更多数学建模资料获取

B 站数学建模试看课程：
https://www.bilibili.com/video/av20238704
清风数学建模视频课程的课表：
http://note.youdao.com/noteshare?id=2e1cc342928466bb632b33304a56e368
视频录制了 45 个小时，累计 800 多页 PPT，讲解的了 20 多个常用模型，内容非常详细，并配有作业讲解。

关注微信公众号：《数学建模学习交流》可获得更多数学建模技巧和比赛信息。
怎么在手机上购买资料：在公众号后台发送"买"这个字，就可以获得一个链接，点进去后就可以购买了（购买方式和淘宝一样）

Planes and Helices

Samar Lotia
Eric Musser
Simeon Simeonov
Macalester College
St. Paul, MN 55105
{slotia, ssimeonov}@macalstr.edu
emusser@prodea.com

Advisor: A. Wayne Roberts

Introduction

We are asked to design, implement, and test a mathematical algorithm that locates in real time all of the intersections of a helix and a plane located in general positions in space. In addition, we must prove that our algorithm is mathematically and computationally correct.

Assumptions

- **"Real time."** We assume that "real time" means that the time to solve a reasonably "difficult" problem must be very small. For example, an algorithm that is used for the back end of a Computer Aided Geometric Design program must not impose unacceptable delays on the user. When the user repositions the helix or plane, the user expects an immediate refresh of the screen—usually in only a fraction of a second. For other applications, our algorithm should take little time compared to the calling program. To meet these requirements, the algorithm must have a time complexity linear in the number of intersections; our algorithm is indeed linear.
- **Helix/Plane.** We assume the strict mathematical definition of a single helix: that it is infinite and nonelliptical, and that it "wraps around" a cylinder. We believe that our algorithm will work for other "helices," such as those having elliptical bases; but we have not tested these cases to any extent. We also assume that the plane extends infinitely.
- **Correctness.** We assume that twelve digits of precision is acceptable for all calculations. This is sufficient for most biotechnological applications. Twelve digits means that we have error less than the radius of some atoms [Chang 1994]. For some applications, engineers often expect fewer

digits of accuracy; so our program allows the user to change the desired precision.

Summary of Approach

Our approach to the problem involves

- definition of design requirements,
- development of a mathematical model for the problem,
- design and implementation of an algorithm,
- debugging and testing, and
- evaluation.

Definition of Design Requirements

The algorithm will be used as a support routine for mission critical processes, where its failure to produce correct results, or failure to produce results on time, could have dire consequences. This fact leads us to the following design requirements, in decreasing order of importance:

- **Correctness.** The algorithm must produce correct results.
- **Robustness.** The algorithm must handle exceptions well and not terminate abnormally.
- **Performance.** The algorithm must execute in real time.
- **Efficiency.** The algorithm must spare system resources, as long as the above three requirements are not adversely affected.
- **Flexibility.** The algorithm must allow users to formulate the problem in different ways, e.g., it must allow more than one way of defining a plane or a helix. Also, users must be able to fine-tune the algorithm to improve performance.
- **Portability.** The algorithm must be machine-independent, written in a common programming language, and easy to transfer to a different programming language of choice (e.g., to include it in an embedded system).

Because algorithms execute on physical entities (computers) that have some finite working precision, it is possible that a computationally correct algorithm may produce incorrect results due to roundoff and compound errors. In our case, two types of errors are possible: skipping an existing root and reporting a root that does not exist. It is difficult to claim that one type of error should be preferred to another; we choose, if possible, to minimize the second kind of error given the possibility of some error of the first type.

Development of a Mathematical Model

Initial Development

The General Case

Having examined the problem in general Cartesian coordinates, we found that a graphical or vector analysis–related approach would fail us, in the sense that it would be incredibly hard to program. Hence, we attempted to reduce the problem to an algebraic problem, since algebraic techniques are generally much more suited to programming. In effect, we "projected" the helix onto the plane in the following manner.

