Enabling Diagnostics in User Interfaces for CAD Applications

by

Alexander Bakst

S.B., Massachusetts Institute of Technology (2008)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2009

Author ....
Department of Electrical Engineering and Computer Science
May 18, 2009

Certified by ....
Daniel Philbrick
Software Development Manager, Autodesk, Inc.
VI-A Company Thesis Supervisor

Certified by ....
Srini Devadas
Associate Department Head, Professor of EECS
M.I.T. Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Enabling Diagnostics in User Interfaces for CAD Applications

by

Alexander Bakst

Submitted to the Department of Electrical Engineering and Computer Science
on May 18, 2009, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

Computer aided design (CAD) applications such as Autodesk Civil 3D allow the user to specify design constraints for a number of common geometries. These applications typically prompt the user for the required constraints and then attempt to find a feasible solution. When there is no feasible solution, however, there is little or no explanation given to the user. Furthermore, given the number of degrees of freedom, it is unreasonable to expect the user to be able to analyze the solution space of the problem in order to correct his input. In this thesis I describe an extension to the geometric solvers in Civil 3D that will enable new user interfaces to assist the user in correcting his input. Furthermore I present several example user interfaces that demonstrate these new capabilities.

VI-A Company Thesis Supervisor: Daniel Philbrick
Title: Software Development Manager, Autodesk, Inc.

M.I.T. Thesis Supervisor: Srini Devadas
Title: Associate Department Head, Professor of EECS

Acknowledgments

The opportunity to write this thesis was the result of the M.I.T. VI-A program and an internship at Autodesk, Inc. A number of people both at Autodesk and M.I.T. were invaluable in their support.

I would like to thank Daniel Philbrick and Bhamadipati Rao for their support and guidance at Autodesk. Thank you to everyone at Autodesk for making me feel welcome during my internship.

I would like to thank Professor Srini Devadas for his recommendations and advice, and for agreeing to be my supervisor on campus.

I would also like to thank Kathy Sullivan for telling me to fill out a VI-A application, despite the fact that she had just informed me it was due in two hours.

Victor, thank you for driving to and from Manchester with me, and for your support throughout.

Finally I would like to thank my family, without whom none of this would have been possible. Hopefully I will finally be able to explain what I was working on with this thesis.

THIS PAGE INTENTIONALLY LEFT BLANK

Contents

List of Figures

Chapter 1

Introduction

Computer aided design (CAD) applications are one of the essential tools in any engineering firm, and since their introduction have simplified the design process considerably. However, there certainly is room for improvement, especially when considering the interaction between the user and the application. One of the problems that has not been completely solved is the over-constraint problem - what to do when the user specifies a design for which there is no real or feasible solution.

The over-constraint problem is of particular interest because of its complexity and importance. Clearly any situation in which the user inputs an invalid design needs to be addressed. However, the problems being solved by CAD applications often have sufficiently many degrees of freedom that it is unreasonable to expect the user to be able to navigate the solution space.

AutoCAD Civil 3D is a CAD application that is used by civil engineers. The software contains a number of tools that solve civil engineering problems. Of particular interest are the tools used to design roads. This is discussed in Chapter 2. In this thesis I discuss a number of modifications to the program that enhance the user's ability to deal with over-constrained designs.

As would be expected, the math module of Civil 3D contain solvers that are specifically designed to solve the geometric problems encountered in civil engineering. In this thesis I outline a method that takes advantage of the existing specialization present in the math module of Civil3D. This "special knowledge" is used to construct a solution that will facilitate the implementation of a user interface that will aid the user in correcting his or her input in the event that the provided constraints are not solvable.

The motivating example is described in Chapter 2. Chapter 3 presents the design and implementation of the changes to the math module of Civil 3D. Chapter 4 describes several user interfaces that have been developed as demonstrative examples. Finally, ideas for further work are discussed in Chapter 5.

Chapter 2

Motivation

2.1 Civil3D and Road Alignment Design

The programs described in this thesis were implemented for Autodesk Civil3D. Civil3D is a CAD tool for civil engineers. A particular aspect of civil engineering is road design. An alignment is a path that describes the shape of the road. In particular, a horizontal alignment describes the path of the road projected onto the ground plane. An illustration of a road alignment is in Figure 2-1. In Civil3D these paths are de-

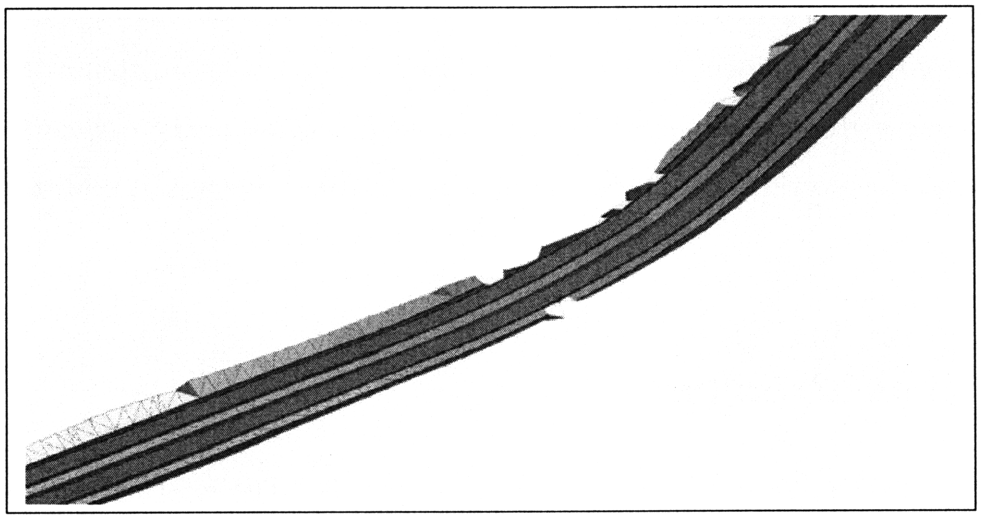


Figure 2-1: Rendered view of a road alignment

scribed by geometric entities called curve groups. A curve group is built from a set of canonical shapes: line segments, circular arcs, and spirals. These are often abbreviated as T (for tangent), C, and S, respectively. Thus an SCS curve group is composed of a spiral which connects to a circular arc segment, and this arc also connects to a second spiral. An illustration of this curve group is given in Figure 2-2

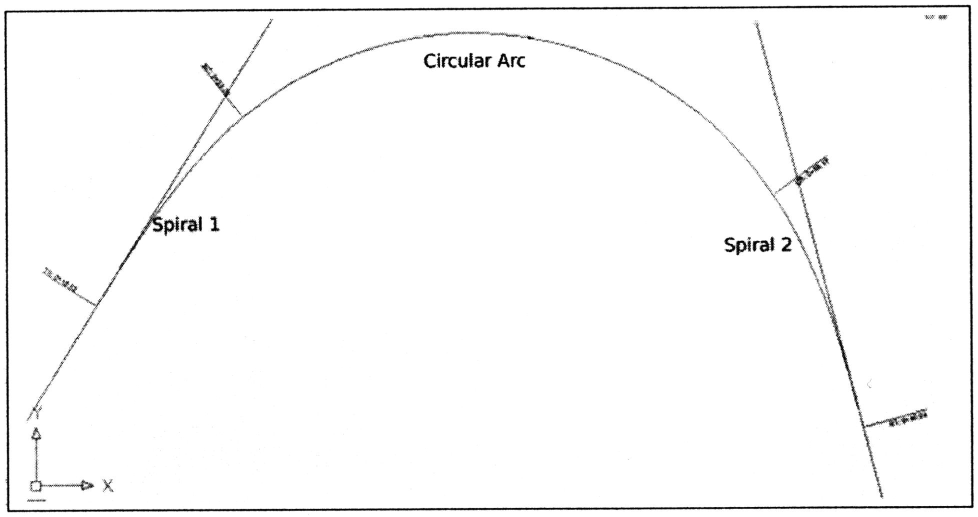


Figure 2-2: An SCS Curve Group in Civil3D

The design of these alignments is accomplished in Civil3D as follows. First the user selects any existing geometry to which a new curve group will be attached. Then the user is prompted for a number of parameters relating to the curve group. These parameters include pass-through points, arc radii, spiral radii, spiral lengths, etc. After all of the parameters have been entered by the user, the program attempts to create a curve group satisfying these constraints. If successful, the new curve group is added to the drawing. Otherwise, nothing is added and the user is told there is no solution.

2.2 The Problem

From this description it is clear what the problem is. The existing interface tells the user nothing about the problem when there is no solution. Because the user is prompted for all of the parameters one at a time in a specific order, it is not obvious where the mistake was made: at least one of all of the given parameters could be wrong. For example, entering a bad parameter in the very first step could already restrict the problem's solution space to such a degree that it is unlikely the user will find a solution.

