The solver doesn't know about phase unwrapping, so it might sometimes choose p_θ outside the desired interval.

Line segment. By its endpoints.

Circle. By its center point and radius.

Arc of a circle. By two points A and B that lie on the arc, and by the center C of the circle. This is three points total, or six real unknowns, which is one too many. I therefore generate one hidden constraint, that

$$\text{distance}(A, C) = \text{distance}(B, C)$$

The arc is counterclockwise from A to B; to convert a 10 degree arc into a 350 degree arc, swap the positions of the two on-curve points.

I don't like the idea that entities generate extra constraints, but the center of the circle is useful enough that you almost always want to have it around, and so are the endpoints. I am aware of the usual representations of an arc in computational geometry, and tried several of them, but all of those seemed worse.

The slope of the arc at its endpoint is perpendicular to the radius at the end point. This means that a tangency constraint may be written as an angle (perpendicular) constraint on the line segment connecting the arc's endpoint and center.

The 'arc' is actually drawn as a section of a linear spiral. This forces it to interpolate A and B. This guarantees that no matter what the solver does, a closed curve that includes our circular arc will remain a closed curve.

Cubic spline. In Bezier form. The spline is C1-continuous. A spline consisting of n piecewise cubic segments is described by

$$2 + 2n$$

points: the two (on-curve) endpoints, and two (typically off-curve) control points for each segment. The missing on-curve control points lie at the midpoint of the line segment connecting the two adjacent off-curve control points.

The slope of the curve at either endpoint is equal to the slope of a line from that endpoint to the next control point. This simplifies tangency constraints. Except at the endpoints of the spline, on-curve control points are not needed; we can calculate them, because they lie at the midpoint of the line connecting the two off-curve control points on either side.

Most drawing programs provide G1-continuous cubic splines, but those do not have a nice representation in terms of an integer number of points.

Most 3-d CAD programs have tools for second derivative continuity. This is important when the plane curves will later be used in the definition of 3-d surfaces. I think it is less important when the plane curves are used to make flat objects.

Text in a TrueType font. By two points. The character height is equal to the distance between the two points, and the baseline is perpendicular to the line connecting the points. Lines and quadratics from the glyphs appear in the sketch's table of curves, the same as do the curves from any other primitive.

4 Constraint Equations

Each constraint corresponds to one or more equations. Then equations are written in terms of the parameters that describe the geometric primitives. When a constraint equation holds, the geometric primitive satisfies the specified constraint. The equations are written as follows:

Distance, Length. If the requested length is nonzero, then the difference between the desired and actual length is equal to zero. If the requested length is zero, then this degenerates to a point-coincident constraint. A point-coincident constraint restricts two degrees of freedom, while a nonzero distance constraint restricts only one, and is therefore a special case.

Distance from point to line. Start with the parametric equations of the line, in the form

$$(x(t), y(t)) = (x_0, y_0) + (dx, dy)t$$

The point is at (x_p, y_p). The constraint equation is

$$\frac{dx(y_0 - y_p) - dy(x_0 - x_p)}{\sqrt{dx^2 + dy^2}} - d = 0$$

where d is the desired distance. This distance d is signed; for a vertical line, it corresponds to whether (x_p, y_p) lies to the right or to the left of the line. The sign of d is not displayed on the sketch, but if a negative dimension is entered, then the sign of d is flipped.

If d goes to zero, then this becomes a point-on-line constraint.

Angle, perpendicular, parallel, tangent. All of these are the same constraint. This takes the form of dot product; so if AB is perpendicular to CD, then

$$\frac{(B-A)\cdot(D-C)}{|B-A||C-D|} = 0$$

For angles other than ninety degrees, one of the vectors is first rotated by the required fixed angle. It's necessary to normalize by the length of each vector; otherwise the solver can satisfy this constraint by making one of the lines zero-length, and will do so if no other constraint prevents that.