Consider the general parametric equations of a helix in space (for derivation, see the **Appendix**):

$$\begin{aligned} x &= a_{11}\cos\alpha t + a_{12}\sin\alpha t + a_{13}t + a_{14} \\ y &= a_{21}\cos\alpha t + a_{22}\sin\alpha t + a_{23}t + a_{24} \\ z &= a_{31}\cos\alpha t + a_{32}\sin\alpha t + a_{33}t + a_{34}. \end{aligned}$$

The equation of a plane in space may be written as

$$ax + by + cz - d = 0.$$

Our transformation replaces x, y, and z on the left-hand side in this equation with the corresponding parametric forms of the helix, thus returning an expression in t, which we call $f(t)$:

$$f(t) = A\cos t + B\sin t + Ct - D,$$

where A, B, and C are appropriately transformed coefficients. The α in the $\cos$ and $\sin$ terms has been incorporated into C via a change of parameter from t to t/α.

The task now is to solve the equation $f(t) = 0$. After perusing relevant literature, we concluded that this equation must be solved numerically [Plybon 1992]. Hence, we developed a numerical technique that, given the parameters A, B, C, and D, attempts to locate all the roots of the equation. Many well-documented algorithms guarantee convergence to roots—given certain bounds on the problem—and give an easily computable bound on the error in the result. The numerical technique that we employ is heavily influenced by information that we have about the equation $f(t) = 0$. For instance, we know that the extrema of the function (if any) occur periodically, and we use this fact at several stages of our method, hence ensuring an efficient algorithm. We essentially built a robust, *strong* equation solver (one that uses problem-specific information to maximize the efficiency of the algorithm).

The first step in our approach is to examine the general form of the function, as in **Figure 1**. We note that all roots must be located between two extrema that are on opposite sides of the t-axis (assuming a continuous function). The only other case, which we handle separately, is when a root and an extremum or inflection point occur simultaneously.

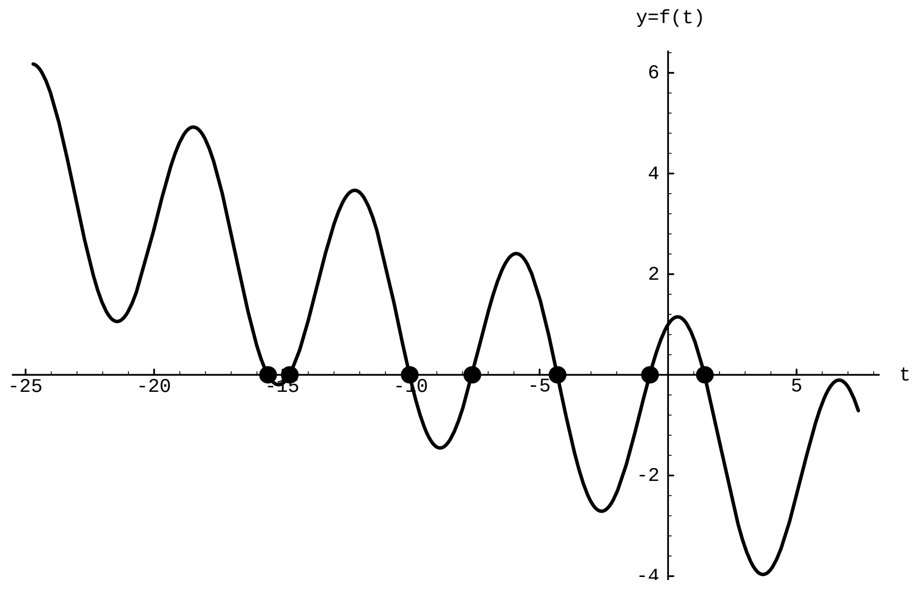

Figure 1. General functional form.