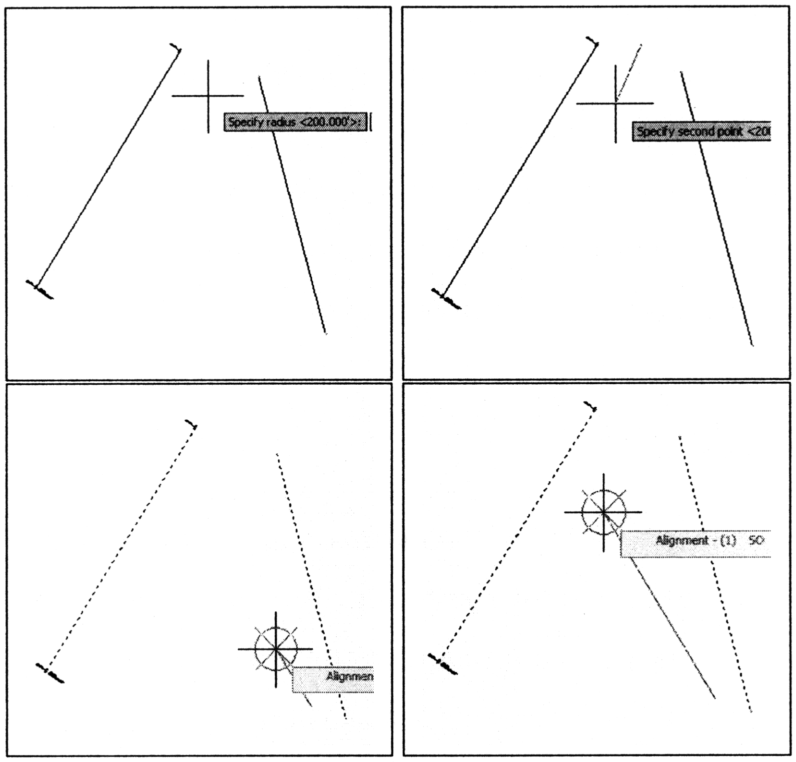


Figure 2-3: Illustration of the user's interaction. Top left: the user is prompted for a radius length. Top right: the mouse is dragged to input the radius length. Bottom left: the program indicates that for the last parameter, the given length produces no solution. Bottom right: the user attempts to modify the last parameter.

An example of the problem is illustrated in Figure 2-3. In this case, the user is designing an SCS curve group, as described earlier. In the top row of images, the user inputs the radius of the circular arc by dragging the mouse. This distance is displayed as a red line. In the bottom row of images, the user inputs the last parameter of the curve group, the length of the outgoing spiral. However, the length the user has selected does not describe a valid solution. This is indicated by means of a red cross. The user then attempts to modify the input by exploring longer spiral lengths, but is met with the same results. There is no cue that indicates if the length should be larger or smaller or if other parameters can be modified to accept the given spiral length. In this example, there is no valid range for the second spiral's length: either the radius or the first spiral's length needs to be modified. In fact, to modify the other parameters, the user must cancel the current operation and enter all of the parameters again, which can be frustrating for some of the more complicated curve groups, which take two to three times as many parameters.

A solution to the problem must present information to the user so that he or she can modify his or her input appropriately in these situations. This involves identifying which parameters must be modified and how they must be modified. Ideally the answer to the latter is provided quantitatively, but this might not be possible in all situations, so at the very least a qualitative answer should be given[1].

2.3 Related Work

To my knowledge, there is no related work that approaches the problem from a perspective similar to that outlined in this thesis. However, the over-constraint problem is much discussed in the literature, and there are a number of different sources from which I have attempted to extract applicable concepts. Most of these sources approach the problem in the context of geometric constraint solvers, which Civil3D is not.

2.3.1 Geometric Constraint Systems

The authors of [9] present a constraint solver that reduces the constraint network incrementally by analyzing the degrees of freedom of the constraint network. This solver is able to solve the problem incrementally, which is a desirable property.

In [2], a system is described which calculates the ranges of parameters in a geometric constraint network. This system decomposes a problem into sub-problems of length and angle constraints between 3-D points. These sub-problems are then solved in order to identify parameter ranges and critical values.

The over-constraint problem is addressed in [5]. The paper describes an algorithm that identifies the reducible edges in a given constraint network. This algorithm

[1]By quantitative I mean that a range of values for the parameters is given; by qualitative I mean that a description of how to modify the parameters (or equivalently, what the error is) is given (e.g., "increase x", "x is too large").

enables the user to navigate an over-constrained problem without a deep knowledge of constraint solving mathematics, which is a desirable property.

The results of [6] and [4] describe systems that solve a system of geometry constraints using what can be considered a constructive technique. The system in Civil3D resembles a constructive a technique to a degree.

Finally, in [11], a method is given that allows a user to explore the solution space of a constraint problem. This is achieved by solving the problem incrementally, and allowing the user to make choices during the process. As will be shown, these concepts are taken into account in the modifications to the solvers in Civil3D.

While the subjects of these papers are in general relevant to the main topics of this thesis, they are too general in order to be efficiently applied to the code base in Civil3D without substantial redesign. Therefore the methods are taken into consideration, but could not be directly implemented.

2.3.2 Other Constraint Systems

In [7], a method is proposed to calculate what is called the feasible solution space of a problem. This method iteratively refines a set of upper and lower bounds for each variable in a constraint system. Their method requires equations of the forms, $v = e$, $v < e$, or $v > e$ for each variable v. Rewriting the constraints in Civil3D, which are often implicit, in this form is infeasible.

The methodology proposed in [1] solves a problem related to that tackled in this thesis: the multiple solution problem. Sometimes for a set of constraints there are multiple solutions, but the user only wants one. This system is relevant for creating richer user interfaces that might gauge the user's intent, and would complement the results described here.

2.3.3 Problem Solving and Visualization Techniques

A method to visualize the solution space of a problem is given in [3]. This is of particular interest because the problems are of such a high dimension that it is difficult for a user to reason about a failed solution or visualize the amount a particular parameter can be varied.

Although the application was very different, the main idea described in [10] was of particular interest in tackling the over constraint problem. The authors describe solving a problem by breaking it into sub-problems and finding partial solutions. The intent was to present information to the user that indicated why a solution was selected.

In [8] the problem solving process is described as an exploration of candidate solutions and the transitions between them. This idea is applied later in order to construct a solver that can work incrementally (i.e., a solver that reports feedback to the user as individual parameters are entered, rather than only after all parameters have been entered).

Chapter 3

Solver Interface

In this section I will describe the changes to the math module of Civil 3D. The math module contains abstractions and routines for geometric entities that are specific to the application. These are abstractions such as 2-D and 3-D points, lines, circular arcs, spirals, etc. The routines include calculating Euclidean distances and numerical methods for root finding.

This module also has routines that solve compound curve types given various parameters. These routines generally operate on geometric abstractions and return new geometric objects on success. These specific routines, which will be referred to as "solvers," are where the additions described in this thesis were made. Specific solvers of interest will be discussed later in this section.

The rest of the section outlines the specific changes that were made that the user interface will rely on. These changes were made in layers. One layer consists mainly of modifications to existing code. Although there are many changes made, the modified solver class still behaves more or less the same way: given a set of constraints, the solver attempts to return a solved geometry. The second layer consists of new objects whose behavior will be useful in the user interface.

The first set of changes is several modifications to the existing solver classes. These include redefining the main solver routines to do more than simply indicate failure when the solver routine fails due to poor constraints. The implications of changing the return conditions are discussed.

The second set of changes is the addition of a new solver class that is dependent on the existing solvers. This solver allows constraints to be specified one by one. This new class then attempts to solve as much of the problem as it can with the given information. In the event that the solver fails, the solver will also attempt to backtrack to a state where less information is known but the solution space is not empty.

3.1 Useful Terminology

There will be several concepts that are central to the presentation and discussion of the work done to the solver interface. It will therefore be useful to give these concepts names to make it easier to refer to them.

Geometry The term geometry will be used to refer to either the constituents of a curve group, or the curve group itself.

Parameter In this context, a parameter can be viewed as a constraint on a geometry. Parameters are given values by a user.

Solution Space Simply the space for which there is a feasible solution for a specific geometry, given a set of parameters, as in [7].

Feasible Solution A feasible solution is a set of parameters that not only satisfy a set of equations, but have values that are meaningful in the context (i.e., all lengths greater than 0), as in [7].

Correctable Parameter Given a geometry with a null solution space, a parameter is correctable if there is some set of ranges of values for that parameter such that the solution space is not null when the parameter takes on values in that set.