The dot product does not distinguish between rotations of $+90$ versus -90 degrees. This means that all angles are taken modulo 180 degrees. A $30°$ angle is the same as a $-150°$ angle. When two lines intersect, four angles are formed, two equal to θ, two equal to $180° - \theta$. The form of our constraint is such that we don't care which one we're constraining, but we would like to draw an arc on the sketch, to indicate to the user which angle is intended. We resolve this ambiguity by entering the angles modulo 360 degrees; the extra bit of information is used to choose between θ and $180° - \theta$. The choice between the two equal angles is made according to the position of the dimension label, that the user can drag on the sketch with a mouse.

Coincident points. We are given two points, (p_{0x}, p_{0y}) and (p_{1x}, p_{1y}). If they are coincident, then

$$p_{1y} - p_{0y} = 0 \qquad p_{1x} - p_{0x} = 0$$

Each equation just marks two parameters as equal. We can therefore solve these equations by forward-substitution, and not numerically. This makes them very stable and fast.

Horizontal, vertical. We are given a line segment, with endpoints (p_{0x}, p_{0y}) and (p_{1x}, p_{1y}). For a horizontal line,

$$p_{1y} - p_{0y} = 0$$

and for a vertical line,

$$p_{1x} - p_{0x} = 0$$

This just marks two parameters as equal. We can therefore solve these equations by forward-substitution, and not numerically. This makes them very stable and fast.

Symmetric. Two points are symmetric about a datum line if (a) the line segment connecting the two points is perpendicular to the datum line, and (b) the two points lie at equal perpendicular distances from the datum line. A 'symmetric' constraint is therefore a combination of an angle constraint and a point-line-distance constraint.

5 Assumptions and Consistency Checking

Before we try to solve the system, we must determine whether it is consistent or inconsistent, and whether it is under- or exactly-constrained. For a linear system, these terms are well-defined. For the nonlinear system that we are solving, I will define them as follows:

Under-constrained. The system has an infinite number of solutions that satisfy the constraints, in one or more parameters.

Exactly constrained. The system has a finite number of solutions that satisfy the constraints. It's possible that the system has more than one valid solution; if that is the case, then our initial numerical guess will determine which solution we find.

Consistent. The system has at least one solution.

Inconsistent. The system has no solution. A system that is redundantly constrained (e.g., a point constrained to lie 3 mm from the x-axis, 4 mm from the y-axis, and 5 mm from the origin) is treated as inconsistent.

If the system is inconsistent, then we cannot go any further. We should report this to the user, and perhaps give the user advice on what he might do to make the system consistent again. We are sure that we cannot solve.

If the system is exactly constrained, then we are ready to solve immediately. The constraints fully describe the geometry, so there is no need to make assumptions.

If the system is under-constrained, then we can solve, but we must make some assumptions. If we have m equations, and n unknowns, then we will have to make $n - m$ assumptions. We must be careful to make these assumptions in such a way that the system stays consistent. For example, given the under-constrained system

$$\begin{aligned} p_3 &= 3 \\ p_1 + p_2 + p_3 &= 17 \end{aligned}$$

we will create an exactly constrained system by assuming a value for p_1 or p_2, but an inconsistent system by assuming a value for p_3.

The assumptions could take any form. Ideally, we would like to make assumptions that are orthogonal to the degrees of freedom that are already constrained for each point. If a point's distance to the origin is constrained, for example, then we would like to assume its angle with respect to the origin. That would be rather complex; instead we will choose to assume either a point's x coordinate or its y coordinate, so that the user can drag the points along (locally) vertical or horizontal lines.

Consistency checking is a self-contained process, that is undertaken before we try to solve the system. My algorithms for consistency checking are based around a linearized version of our constraints. The first step is to obtain the system Jacobian. This is easy, because the constraint equations are written symbolically; I can differentiate symbolically, and not worry about numerical problems with numerical derivatives.

The Jacobian is then placed in reduced row echelon form, using Gauss-Jordan elmination, to obtain J_r. If a row of zeros appears in the J_r, then J_r does not have full rank, and the linearized constraints do not form a basis. Our constraints must therefore be inconsistent or redundant.[1]

If we did find a row of zeros, then we report to the user that the system is inconsistent. As a courtesy, we can also determine which constraints the user might wish to remove, in order to make the system consistent again. I do this by brute force: I re-solve the Jacobian with each constraint's equations missing. If I still get a row of zeros, then it would not help to remove that constraint. If the row of zeros goes away, then the user can fix the sketch by removing that constraint, and I report that constraint in a list.