We begin by locating the minima and maxima of the function $f(t)$, which are periodic with period 2π. These are found by solving the equation

$$f'(t) = -A\sin t + B\cos t + C = 0.$$

From

$$\cos t = \frac{A\sin t - C}{B},$$

by using $\cos^2 t + \sin^2 t = 1$ we find

$$\sin t = \frac{ac \pm b\sqrt{a^2 + b^2 - c^2}}{a^2 + b^2}.$$

However, this method returns some extraneous roots because of the squaring (just as does squaring $t = 1$ and solving $t^2 = 1$). These are discarded via a simple test, namely, substituting the values back into $f'(t)$ and checking whether or not the derivative is indeed zero.

We then interpolate the root by connecting the two extrema via a line segment (as in **Figure 2**) and pass the interpolated value to our root-finding algorithm.

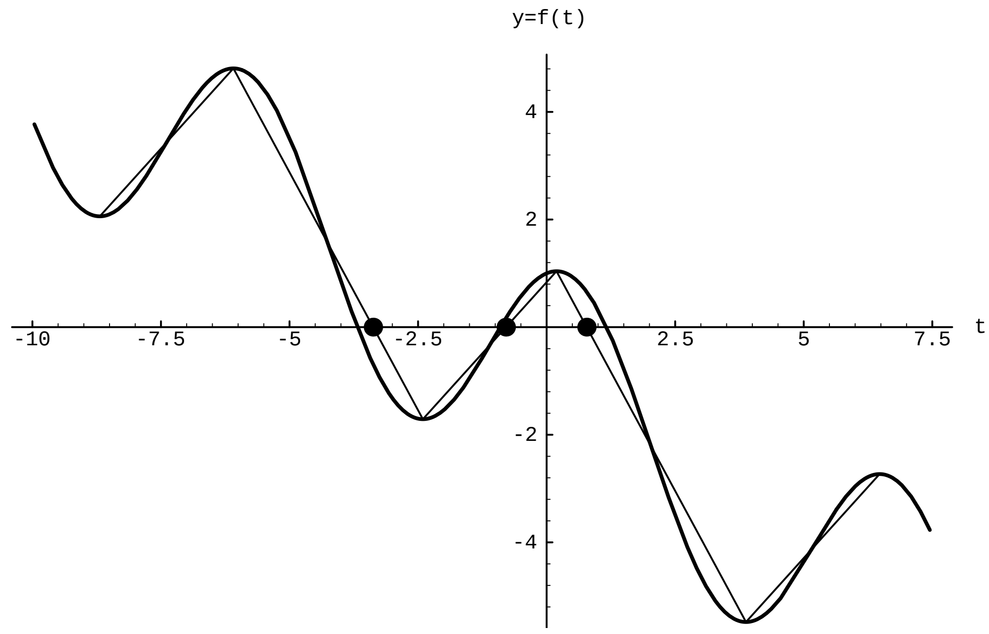

Figure 2. The interpolation method.

We must now judiciously choose a value of t that guarantees roots in its immediate neighborhood. We choose the value $t_0 = D/C$, as we are ensured that—if roots exist—there is one within 2π of this value of t (see the **Appendix** for a detailed argument).

Certain Special Cases

If the coefficient C is 0, then the function $f(t)$ is periodic and oscillates to within $\sqrt{A^2 + B^2}$ (the maximum possible value of $A\cos t + B\sin t$) about the line $g(t) = D$. Thus, if $|D| > \sqrt{A^2 + B^2}$, then the function never intersects the t-axis and we have no roots; the plane is parallel to the helix and outside the "reach" of its radius. In such a case, our program returns the message "No Roots." If, on the other hand, $|D| \le \sqrt{A^2 + B^2}$, then we have infinitely many roots; this case is handled appropriately.

Another important case is that of only one root (see **Figure 3**). This occurs when $a^2 + b^2 - c^2 \le 0$. Then the equation has either one real solution or none. If a solution is found, it is at an inflection point. This condition is recognized by our algorithm and appropriately dealt with by calling the bisection method instead of the usual Newton-Raphson (which exhibits very slow convergence when dealing with simultaneous occurrence of roots and extrema/inflection points). The bisection method guarantees convergence as long as we bracket the root properly. We are confident that we do so, as we give bisection an interval of radius 2π about the point $t_0 = D/C$. The bisection method achieves our prescribed goal of 12-digit accuracy in no more than 44 iterations.