Solver A solver in the context of this thesis is a module that, given a set of constraints, returns a specific geometry that can be added to the current drawing.

3.2 Solver Classes

The objects in the Civil 3D code base that are central to the discussion are the solver classes. The math module of the program contains several classes, each containing specialized routines that analytically solve a set of constraints for a given type of compound geometry. For example, given the lengths of four spirals, the radii of two curves, and the angle of a spiral-curve-spiral group, one routine will solve for the start and end points of a geometry that arranges the spirals and arcs in a spiral-curve-spiral-spiral-curve-spiral order.

The format of these solver classes is largely the same. A client interacts with the solver with one routine, `solve`, that takes a set of constraints as an argument. From these constraints, the solver routine constructs a curve group that satisfies them. In the event that the specified constraints are unsolvable, the `solve` routine reports that the operation failed. This failure is reported to the user, and the command that resulted in the call to `solve` terminates.

Here we see that the first step in augmenting the solver modules is changing the type of information that the `solve` routine uses. First I will describe the modifications to the data types used to specify constraints, then the modification to the return type of `solve`, and finally the method used (both at a high level and in this particular implementation) to calculate the necessary information to aid the user.

A diagram showing the relationships between the different modules is shown in Figure 3-1. The important methods and classes that will be discussed are shown, and I have omitted methods that are not relevant to the discussion.

3.3 New Datatypes

3.3.1 Constraints

The `Constraints` type is an abstract datatype that represents a collection of parameters, and replaces a similar, pre-existing datatype. The `Constraints` class enumerates the different parameters that can be specified as input to the different solvers.

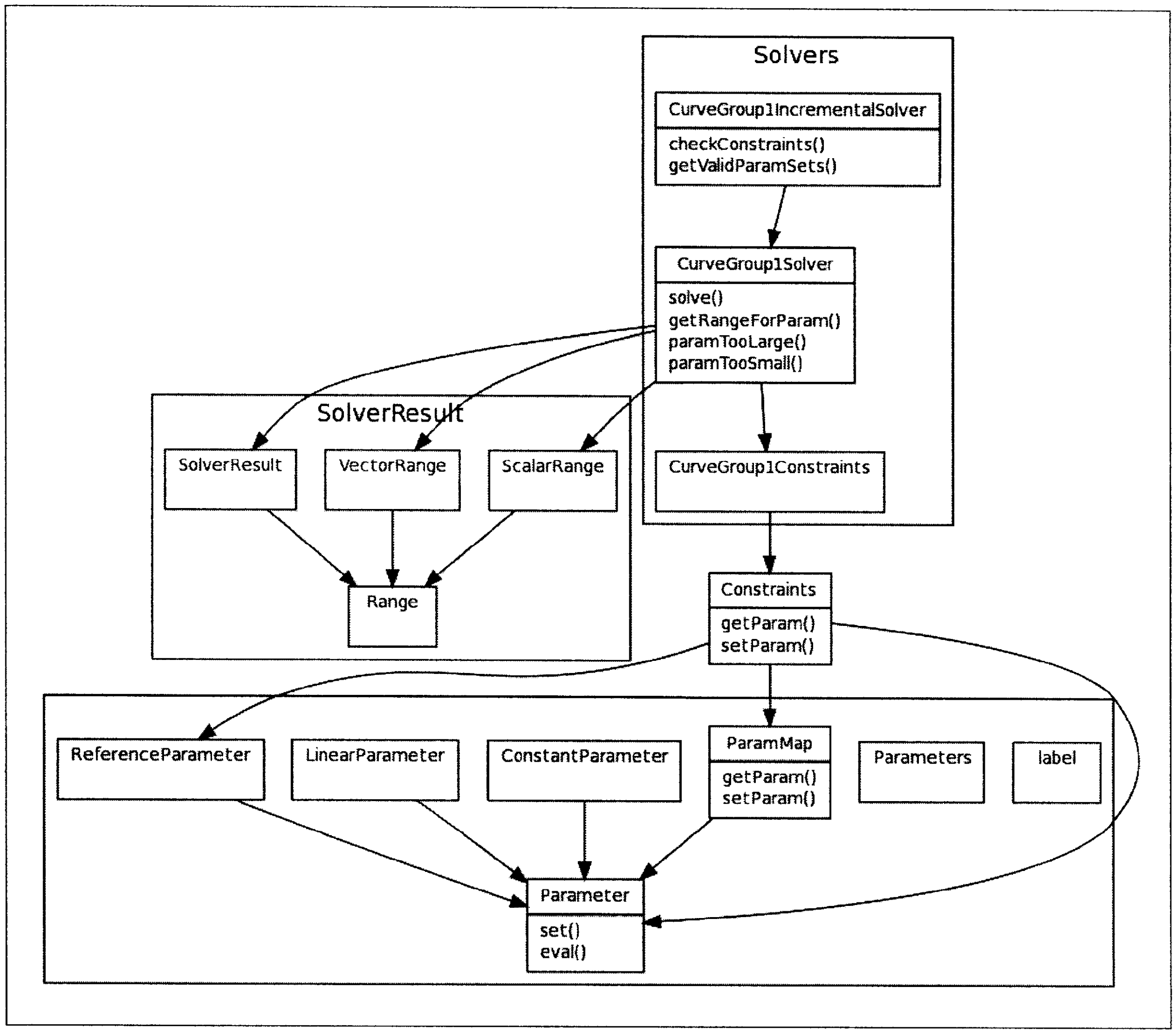


Figure 3-1: Dependency diagram of relevant classes and methods

These parameters include "Spiral 1 Length", "Angle 1", etc. The interface defined by the `Constraints` type provides the behaviors one would expect, such as setting and getting of parameters by name.

3.3.2 Parameter Map

The `Constraints` type is built on top of a mapping environment that maps parameter identifiers to a `Parameter` type. The `Parameter` type is generic with respect to the underlying data type that has meaning in the program (such as `float`, `vector`, etc.). This is useful for two reasons: (1) parameters can be manipulated by name without having to create new accessor/mutator functions for them, and

(2) this enables the specification of arbitrary relationships between parameters. A `Parameter`'s value is set with `set()` and returns its value with `eval()`. The `set()` method makes no guarantees about affecting other `Parameters`. The reason for this will be shown in the description of the concrete `Parameter` types, of which there are two: `ConstantParameter` and `LinearParameter`.

`ConstantParameter` is the type that would be expected in this context. The `ConstantParameter` is initialized with a value. This is the value that will be returned by `eval()`. This value is only changed if a call to `set()` replaces it with a new value.

A `LinearParameter` represents a fixed ratio between two `Parameters`. This ratio is preserved in calls to `set()` on either the `LinearParameter` or the other `Parameter`. To keep the design of the Parameter Map simple, it is up to the programmer to ensure that there are no cycles when creating `LinearParameters`. The `eval()` method looks up the `Parameter` on which it is dependent, calls `eval()` on it, and returns the correct ratio of this value. Because it is possible to have one `LinearParameter` be a ratio of another `LinearParameter`, at some point one of these `Parameters` must be dependent on a `ConstantParameter` in order to evaluate to some value. In practice this is not a difficult requirement to fulfill, as the most complicated solvers require only on the order of 10 parameters.

3.3.3 Parameter References

The `Constraints` datatype returns `ParamRefs` from its `get` methods. A `ParamRef` is a wrapper around the the `Parameters` in the identifier to `Parameter` map. These references have the assignment operator (`operator=()`) and cast operator (`operator()()`) overloaded. As a result, `ParamRef` objects can be used nearly identically to the underlying data type of the represented parameter (`float`, `vector`, etc. as before). The programmer thus does not need to worry about preserving arbitrary relationships between parameters.

3.3.4 SolverResult

The existing interface defines the `solve` method to return either success or failure. On success a geometric entity satisfying the user constraints is also returned. This is of course satisfactory in the case where the constraints define a feasible solution. However, for reasons that have been described, it is clearly unsatisfactory when the constraints do not define a feasible solution. In particular, the user would like to know, in addition to success or failure, (1) why the solver failed and (2) how the constraints can be modified so that the user can modify his or her input to ensure that later attempts to solve the constraints will be successful.

To facilitate the construction of a solution to these problems, I modified the return type of the solver interface to return a simple object that either indicates success and includes a reference to the solved geometry, or indicates failure and includes a number of different types of `Range`. This object is called a `SolverResult`.

3.3.5 Error Codes

The first part of the `SolverResult` is the indication of failure. The `SolverResult` object that is returned from the `solve` method of a particular solver contains a set of error codes. Each of these codes is simply a value from an enumerated data type defined in the solver class from which the object was returned. Each value corresponds to a known cause of failure. For example, in the SCS case, a returned `SolverResult` could contain the codes for the first spiral being too long, and the second radius being too small.