If J_r has full rank, then we must have m rows with a leading 1. The columns in which a leading 1 appears correspond to the variables that will be solved for; all the others will be assumed at their current values. These assumptions correspond to striking out those $n - m$ columns, leaving an m by m identity matrix.

This guarantees that the linearized system is consistent. It does not guarantee that the actual nonlinear system is consistent, if our constraints are very nonlinear and our initial guess is far from the desired solution. It also doesn't guarantee that we've made good assumptions. For highly nonlinear constraints, these assumptions of the form $p_j = p_{j0}$ might always lead to an inconsistent system; consider the constraint

$$p_1^2 + p_2^2 - 100 = 0$$

with $p_1 = 14$ and $p_2 = 20$ initially.

We would like to make the assumptions that are most likely to lead to a consistent subsystem. A given set of equations might (and will usually) permit us to make any of a large number of possible sets of assumptions. I've investigated several different schemes to choose. My favorite of these is based around the sensitivity of the other parameters to the parameter that we are thinking of

[1]It's obvious that a row of zeros means something bad. The assumptions that we make will correspond to crossing out columns, until the matrix is square. This does not get rid of the row of zeros. Our Newton's method is therefore certain to get a singular Jacobian and fail.

assuming. Let us say that we have our Jacobian, of the form

$$\begin{bmatrix} \frac{\partial f_0}{\partial p_0} & \frac{\partial f_0}{\partial p_1} & \cdots & \frac{\partial f_0}{\partial p_a} & \cdots & \frac{\partial f_0}{\partial p_n} \\ \vdots & \vdots & & \vdots & & \vdots \\ \frac{\partial f_m}{\partial p_0} & \frac{\partial f_m}{\partial p_1} & \cdots & \frac{\partial f_m}{\partial p_a} & \cdots & \frac{\partial f_m}{\partial p_n} \end{bmatrix}$$

for some parameter p_a. Initially, assume that the constraints are satisfied. Our parameter p_a then changes slightly, to $p_{a0} + \Delta p_a$. In general, the constraints are no longer satisfied. It might be possible to change the other parameters $\{p_j, j \neq a\}$ in such a way as to satisfy the constraints again; we achieve this if

$$\Delta f_i = \Delta p_0 \frac{\partial f_i}{\partial p_0} + \Delta p_1 \frac{\partial f_i}{\partial p_1} + \cdots + \Delta p_n \frac{\partial f_i}{\partial p_n} = 0$$

for each f_i. We can write a system of linear equations

$$\begin{bmatrix} \frac{\partial f_0}{\partial p_0} & \frac{\partial f_0}{\partial p_1} & \cdots & \frac{\partial f_0}{\partial p_{a-1}} & \frac{\partial f_0}{\partial p_{a+1}} & \cdots & \frac{\partial f_0}{\partial p_n} \\ \vdots & \vdots & & \vdots & \vdots & & \vdots \\ \frac{\partial f_m}{\partial p_0} & \frac{\partial f_m}{\partial p_1} & \cdots & \frac{\partial f_m}{\partial p_{a-1}} & \frac{\partial f_m}{\partial p_{a+1}} & \cdots & \frac{\partial f_m}{\partial p_n} \end{bmatrix} \frac{\mathbf{\Delta p}}{\Delta p_a} = - \begin{bmatrix} \frac{\partial f_0}{\partial p_a} \\ \vdots \\ \frac{\partial f_m}{\partial p_a} \end{bmatrix}$$

If a solution exists, then we can change the other parameters in such a way as to satisfy the constraints. This linear system will typically be underdetermined. (If we are making assumptions, then we have more parameters than unknowns.) We can solve this system in a least squares sense, minimizing the norm of $(\mathbf{\Delta p}/\Delta p_a)$. Let s_a be the norm of our solution. If s_a is large, then a small change in p_a will yield a large change in the other variables. That's bad. If s_a is small, then a large change in p_a will not affect the rest of the sketch very much. This means that p_a is a good candidate for assumption. The situation is more complex, of course, because our s_j values change as the assumptions are made.