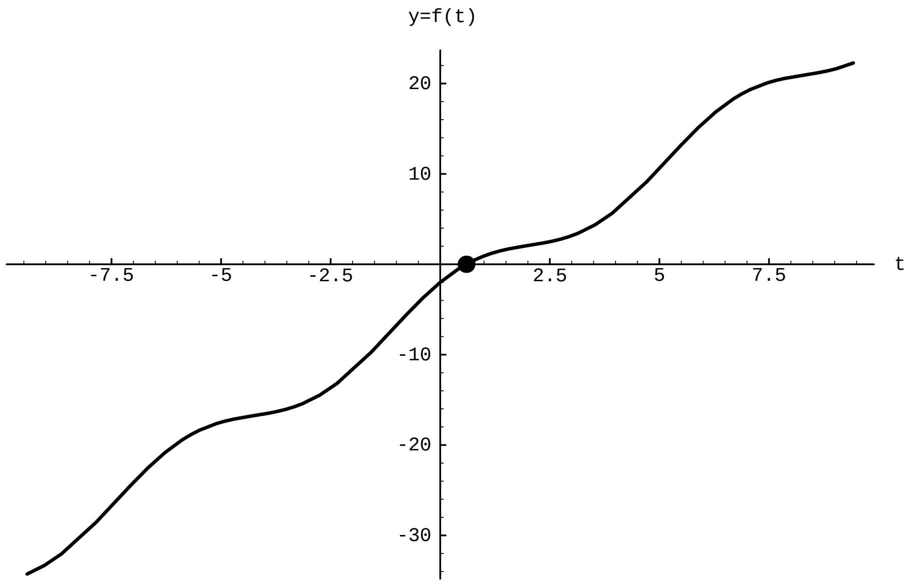

Figure 3. The single-root case.

Algorithm Description

To facilitate understanding, we provide four levels of abstraction in the description of our algorithm. At the top level, we use a linearized flowchart that shows the main subproblems that need to be solved. [EDITOR'S NOTE: Because of space considerations, we do not reproduce the flowchart.] Parallel to the flowchart, at the second level of abstraction, we offer comments that provide more detail of the workings of the algorithm. They refer to the third level of abstraction, mathematical proofs and detailed explanations. Finally, at the lowest level of abstraction is the C++ code of our program, which includes many comments with Mathematica code and references to relevant literature.

Before we go into the details of the algorithm implementation, we mention some general conventions.

- **Input**
  - Planes can be defined by the user in three ways: by general Cartesian equation of the form $ax + by + cz = d$, by two vectors and a point, or by three points.
  - Helices can be defined by the user in two ways: by general parametric equations, or by the three Eulerian angles and the translation vector

that map the z-axis to the central axis of the helix.

- **Output**
  - If the helix does not intersect the plane, no roots are returned.
  - If there are infinitely many solutions (a case in which the plane is parallel to the helix axis), sufficient information is provided so that the user can generate all of the intersection points.
  - Otherwise, a structure containing the x, y, and z coordinates of the points of intersection is produced.
- **Accuracy of Estimation.**

  The default working precision of calculation in our C++ program is twelve digits, but it can be changed by modifying a single variable in the code. The maximum working precision is limited by the floating point precision of the computer.
- **Portability.**
  - Our algorithm is implemented in ANSI C++, ensuring portability across most computing platforms. The code does not use machine-dependent features, and it could be translated to any procedural language.