Each error code is a message that is specific to the geometry being solved. The message indicates why the solver failed to produce a solution. In particular, this message is of the form "x too large," where x is a particular design parameter, such as a length. In practice, there are generally several ways to correct the user's input. That is, using the above example, it is not necessarily the case that the user must decrease the length of the spiral *and* increase the size of the radius. It is most likely that either correction will suffice. The main point is that the user is given the complete list of options, since one correction may not be useful given a particular construction, of which the desired solved geometry it but one part. That is, it may be more convenient to increase parameter x_1 in one situation, and to decrease parameter x_2 in another situation.

3.3.6 Ranges

The second part of the `SolverResult` is the `Range` datatype. A `Range` is returned with an identifier for the parameter being corrected for each correctable parameter. The abstract `Range` type, and its derived types, are generic with respect to the geometry, and are thus not dependent on any specific type of solver or geometry. There are derived types for scalar parameters (e.g., lengths in the Civil 3D context) and vector parameters (directions). There is also a correction type that returns the area in which the parameter (such as a point in 2d space) could be located. These are `ScalarRange`, `VectorRange`, and `AreaRange`, respectively.

3.4 Required Modifications to Existing Solvers

The datatypes `Constraints`, `ParamRef`, `SolverResult`, and `Range` have been introduced. Now I will describe how the existing solver classes must be modified. This process is done in two passes. The first pass requires changing the existing code in order to reduce ambiguities. This can be thought of as augmenting the working set of information at each of several key steps. The second pass leverages this new information to create at least one new method that will be the backbone of this thesis. The changes are actually very simple. However, this simplicity, introduced in the right places, yield some very powerful results.

3.4.1 Solver Routines

The first pass is substantially more domain-specific than the second pass. However, it is critical in importance. This step is identifying the points of failure in all of the solver subroutines. An example subroutine might look as it does in Figure 3-2.

```
//Rotate vector1 by theta and return result as vector2
subroutine vector_from_theta(IN vector1, IN theta, OUT vector2):
    vector2 = rotate(vector1, theta)
    value = dot_product(vector1, vector2)
    if value < 0:
        return vector1_vector2_opposite_directions
    return OK
```

Figure 3-2: Example Pre-existing Subroutine

While there is nothing necessarily wrong with this code, the error message is not specific to any of the given parameters. The purpose of this step, therefore, is to replace these error messages (or codes), with codes that are specific to the input parameters. That is, the error messages should indicate whether or not a parameter is too big or too small, for example. Therefore, in this pass the developer replaces the existing error codes or messages with a set of the previously defined `ErrorCodes`. The code in Figure 3-2 should be amended to read as in Figure 3-3.

```
//Rotate vector1 by theta and return result as vector2
subroutine vector_from_theta(IN vector1, IN theta, OUT vector2):
    vector2 = rotate(vector1, theta)
    value = dot_product(vector1, vector2)
    if value < 0:
        return Set(theta_too_large)
    return EmptySet
```

Figure 3-3: Amended Subroutine

It is almost always the case that it is impossible to identify the specific parameter[1] with the bad value, given a set of possible parameters. However, it is sufficient to simply give a "direction" (i.e., larger or smaller) for the possible parameters. This

[1]Certainly this should be the case, as the program has no way of knowing the relative importance of each of the parameters to the user.

information will be enough to guide the rest of the program in determining possible fixes to the given parameters.

At this point, the solver has been modified in such a way that it is already returning much more information than before. However, this is mostly a consequence of domain-specific knowledge about the problem, and is not particularly more useful than the existing program. The rest of the modifications, and indeed the rest of the thesis, demonstrate how this specific knowledge about the problem can be transformed to build an interface that will greatly enhance interaction with the software. In fact, the rest of the modifications need know very little about the actual problem they are solving.

3.4.2 GetRangeForParam

The `GetRangeForParam` routine is the first step in creating a user interface with feedback. `GetRangeForParam` is a routine that takes a parameter identifier as an argument. If possible, the routine returns sets of ranges for that parameter. In particular, it returns a set of `Ranges`.

The idea behind the algorithm is simple. First, the range of possible values for the given parameter are bracketed. This is usually done with information that ensures the solutions will be feasible. For example, an angle might be bracketed to be between 0 and 2π radians. Second, a bisection search is used to find the minimum and maximum values for the specified parameter. This search is guided by error codes that are returned by the `solve` routine. Figure 3-4 contains the pseudo-code for the `GetRangeForParam` algorithm.

Analysis of Algorithm

`GetRangeForParam` leverages the existing `solve` routine in order to solve for a particular parameter. This takes advantage of two facts: (1) the analytical solvers are relatively fast, and (2) when a set of parameters does not define a feasible solution, the solver routines "error out," rather than complete the solution computation. The first fact is important because the solver routine is called over and over again as part of the bisection routine. The second fact is important for a similar reason. Because the computation is immediately halted when it obvious that no solution exists, the bisection search can immediately compute the next value to try.

The `GetRangeForParam` procedure is essentially a bisection algorithm. It is initially called with coarse bounds for the given parameter, and an empty set of `Ranges`. The two while loops then refine the upper and lower bounds for the parameter. At the end of the procedure, these bounds are checked by checking for a valid solution in the computed range. In practice, this single check is good enough to determine whether or not the range is valid. If the range is determined to be good, then the procedure is called recursively to find ranges which are disjoint from the range identified by the current procedure call.

The calls to `paramTooLarge` and `paramTooSmall` simply check the set of `ErrorCodes` to see if there are any `ErrorCodes` that indicate the supplied parameter is too large or too small, respectively. This is most often a very simple check, but is included to account for cases that might require extra computation to determine if the parameter is too small or too large.

The minimum number of iterations required for each of the bisection search is logarithmic[2] in the size of the interval (that is, $iterations > \log_2(\frac{upperBound - lowerBound}{\epsilon})$, for some program specified ϵ).

Clearly this solution works by taking advantage of the domain-specific knowledge about the problem that is present (either explicitly or implicitly) in the program. Perhaps most profound is that a simple transformation on the solver procedures enables the rest of the program to exhibit powerful behaviors.

Of course, the bisection method is not explicitly required. A different numerical algorithm could be used to solve for the desired parameter. The bisection method has the advantage of being simple to implement and reasonably efficient. It is also takes advantage of the information returned by the `ErrorCodes`.

[2]Actually, if the input is considered to be the number $upperBound - lowerBound$, then the bisection searches are bounded by function which is linear in the number of bits of $|upperBound - lowerBound| = \log_2(upperBound - lowerBound)$

```
GetRangeForParam(IN param, IN upperBound, IN lowerBound, IN/OUT ranges):
   iterations = MAX_ITERATIONS
   esMin = esMax = {} // empty sets of ErrorCodes
   searchMax, searchMin = upperBound, lowerBound
   paramMax = upperBound

   while iterations > 0 and searchMin < searchMax:
      setParam(Constraints, param, (searchMin + searchMax)/2)
      esMax = solve(Constraints)
      if paramTooLarge(param, esMax):
         searchMax = eval(Constraints, paramReference)
      else:
         searchMin = eval(Constraints, paramReference)
      iterations = iterations - 1

   searchMax = paramMax = eval(Constraints, paramReference)
   searchMin = lowerBound

   while iterations > 0 and searchMin < searchMax:
      setParam(Constraints, param, (searchMin + searchMax)/2)
      esMin = solve(Constraints)
      if paramTooSmall(param, esMin):
         searchMin = eval(Constraints, paramReference)
      else:
         searchMax = eval(Constraints, paramReference)
      iterations = iterations - 1

   paramMin = eval(Constraints, paramReference)

   if paramMin != paramMax:
      setParam(Constraints, param, (paramMin + paramMax)/2);
      es = solve(Constraints)
      if es == {}
         insert(ranges, ScalarRange(param, paramMin, paramMax))
         if lowerBound < paramMin
            GetRangeForParam(param, lowerBound, paramMin, ranges)
         if paramMax < upperBound
            GetRangeForParam(param, paramMax, upperBound, ranges)
      return es
   else if paramMin == upperBound:
      return esMax
   else:
      return esMin
```

Figure 3-4: Pseudo-code for GetRangeForParam

3.5 Incremental Solver

The final piece that adds significant functionality is the `IncrementalSolver`. This object makes use of information from the modified solver classes in order to provide as much information as possible, and as early as possible. The intent is to prevent failure as soon as it is preventable. The idea is that solving the curve group problem is composed of solving the sub-problems of the constituent curves in such a way that there is always room to define any unsolved curves, similar to the idea described in [10]. An illustration of this idea is in Figure 3-5.

