DAN KALMAN

Calculus in Orbit

Interestingly enough, my experiences in orbit all took place on the ground. As an applied mathematician at the Aerospace Corporation, I worked on computer simulations of satellite systems. These are programs that compute the positions of satellites relative to the earth at various times, so that questions about communications between satellites and points on the ground can be analyzed. As you might imagine, the methods of calculus were indispensable for these simulations. In fact, in one project I relived my calculus past by tackling a standard textbook problem: related rates. For that project I helped develop a computer system using a new approach called object oriented programming. Although at the core this was a project in computer software, it required a thorough understanding of calculus.

DAN KALMAN is an assistant professor of mathematics at The American University.

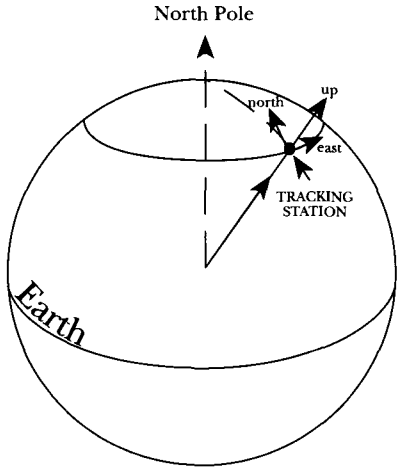
Related Rates with a Vengeance

Here is a related rates problem: A satellite travels in an elliptical orbit centered at the earth. We know how to predict the position, velocity, and acceleration of the satellite at any time. As the satellite travels along its ellipse, the earth turns beneath it, carrying along a tracking station at a known latitude and longitude. The tracking station has an antenna dish that can move like a desk chair. That is, it can swivel in a complete circle about a vertical axis, and it can simultaneously rock backward. As the satellite flies overhead, the satellite dish follows it. Picture a desk chair with a head brace. You are strapped in so that your head is held in place and you can only look straight ahead. Now swivel and rock the chair so that you can see the satellite as it flies past. It won't go straight overhead. If you sit still it will pass somewhere to the side of your right shoulder. But you don't sit still. You turn to the right and tip back so that you are always looking right at the satellite. That is what the antenna has to do.

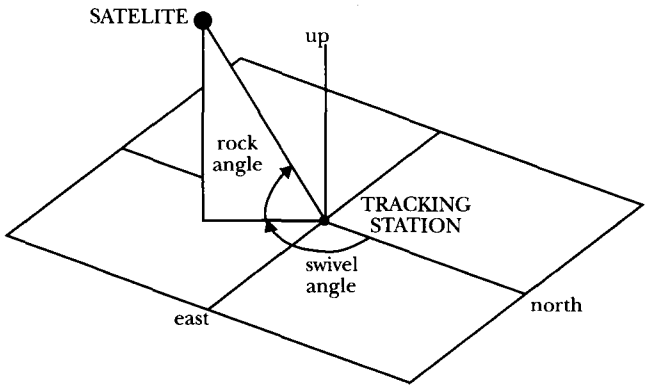
Now here is the related rates part: the antenna turns in two ways: a swivel and a rock. For each of these motions, determine the velocity and acceleration at any time during tracking.

Differentiation: Don t try this at home

As in most related rates problems, it is possible to use explicit differentiation to solve this problem. First, you must write down an explicit formula for the instantaneous swivel or rock angle as a function of time. Then take the derivative, plug in the time, and presto—you have the velocity. Another differentiation gets you the acceleration. But you wouldn't want to do this, even using a symbolic algebra system like *Mathematica*. The equations just get too long and involved.