Testing and Quality Control

We devoted more than half of our algorithm design and implementation efforts to testing. We checked the correctness of ideas and implementations at four different levels:

- **Math Model.** All transformations and function forms that we used were generated symbolically using Mathematica's standard features and the `Vector Analysis` and `Rotations` packages. All symbolic solutions to equations were checked using Mathematica. Whenever possible, we simplified expressions, sometimes by applying trigonometric substitution rules manually.
- **Algorithm Design.** Our root-finding procedure was carefully chosen always to find a bracketed root (if necessary, by invoking bisection).
- **Implementation.** We applied the function `CForm` to convert Mathematica expressions to C code when transferring expressions to our implementation. That minimized the chances of erroneous expression entry. The root-finding procedure that we use was taken from Plybon [1992]. The procedure was independently tested against the built-in Mathematica

routine `FindRoot`, which implements a combination of Newton-Raphson and the secant methods [Wolfram 1991]. Our procedure never failed to find a root and never reported a root where there was none. In several cases where `FindRoot` failed, our procedure correctly managed to find a root.

- **Runtime.** We performed three different types of checks on the output of our program:
  - We generated more than 50 functions of the form $f(t) = A\cos t + B\sin t + Ct - D$ and checked whether our program correctly found their roots. We implemented a set of Mathematica routines that finds the roots of the equation $f(t) = 0$ by the same algorithm as our program but with higher accuracy. In all test cases, the output of our program agreed with Mathematica's output, suggesting that round-off error is not a major problem in our implementation. In most cases, we visually inspected the graph of $f(t)$ to ensure than no roots were missed and that no false roots were introduced. The tests included a mix of test cases including none, one, many, and infinitely many roots. We considered potentially problematic cases, such as roots at tangency points and roots at inflection points. We explored and tested all control paths of the algorithm. Debugging output was generated and investigated carefully.
  - We inputted more than fifty helices and planes in various input formats and used our algorithm to find the coordinates of the intersection points between them. Then for each test run we used Mathematica to do a 3-D plot of the plane, the helix, and the intersection points (see **Figure 4**). We inspected the 3-D plots from various viewpoints to ensure that no intersection points were missed and that no extraneous points were plotted. Our program passed all tests.
  - In the testing, often we were uncertain about the actual location of the intersection points. So we designed an additional battery of tests to check the results of our program against a known (sub)set of the intersection points. We obtained the known intersection points by starting with an arbitrary helix and defining an intersecting plane either by choosing three points on the helix or by choosing a point on the helix and an arbitrary vector. In the first case, we knew the coordinates of at least three intersection points. In the second case, we could position the plane so as to experiment with different patterns of intersection. We then checked the results of our program against subset of known roots. In all of the more than 50 test cases, our algorithm performed correctly.

In *all* test cases, our program performed successfully. This result—added to the fact that our algorithm is closely based on a rigorously proven mathematical model—gives us a high degree of confidence that our program is indeed computationally correct. Also, we have seen no evidence that roundoff or compound errors are a significant source of error.

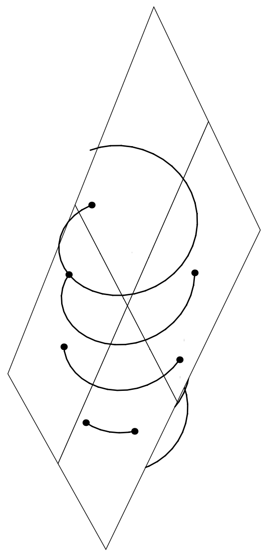

Figure 4. Our solutions plotted on the given helix and plane.

Evaluation

Correctness

Our tests have shown that mathematically and computationally our algorithm is correct. Numerically, however, problems can arise. The possibility of compounding error and loss of precision is inevitable in some cases (though rare).

Compounding error is introduced in routines that map our original helix onto a helix located about the z-axis (i.e., computing the coefficients A, B, C, and D, for $f(t)$). We are uncertain of the error bounds on these calculations, but comparisons with high-precision Mathematica runs show that the error is small (less than 10^{-12}).