A few assumptions need to be made in order to implement this step. I will present the assumptions I made in my implementation for Civil 3D, justify them, and then show how the underlying method does not rely on these assumptions. I will also show that these *specific* assumptions are not critical to the operation of the program.

The `IncrementalSolver` has one basic method, `CheckConstraints`, that is called by the client. This method takes a parameter identifier and a value as an argument. The method will then calculate feasible ranges for the parameters which have not yet been set by the user. The `IncrementalSolver` class also has a `GetValidParamSets` method that can be used to find the largest subsets of the given constraints for which there is a feasible solution.

3.5.1 Assumptions About the User

The central principle guiding the operation of the `IncrementalSolver` is to calculate as much information as possible at every step. Clearly a unique solution cannot be determined without defining all of the parameters (or restricting the domain of one parameter to one element) for a given curve group. If the essence of the incremental approach is to guide the design, then some assumptions must be made by the program about what the unspecified parameters will look like.

Consider the spiral-curve-spiral group as described in Chapter 2. The user is prompted for three parameters: S_1, r_1, and S_2. The spiral-curve-spiral group is fit between two lines such that the start point of S_1 lies on the first line and the end point

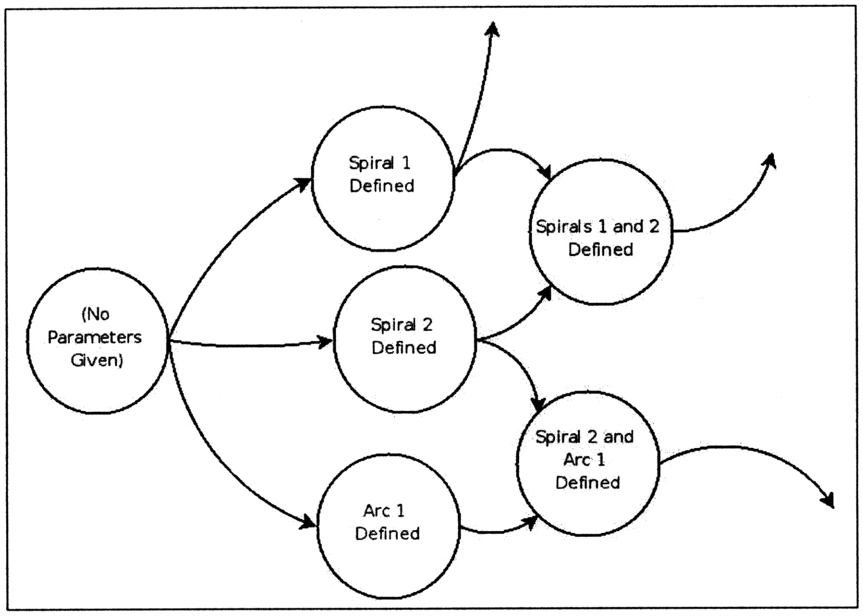


Figure 3-5: Visualization of incrementally exploring solution space. As new parameters are added (edges), it is possible to solve for additional parameters. Each circle represents a state in which enough parameters have been given to define a component of the curve group.

of S_2 lies on the second line. It is generally the case that increasing the parameters "push" the curve group along the two lines until the points no longer lie on them, and that decreasing the parameters results in a similar effect. This is illustrated in Figure 3-6: as the spiral length is increased, the end point of the curve group slides along the rightmost line until it is pushed past the line's endpoint, yielding no solution.

When a parameter isn't given, the assumption is that the parameter is equal to 0. The justification is found in the spiral-curve-spiral example. Assume only r_1 is

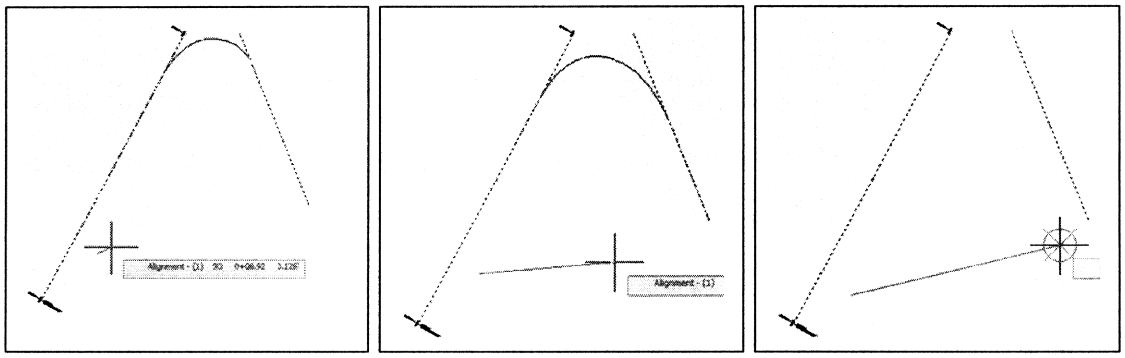


Figure 3-6: Demonstration of how the curve group varies as the length parameter varies

given. In theory, S_1 will have the largest value when S_2 is 0. Therefore, if S_1 has no valid range for a given r_1, then there is no value of S_1 or S_2 for which there is valid solution for the given r_1. At this point the `IncrementalSolver` can determine that there is no valid solution, and this is returned to the user.

However, it is possible that for some problems, the solution space appears to be null, but a solution appears once more parameters are set. This problem can be addressed by taking advantage of common relationships between different parameters. By using the `Parameter` classes, these defaults can be set to either constant values or functions of other parameters. For example, perhaps it is the case that it is a reasonable assumption that $r_1 = r_3$ in the previous example. Then, if r_1 is given, a nominal value for r_3 is immediately given as well. The value for r_3 can then be refined independently of r_1 later. As a secondary benefit, setting default relationships between parameters effectively reduces the search space of the problem.

3.5.2 Incremental Solver Operation

The `IncrementalSolver` class has two main routines that the client uses: `CheckConstraints` and `GetValidParamSets`. The first routine takes a parameter identifier and a value as arguments. Using any previously entered values, the routine then attempts to determine whether or not the feasible solution space is empty. If it concludes that the solution space isn't empty, it also uses the given information to solve for ranges of any unspecified parameters. Otherwise it attempts to backtrack to find the largest subsets of the given parameters for which there is a non-empty solution space. Pseudo-code for this algorithm is given in Figure 3-7.

`GetValidParamSets` uses `CheckConstraints` as a subroutines. The function enumerates the subsets of the currently specified parameters which are one element smaller. It then checks these constraints. If none of these subsets have a valid solution space, then the backtracking algorithm is run again on these subsets until a set of valid constraints is found. Pseudo-code is given in Figure 3-8

The first two steps are actually combined into one step. Determining whether or not the solution space is empty is equivalent to determining valid ranges for unspecified parameters. If no such ranges exist, then the solution space is empty. Otherwise these ranges can be returned.

In Civil 3D, curve groups are composed of several individual curves. An example is the SCS curve group, which is composed of two spirals and one circular arc. Because the shape of each spiral is determined by two parameters, a further complication is added to the problem. If neither the length nor the radius are given for one of the spirals, it doesn't make sense to search over the possible lengths or radii. This is due to a constraint implicit in the problem that the subtended angle of the spiral can not exceed $pi/2$ radians. Since the angle is related to the ratio of the length to the radius, searching over possible lengths with the radius set to 0 is a meaningless exercise. This issue is dealt with by defining the related sets of parameters in the `Solver` class. The `IncrementalSolver` then uses this information to determine which parameters to search for. In the event that there is not enough information to search for any parameters, the `CheckConstraints` routine simply assumes that the solution space is non-null and returns no parameter ranges.

Analysis of CheckConstraints

`CheckConstraints` must call `GetRangeForParam` on at most all of the unspecified parameters. The running time is therefore $O(n)$ where n is the number of unspecified parameters. Given that the current implementation of `GetRangeForParam` which uses the bisection method, this means the running time of `CheckConstraints` is $O(n \log l)$, where l is the maximum length of the geometry[3].