This approach seems elegant, but I have not been able to find a fast implementation. It would likely be practical if I cached the assumptions between solver runs, and re-assumed only when the system's 'operating point' (Jacobian) had changed dramatically. That's a lot of work, though.

At the moment, my assumption heuristics are very simple. The Gauss-Jordan solver does only partial pivoting (rows, but not columns); this means that the order in which the unknowns are assigned to columns determines the order of preference with which they will be assumed. I assign the points in the order that they are drawn. If the norm of the column corresponding to a point's x coordinate is greater than the norm of the column corresponding to a point's y coordinate, then I swap the order. Consider a point constrained to lie some fixed distance d from the origin; we should allow the user to drag its y coordinate when it is close to the x-axis, and its x coordinate when it is close to the y-axis. This simple rule will do that.

If a point lies on the line $y = \pm x$, then this simple rule will cause trouble; it switches between the two possibilities erratically, because the point lies exactly on the boundary between them. I therefore add hysteresis. The ratio of the norms must exceed some threshold before I will change from assuming the x coordinate to assuming the y. With that modification, my column-norm rule seems to generate acceptable assumptions for most sketches.

This rule is also susceptible to variations in the way that constraint equations are expressed. Depending on whether a distance constraint is written as

$$\sqrt{(x_a - x_b)^2 + (y_a - y_b)^2} - d = 0$$

or

$$(x_a - x_b)^2 + (y_a - y_b)^2 - d^2 = 0$$

the sensitivities will change by orders of magnitude. This problem could be avoided by requiring, for example, that all constraint equations be of the form $a = 0$, where a has dimensions of length. This is natural for a distance constraint, but less natural for a dot product angle constraint, where we would end up with something like

$$\frac{\mathbf{u} \cdot \mathbf{v}}{\sqrt{|\mathbf{u}||\mathbf{v}|}}$$

It's also possible to normalize by constant factors, so that the dot product would become

$$k\frac{\mathbf{u} \cdot \mathbf{v}}{|\mathbf{u}||\mathbf{v}|}$$

where k is on the order of a typical length in the sketch. This constant might be hard-coded, or it might be derived from the bounding box of the sketch that the user has drawn.

In practice, I haven't implemented anything particularly clever. To mitigate this, I take special care in several places, especially when row-reducing the Jacobian to check consistency and make assumptions. There, I refuse to pivot on an entry in the matrix if its absolute value is less than some fraction of the magnitude of its row. That rule is insensitive to the 'scale' of any particular constraint: if an equation

$$f_i(p_0, \cdots, p_n) = 0$$

is replaced with the equation

$$k_{fi} f_i(p_0, \cdots, p_n) = 0$$

then nothing changes. But, it is sensitive to the scale of a particular unknown. If we replace an unknown p_j with $q_j = k_{pj} p_j$, then our equations (justifiably) seem more or less sensitive to q_j than they did to p_j, according to the value of k_{pj}. This is a problem for angle parameters, which are not necessarily on the same order of magnitude as distances. To fix that, I scale all sensitivities to angles by a constant factor before considering them in the magnitude-vs-magnitude-of-row rule.[2]

[2] This problem would go away if I chose units such that my angles and distances were of the same order of magnitude. I currently use microns and radians, but if I used e.g. meters and radians, then the problem would go away. My fudge factor approach is possibly better, if I wish to choose the fudge factor at run time, based for example on the size of the bounding box of the sketch. Otherwise it's equivalent.

6 Solver

Once we have made our assumptions, we have a system of n equations in n unknowns. The n unknowns are the parameters. Of the equations, m are the constraint equations, and $m - n$ are assumptions of the form

$$p_j = p_{j0}$$

where the right hand side is a constant that the user enters with the mouse.

We would now like to solve. We will do so primarily by Newton's method, but with several optimizations to exploit the structure of our problem. First, the assumption equations are obviously not worth solving numerically; they just correspond to striking out one parameter from the unknowns, and treating it as a constant. This means that we actually have just m unknowns.