Fortunately, given correct values of A, B, C, and D, we can guarantee that the roots found are accurate to the desired working precision. This is because we use Newton-Raphson and the bisection method as our root-finding techniques. With Newton's method, it is possible to get some idea of absolute error of a root x_0 simply by looking at the value of $f(x_0)$ (it should be 0 if we have a root). Also, error is not compounded with Newton's method; each iteration actually decreases the error. Newton-Raphson converges quadratically [Plybon 1992], and near a root the number of significant digits approximately doubles with each step.

Detrimental error can also enter the problem when calculating the brackets for each root. If error in numerical computation places a bracket on the wrong side of a root, that root may be lost. We have never encountered such a case but expect that cooked-up data could produce such an error. A possible remedy (not guaranteed to work in all cases) is first to approximate the bracket using our exact formulas and then to run a root-finding method on them to reduce the error. We did not implement this strategy, because we feel that the chance of this error occurring does not warrant the loss in speed that will occur.

Further, in searching for solutions of $f(t) = 0$, if two roots are extremely close, they can unfortunately be mistaken as one. However, if two extremely close roots are found, it may not even make sense to consider them as distinct, since they may be the product of numerical error.

Robustness

Our algorithm checks extensively for exception cases and will not terminate abnormally due to a computation error or lack of system resources. We implemented checks for special cases that in effect trap errors. One example: We check for an infinite number of roots before we search for roots. We handle "special" cases such as tangencies, inflection points, double roots, etc.

Our algorithm ensures that all roots are bracketed. This prevents Newton's method from accidentally going off and finding another root. Our implementation of Newton's method incorporates the bisection method in cases where Newton's does not converge fast enough or Newton's method departs the bracketed interval. The bisection method is guaranteed to find a root within a bracketed interval [Plybon 1992].

In our algorithm, there is an inherent limit on the number of possible roots; but this limit can be easily increased or even eliminated (allowing memory of the computer to be the only limitation).

Performance/Efficiency

The performance of our algorithm is linear in the number of intersections. This is because we do a single root-find for each intersection (including bracketing). The complexity of a root-finding method is dependent on the how the function is shaped and the number of digits desired. Typically, when Newton's method is used, 5 or 6 iterations are required to find each root to 12 digits of precision, while the bisection method can be shown to take fewer than 44 iterations.

For mapping the helix, finding first-order roots, etc., the time required is relatively constant. Therefore, the time complexity of the algorithm is dominated by the number of intersections.

In addition, our code is efficient in use of space, since the space complexity is also linear in the number of intersections.

Suggestions for Improvement

As the general organization of our algorithm is closely based on a mathematical model, we believe that it is not possible to improve it without a thorough revision of the underlying methodology. However, the *implementation* of the algorithm can be improved in several ways:

- One can attempt to modify the evaluation of expressions so as to reduce the compound and roundoff errors. This often necessitates understanding and use of architecture-specific features of the processor on which the algorithm is executing, thus limiting portability.
- One can attempt to find bounds for the error introduced during the calculation of the A, B, C, and D coefficients for $f(t)$. An estimation of the relationship between this error and the error generated by the root-finding algorithm will also be helpful.
- A useful yet hard-to-implement feature would be the inclusion of internal validation routines that improve correctness and robustness by monitoring for unacceptable computational errors while the algorithm is executing. Such routines would adversely affect performance.
- The input procedures could potentially be extended to include alternative definitions of a helix. However, our explorations of the relevant literature yielded no definitional forms different from the ones that we use.
- The Mathematica routines used in the testing process could be extended to infinite-precision calculation, running in batch mode to check at random the correctness of solutions provided by the real-time algorithm.
- The algorithm could be extended to handle other types of helices: alpha helices, double helices, etc. The underlying methodology of the algorithm should remain unchanged.