Analysis of GetValidParamSets

`GetValidParamSets` enumerates the power set of the specified parameters in the worst case. Therefore the algorithm makes $O(2^n)$ calls to `CheckConstraints`. The number of calls that `CheckConstraints` will make to `GetRangeForParam` will almost always decrease as the size of the sets decrease, but how the number decreases is largely dependent on the solution, so no analysis is offered. It is important to note

[3]Again, the size of the input l is $\log l$, so the function is linear in the size of this number

```
CheckConstraints(IN constraints , OUT solverResult ):
    push( constraints )
    for param in getUnsetParams():
        setParam( constraints , 0)

    if setParams == requiredParam:
        solverResult = solve()
        if isSolved( solverResult ):
            return solverResult
        es = getErrorStatus( solverResult )

    else:
        solverResult = new SolverResult;
        solvableParams = getSolvableParams(setParams)
        for param in solvableParams:
            ranges = []
            if not isSet(param):
                es = GetRangeForParam(param, ranges)
                addRangesToResult( ranges , solverResult )

    if not hasRanges( solverResult ):
        setErrors( solverResult , es)

    pop( constraints )
    return solverResult
%\end{verbatim}
%\makebox[\textwidth]{\hrulefill}
```

Figure 3-7: Pseudo-code for CheckConstraints

```
GetValidParamSets(IN setParams, OUT paramResultList):
    paramResultList = []
    paramSetsToTry = {}

    for param in setParams:
        insert(paramSetsToTry, remove(setParams, param))

    while not empty(paramSetsToTry):
        nextParamsToTry = {}
        for paramSet in paramSetsToTry:
            result = CheckConstraints(paramSet)
            if hasRange(result):
                add(paramResultList, (paramSet, result))
            else if size(paramSet) > 1:
                for param in paramSet:
                    insert(nextParamSetsToTry, remove(paramSet, param))

        if empty(validParamSets):
            paramSetsToTry = nextParamSetsToTry
        else
            return paramResultList
```

Figure 3-8: Pseudo-code for CheckConstraints

that although the running time is given as exponentially related to the number of parameters, in practice there are generally on the order of 10 parameters; furthermore the solver rarely backtracks over all of the parameter subsets.

3.6 Summary

I have shown how the existing code has been modified to satisfy the goal of enabling user interface diagnostics. I added a new return type to the existing solver classes, and modified the solver routines to return important information in the event of failure. This information is also used by the solver in order to determine valid ranges for the different solver parameters. I modified the constraint datatype in order to capture the relationships between different parameters. Finally, I designed a new solver class that breaks the problem into several sub-problems, and incrementally solves each of these. This solver class also backtracks from invalid states to states which have a feasible solution space.

In the existing code, different solver classes are associated with their own constraint type. The new design preserves this relationship by not radically modifying the relationship between solvers and constraints.

Chapter 4

User Interfaces

In this section I present several examples of diagnostics that can be presented to the user as a result of the work in Chapter 3. The first interface demonstrates ideas for different types of visual feedback. The example geometry consists of a straight line segment and a circular arc. The second interface illustrates the feedback mechanism for over-constrained solutions on a problem that is slightly more complex, and finally the third interface demonstrates the use of incremental feedback on an even more involved problem. The third interface also demonstrates backtracking from sets of parameters which have no solution.

Pseudo-code describing the basic implementation of the user interfaces is given in Figure 4-1. This is the basic behavior of the user interfaces. For different types of feedback (e.g., the first example in this chapter), the loop is modified accordingly. User input drives the solver. At each step either a solution is found, ranges are found for the parameters, error messages are displayed, or the system is underconstrained (i.e., not enough parameters have been entered).

4.1 Line-Arc Through Point

The example geometry consists of a line segment and a circular arc. The arc is constrained to be (1) tangent to the line segment, (2) of a user-specified radius r, and (3) to pass through a user-specified point p. In Figure 4-2 (a), the location of

```
//Invoked on user input
SolutionLoop():
   read_changes()
   result = solver.solve(constraints)

   if isSolved(result):
      display "Got␣a␣solution."
   else if hasRanges(result):
      for each range in result.correctedParameters:
         display parameter.name
         displayRange(range)
   else if hasErrors(result):
      displayErrorMessages(result)

   updateDisplay()
```

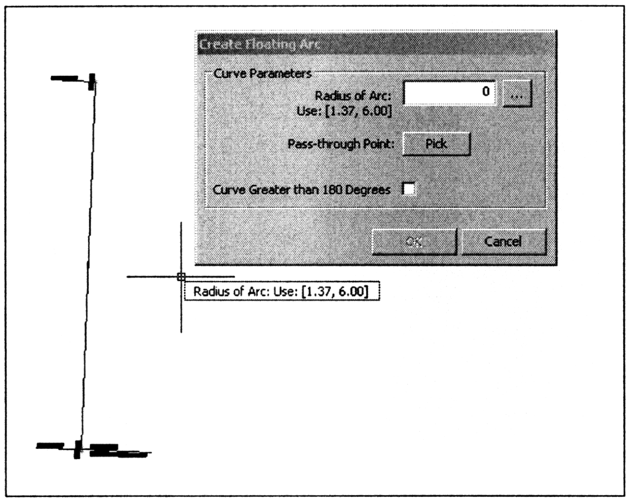
Figure 4-1: Pseudo code describing the basic interface implementation

p is given using the cross-hairs. When the user clicks the mouse on this location, a tool-tip displays the range of values for r (labeled "Radius of Arc") for which there is a solution given p. In this case the valid values for r are between 1.37 and 6.00.

In Figure 4-2 (b), r is set to 1. The solver then calculates the valid range for p. The application displays this range as a shaded region (shown in black). If p is placed in the black region, then it is possible to add a valid geometry satisfying p and r. Attempting to place the point outside of this shaded region will cause the program to accept p, and calculate a new range for r which will be presented to the user as in Figure 4-2 (a).

4.2 Line-SCS-Line

In this example, the alignment geometry consists of two lines, and a curve group that is fit between the two lines, L_1 and L_2. This curve consists of a spiral, S_1, a circular arc C, and another spiral S_2. S_1 is constrained such that the start point of S_1 lies on L_1, and that the tangent of S_1 is parallel with L_1. Furthermore, the radius at the end point of S_1 is equal to the radius of C. S_2 is similarly constrained, with its radius at the start point equal to the radius of C, its end point on L_2, and the tangent at its end point parallel to L_2. Finally, the tangents of S_1 and C at their intersection are

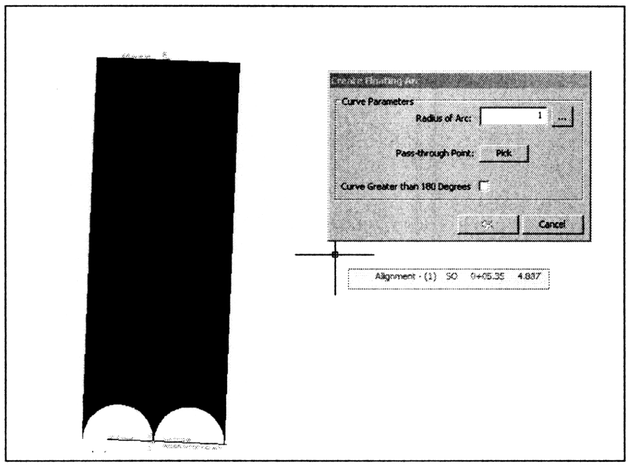


Radius of Arc: 0
Use: [1.37, 6.00]

Curve Greater than 180 Degrees ☐

Radius of Arc: Use: [1.37, 6.00]

(a)



Radius of Arc: 1

Curve Greater than 180 Degrees ☐

Alignment - (1) SO 0+05.35 4.887

(b)

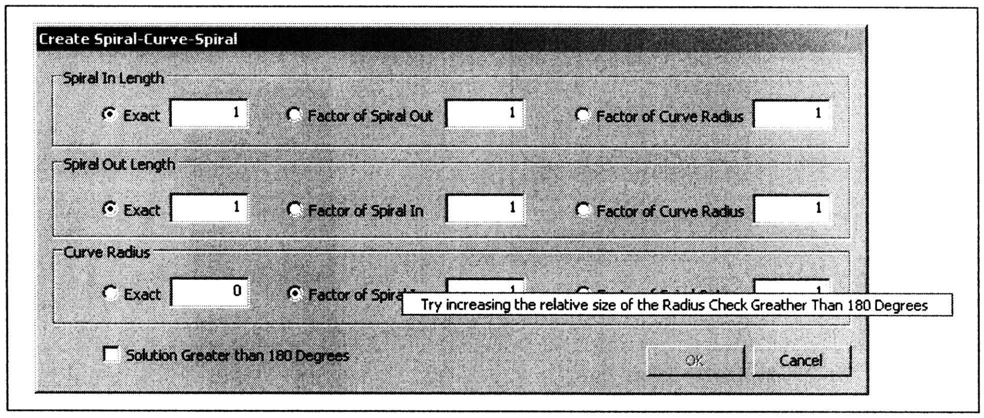
Figure 4-2: Ranges for (a) r given p, (b) p given r

identical. The same is true for S_2.