In that system of m unknowns, some of the equations are trivial. If an equation of the form

$$p_j - p_k = 0$$

exists, then it's silly for us to solve that numerically; we can just forward-substitute. We replace p_j with p_k wherever it appears elsewhere in the system, and strike out the trivial equation. Trivial equations of this form arise very frequently due to point-coincident constraints. They might also arise indirectly, if an assumption permits us to simplify some initially more complex equation. Since the equations are written in symbolic form, it's easy to make the substitution. Partial derivatives (for the Jacobian used in the Newton's method) will be taken symbolically later; as long as we are careful to take those after all applicable substitutions have been made, they will be correct with no additional effort.

We now have our system of no more than m equations. In concept, we could now solve numerically. The system might still have structure, though. Consider the system

$$\begin{aligned} f_0(p_0, p_1) &= 0 \\ f_1(p_0, p_1) &= 0 \\ f_2(p_0, p_1, p_2, p_3) &= 0 \\ f_3(p_0, p_1, p_2, p_3) &= 0 \end{aligned}$$

This is a system of four equations in four unknowns. We could solve it as such, but that is wasteful. Instead, we would be better off solving f_0 and f_1 simultaneously to obtain p_0 and p_1, and then solving f_2 and f_3 simultaneously to obtain p_2 and p_3, while treating p_0 and p_1 as constants. A practical sketch will often have this nearly-triangular structure, if it is kept more or less exactly constrained as the user draws it.

If we can exploit this nearly-triangular structure, then we will be able to solve much more quickly than we otherwise could. The linear system solutions required in the Newton solver are $O(n^3)$, where n is the number of simultaneous unknowns. If we can split an n-unknown problem into two $n/2$-unknown problems, then we've improved our asymptotic performance by a factor of 4. In practice, the gains are often much greater.[3]

Our Newton solver will search for an independently-soluble subsystem of equations. If it finds one, then it will solve that system independently. I don't have a good method to find independently-soluble subsystems; at the moment, I just search by brute force. The symbolic equations make it easy to count which parameters are referenced. The run time of this search is $O(\binom{m}{k})$, where k is the number of unknowns in a subsystem. It therefore blows up very quickly. If k is large and we still haven't found a subsystem, then it's better to give up the search, and solve the remaining problem all at once. I am now using $k_{max} = 5$.

Depending on our choice of k_{max}, it might take us much longer to find the optimal partition than to solve the partitioned system. In general, though, the optimal partition will not change very much as a function of the parameters. This means that we can save our partition in between solution runs, and try to reuse it. Before searching by brute force, I test all of the subsystems that worked last time. If I can find one that is still independently soluble, then I will use it. If not, then I search.

In practice, the sketch is typically built up incrementally—the user draws some entities, then some constraints, then more entities, then more constraints, and so on. This gives us an opportunity to build up our remembered partitions incrementally, so that our brute force search need work over only a small fraction of the total space.

After solving a subsystem, we can treat the parameters that were solved for as constants. This might make it easier to partition off the next subsystem. For example, consider the equation

$$p_1 p_2 + p_3 - 7 = 0$$

when p_1 is known (i.e., was solved for in a previous subsystem), but p_2 and p_3 are not. In general, this equation is not independently soluble, but if p_1 is known to be zero, then we can solve this, obtaining $p_3 = 7$. Such equations occur frequently when lines are parallel to the coordinate axes, so this is a practically useful special case. The symbolic representation of the equations makes it easy to simplify them symbolically.

When the user types in a new dimension (for example, they change the length of a line from 10 mm to 20 mm), we need to re-solve the sketch to reflect that. If the user has made a large change to the dimension, then our initial guess is no longer very close to the solution. This increases the risk of nonconvergence, or of convergence to an undesired solution. I therefore 'source-step'[4] the dimensions from their initial to their final position, re-solving at each intermediate value of the dimension.

[3]Of course, a system that can be partitioned in this way will probably have a sparse Jacobian. (It doesn't have to, but in our case it does.) A sparse linear system solve is not $O(n^3)$. But this partitioning approach does not seem any more complex than a sparse matrix solver, and it also permits me to simplify the yet-unsolved equations after each subsystem is solved, as described later.

[4]By analogy with SPICE, where a DC voltage or current source is ramped up very slowly from zero.