Appendix

General Parametric Equations for a Helix

Consider a general 3×4 rotation-translation matrix:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{33} & a_{34} & a_{34} \end{bmatrix}$$

and the "unit" helix (in vector form)

$$(\cos(\alpha t - t_0), \sin(\alpha t - t_0), t).$$

Applying the matrix produces any general helix:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{33} & a_{34} & a_{34}. \end{bmatrix} \begin{bmatrix} \cos(\alpha t - t_0) \\ \sin(\alpha t - t_0) \\ t \\ 1 \end{bmatrix},$$

giving the general parametric equations of a helix in space:

$$\begin{aligned} x &= a_{11}\cos(\alpha t - t_0) + a_{12}\sin(\alpha t - t_0) + a_{13}t + a_{14} \\ y &= a_{21}\cos(\alpha t - t_0) + a_{22}\sin(\alpha t - t_0) + a_{23}t + a_{24} \\ z &= a_{31}\cos(\alpha t - t_0) + a_{32}\sin(\alpha t - t_0) + a_{33}t + a_{34}. \end{aligned}$$

Upon expanding the $\sin$ and $\cos$ terms, we obtain the same format as presented in the text of the paper.

Why $t_0 = D/C$?

The equation

$$f(t) = a\cos t + B\sin t + Ct - D$$

can be thought of as the intersection of a helix with parametric equations

$$x = \cos t, \qquad y = \sin t, \qquad z = t$$

with a plane with equation

$$Ax + By + Cz = D.$$

As far as the root-finder is concerned, we have a "vertical" helix with radius 1 and the plane as above. Thus, we say that the point at which the helix's central axis (also the z-axis for the root-finder) meets the plane $Ax+By+Cz = D$ is the point around which the roots are distributed almost symmetrically.

The justification for this claim is as follows. The intersection of a plane and a cylinder in space is an ellipse; as the helix lies on a cylinder, its intersections with a plane must lie on an ellipse. The center of the ellipse is the point of intersection of the plane and the helix's central axis. In **Figure 5**, we show the cylinder that the helix sits on, the ellipse of intersection with the plane, and the helix itself. The curve represents the ellipse of intersection and the lines are the helix. The whole picture shows the cylinder "unfolded."

It can be shown that if any roots exist, they must do so within one complete rotation of the center of the ellipse. Hence, we use the center as the starting point for our root-finder.

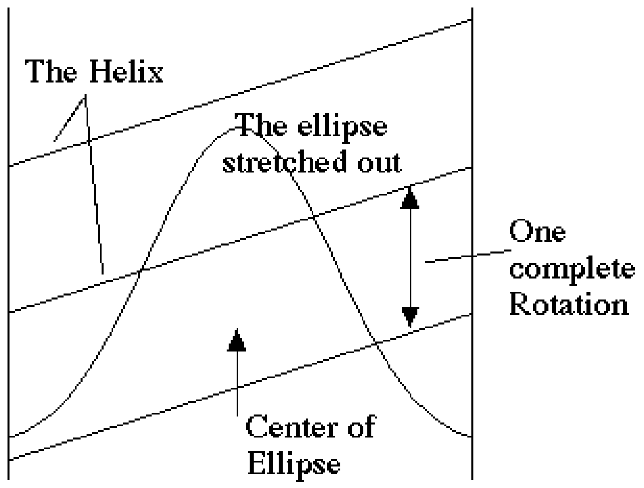

Figure 5. Representation of the intersection of the helix and the plane.

References

Canale, Raymond P., and Steven C. Chapra. 1985. *Numerical Methods for Engineers.* Reading, MA: Addison-Wesley.

Chang, Raymond. 1994. *Chemistry.* 5th ed. New York: McGraw-Hill.

Englefield, M.J. 1987. *Mathematical Methods for Engineering and Science Students.* Edward Arnold.

Press, William F., et al. 1990. *Numerical Recipes in Pascal: The Art of Scientific Computing.* New York: Cambridge University Press.

Plybon, Richard F. 1992. *Applied Numerical Analysis.* Boston: PWS-Kent.

Wolfram, Stephen. 1991. *Mathematica: A System for Doing Mathematics by Computer.* 2nd ed. Reading, MA: Addison-Wesley.