The examples given show feedback for a variety of combinations of user input. The example interface also demonstrates the ability to link different parameters with arbitrary ratios. In Figure 4-3 (a), S_1's length and S_2's length are given, and C's radius is constrained to be equal to S_1's length. This results in a state where no ranges can be found by the solver. However, the solver was able to determine that it is likely the case that the angle subtended by C is greater than 180 degrees, even though this option wasn't selected. This is displayed to the user as a message. In Figure 4-3 (b) values for S_1's length and C's radius are given. S_2's length is constrained to equal that of S_1. The values given in this example don't describe a valid solution. The solver finds ranges for the correctable parameters that will produce a valid curve. In this case, the length of S_1 can take on values in the interval $(0.0, 0.40)$, and the radius of C can take on values in the interval $(0.0, 6.96)$.

4.3 Line-SCSSCS-Line

The final example is built from four spirals and two arcs, in the sequence spiral-curve-spiral-spiral-curve-spiral. This is shown in Figure 4-4. The user is prompted for spiral lengths S_1 through S_4, arc radii r_1 and r_2, and the angle ϕ_1 subtended by the first arc.


Create Spiral-Curve-Spiral

Spiral In Length

[✓ Exact] 1 Factor of Spiral Out 1 Factor of Curve Radius 1

Spiral Out Length

[✓ Exact] 1 Factor of Spiral In 1 Factor of Curve Radius 1

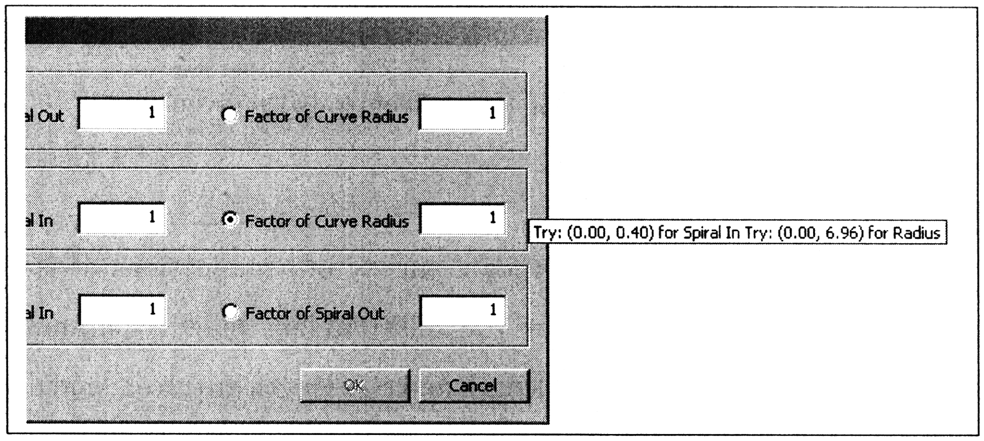
Curve Radius

Exact 0 [✓ Factor of Spir] 1 1

Try increasing the relative size of the Radius Check Greather Than 180 Degrees

Solution Greater than 180 Degrees

(a)


l Out 1 Factor of Curve Radius 1

l In 1 [✓ Factor of Curve Radius] 1

Try: (0.00, 0.40) for Spiral In Try: (0.00, 6.96) for Radius

l In 1 Factor of Spiral Out 1

(b)

Figure 4-3: Ranges given for Spiral-Curve-Spiral parameters

The radius at the end point and start point of S_1 and S_2, respectively, are constrained to equal r_1. There is a similar constraint on S_3, S_4, and r_2. The problem is broken into several sub-problems: one problem for each curve in the group.

Figure 4-5 shows the dialog for the solver and the output shown to the user. In this case the parameters with the value of 0 are not set. Given that the two radii are set, the solver is able to find a range for the angle of the first arc, which is given to the user. The valid range for the angle is in the range $(1.94, 2.41)$ radians.

After choosing a value for the first arc's angle, the problem is sufficiently constrained to find ranges for the lengths of the four spirals. This is shown in Figure 4-6. The output of the solver is shown as a list of four ranges that indicate valid lengths for each of the four spirals. These ranges represent maximum ranges and are independent of each other. Assigning a value to one of the parameters will affect the

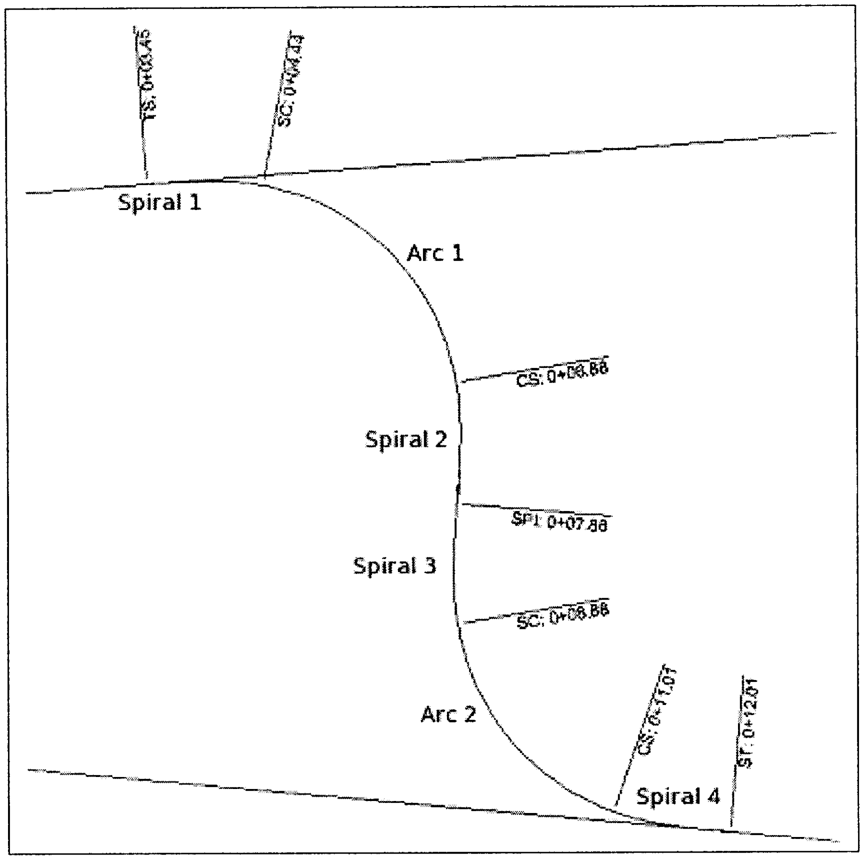
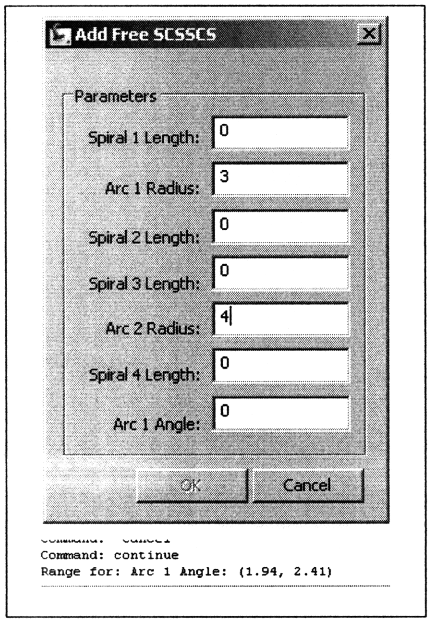


Figure 4-4: SCSSCS Curve Group

valid ranges for the remaining parameters.

Finally, in Figure 4-7, all of the parameters have been entered. In this case, the parameters define a valid SCSSCS group, and the solver outputs that a solution has been found. The output shows how the ranges for the four spirals are refined as the lengths are entered one by one. These ranges decrease in size as more information about the problem is given. The solver outputs that a solution has been found when all of the parameters are given.

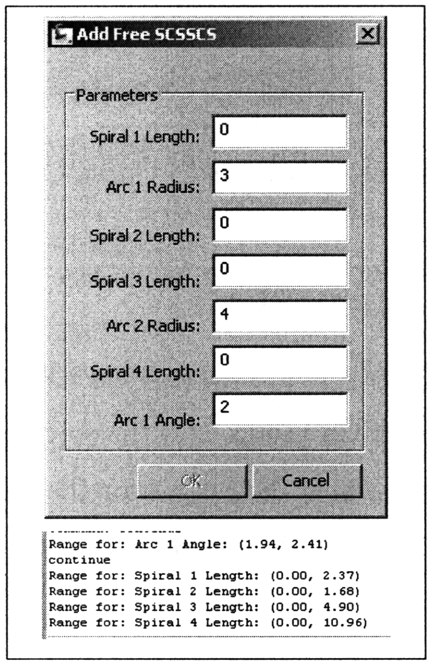
Figure 4-8 demonstrates the results when the given parameters are not solvable. The result was taken from a different set of entered parameters. All of the parameters were entered, but no solution was found. This output indicates that two subsets of the given parameters are solvable: the set containing S_1,S_2,S_3,r_1, and r_2, and the set containing S_2,S_3,S_4,r_1, and ϕ_1. If the parameters not in these sets are removed, then ranges are given for remaining parameters. In this case, the range for ϕ_1 is given



Spiral 1 Length: 0
Arc 1 Radius: 3
Spiral 2 Length: 0
Spiral 3 Length: 0
Arc 2 Radius: 4
Spiral 4 Length: 0
Arc 1 Angle: 0

```
Command: continue
Range for: Arc 1 Angle: (1.94, 2.41)
```

Figure 4-5: Dialog and output after entering r_1 and r_2.

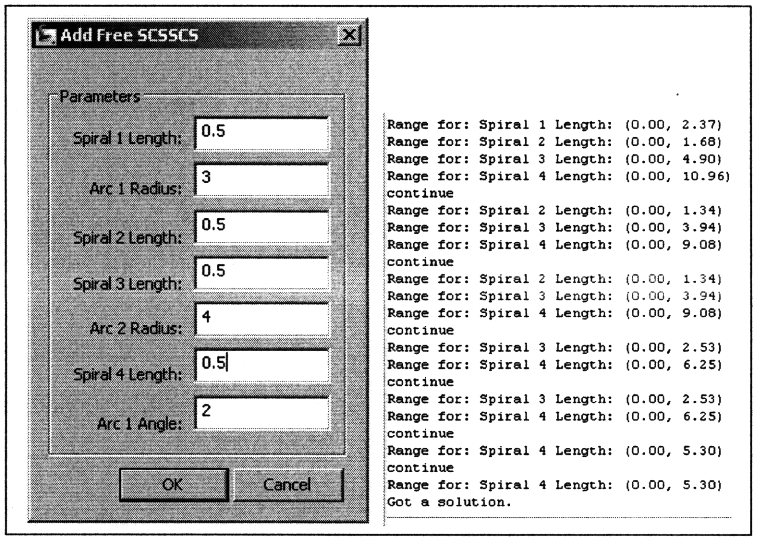


Spiral 1 Length: 0
Arc 1 Radius: 3
Spiral 2 Length: 0
Spiral 3 Length: 0
Arc 2 Radius: 4
Spiral 4 Length: 0
Arc 1 Angle: 2

```
Range for: Arc 1 Angle: (1.94, 2.41)
continue
Range for: Spiral 1 Length: (0.00, 2.37)
Range for: Spiral 2 Length: (0.00, 1.68)
Range for: Spiral 3 Length: (0.00, 4.90)
Range for: Spiral 4 Length: (0.00, 10.96)
```

Figure 4-6: Dialog and output after entering ϕ_1.

if the parameters missing the first set are removed; the range for r_2 is given if the parameters missing from the second set are removed.



Spiral 1 Length: 0.5
Arc 1 Radius: 3
Spiral 2 Length: 0.5
Spiral 3 Length: 0.5
Arc 2 Radius: 4
Spiral 4 Length: 0.5
Arc 1 Angle: 2

```
Range for: Spiral 1 Length: (0.00, 2.37)
Range for: Spiral 2 Length: (0.00, 1.68)
Range for: Spiral 3 Length: (0.00, 4.90)
Range for: Spiral 4 Length: (0.00, 10.96)
continue
Range for: Spiral 2 Length: (0.00, 1.34)
Range for: Spiral 3 Length: (0.00, 3.94)
Range for: Spiral 4 Length: (0.00, 9.08)
continue
Range for: Spiral 2 Length: (0.00, 1.34)
Range for: Spiral 3 Length: (0.00, 3.94)
Range for: Spiral 4 Length: (0.00, 9.08)
continue
Range for: Spiral 3 Length: (0.00, 2.53)
Range for: Spiral 4 Length: (0.00, 6.25)
continue
Range for: Spiral 3 Length: (0.00, 2.53)
Range for: Spiral 4 Length: (0.00, 6.25)
continue
Range for: Spiral 4 Length: (0.00, 5.30)
continue
Range for: Spiral 4 Length: (0.00, 5.30)
Got a solution.
```

Figure 4-7: Dialog and output after entering all parameters.

```
========
Valid set: {Spiral 1 Length, Spiral 2 Length, Spiral 3 Length, Radius 1 Length,
Radius 2 Length}
Range for:
   Arc 1 Angle: (1.56, 2.14)
========
========
Valid set: {Spiral 2 Length, Spiral 3 Length, Spiral 4 Length, Radius 1 Length,
Arc 1 Angle}
Range for:
   Radius 2 Length: (20.96, 24.58)
========
```

Figure 4-8: Backtracking to the largest set of valid parameters

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 5

Conclusion

5.1 Summary

In this thesis I presented a solution to the problem of guiding the user when the provided input has no feasible solution. This involved modifying Civil 3D in two places: the code responsible for solving the user's input and also the interface by which the parameters are specified. These two changes allow the user to easily go back and modify different parts of the input out of sequence. More significantly, these two changes enable several interfaces that give feedback to the user as the problem is solved incrementally.

I demonstrated several examples of user interfaces that could be built as a result of the changes described. These demonstrate the different types of feedback that can be presented to the user, and results from the solver showing valid ranges for user-entered parameters. Finally, these interfaces also exhibit the ability to allow the user to enter parameters and receive feedback incrementally. This also includes backtracking to subsets of the parameters which are valid.

The solution presented is advantageous because requires relatively few changes to the structure of the existing program, and because it takes advantage of the existing solver routines. This is in contrast to other methods, which take advantage of certain structures that aren't explicit in other applications.

5.2 Further Work

There are several opportunities for further work that can build upon the results of this thesis. The problem solved in this thesis is how to aid the user when his or her input is invalid. The solution I presented was to show the user valid ranges for the required parameters. This solution treats the user input as sacred: whatever is given should not be modified (which is fairly convential in CAD).

Other methods might revolve around modifying the input based on either canonical models of the geometric structures, or user input. In the first case, since the application is often tailored to a specific field (in this case, civil engineering), it's possible that a subset of the required parameters have a much more significant impact on the design than the rest. In that case, it is conceivable that modifying the less important parameters without the user's explicit instruction could be done to transform invalid sets of parameters into valid sets. Similarly, a user could assign weights to parameters to indicate relative levels of importance.

Furthermore, an interface could be designed that uses an efficient multidimensional sampling technique to explore the solution space for feasible solutions that are similar to the user-specified unfeasible geometry. These results could be presented to the user graphically. The user could then select a desirable solution, and either search for other solutions close to the selected geometry, or simply use the given geometry.

References

[1] Bernhard Bettig and Jami Shah. Solution selectors: A user-oriented answer to the multiple solution problem in constraint solving. *Journal of Mechanical Design*, 125:443–451, September 2003.

[2] Willem F. Bronsvoort and Hilderick A. van der Meiden. A constructive approach to calculate parameter ranges for systems of geometric constraints. *Computer-Aided Design*, 38:275–283, January 2006.

[3] E. S. Fraga. Generation and use of partial solutions in process synthesis. *Trans IChemE*, 76:45–54, January 1998.

[4] Ioannis Fudos and Christoph M. Hoffman. A graph-constructive approach to solving systems of geometric constraints. *ACM Transactions on Graphics*, 16(2):179–216, April 1997.

[5] Christoph M. Hoffman, Meera Sitharam, and Bo Yuan. Making constraint solvers more usable: overconstraint problem. *Computer-Aided Design*, 36:377–399, 2004.

[6] R. Joan-Arinyo and A. Soto-Riera. Combining constructive and equational geometryic constraint-solving techniques. *ACM Transactions on Graphics*, 18(1):35–55, January 1999.

[7] Aylmer L. Johnson and Zhihui Yao. On estimating the feasible solution space of design. *Computer-Aided Design*, 29(9):649–655, September 1997.

[8] Yehuda E. Kalay. Redifining the role of computers in architecture: from drafting/modelling tools to knowledge-based design assistants. *Computer-Aided Design*, 17(7):319–328, 1985.

[9] Lwangsoo Kim and Jae Yeol Lee. A 2-D geometric constraint solver using DOF-based graph reduction. *Computer-Aided Design*, 30(11):883–896, June 1998.

[10] P. D. Martins. Windowing the solution space of an optimization problem. *Computer-Aided Design*, 16(6):314–320, November 1984.

[11] Meera Sitharam, Adam Arbree, Yong Zhou, and Naganandhini Kohareswaran. Solution space navigation for geometric constraint systems. *ACM Transactions on Graphics*, 25(2):194–213, April 2006.