Map directions at tracking stations



Swivel angle and rock angle

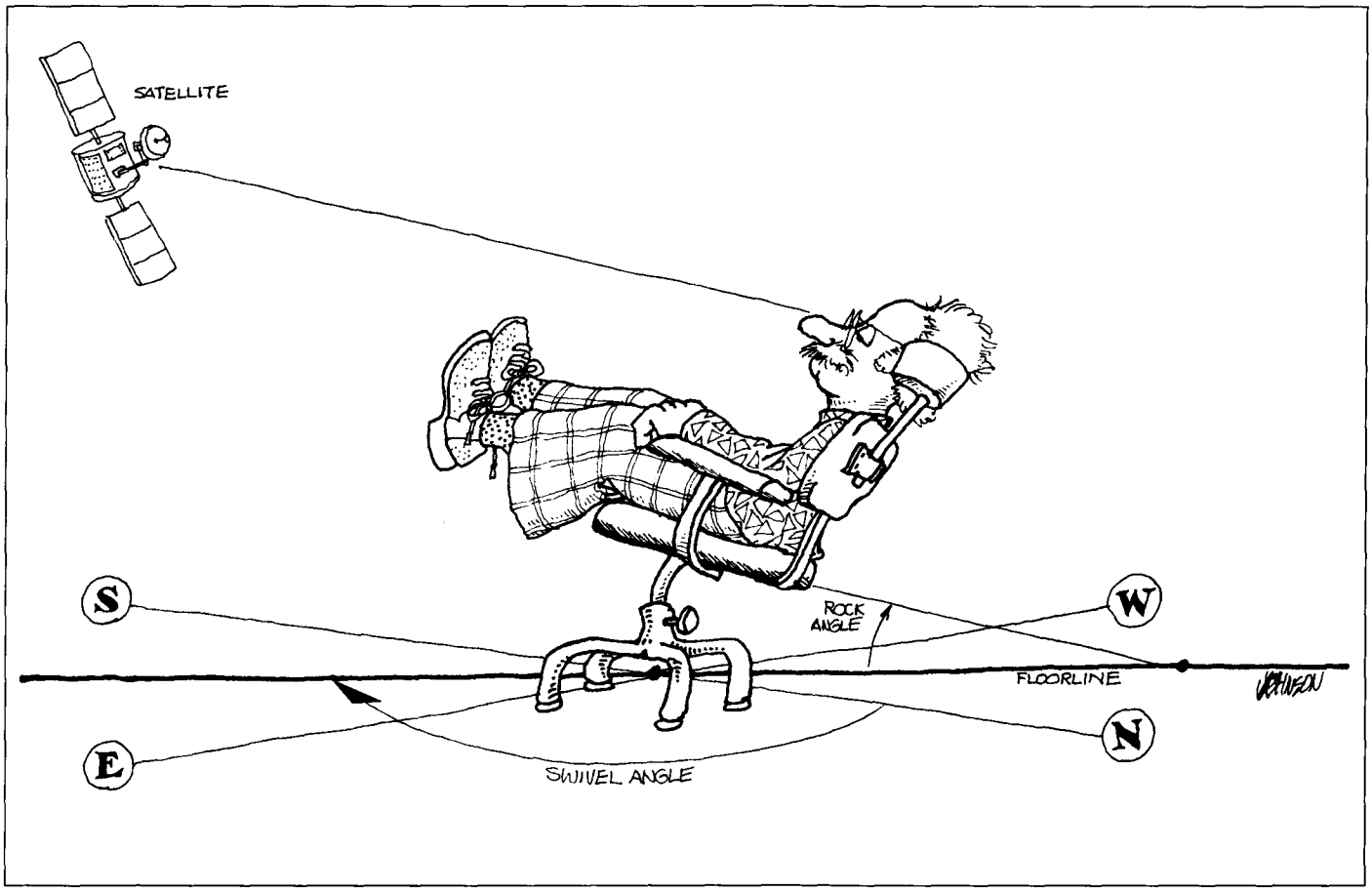


Illustration by John Johnson

To illustrate, let's consider the rock angle. Freeze the satellite in its orbit and the earth on its axis at an instant of time. Now we know the positions of the satellite and the tracking station in space. Draw a line between them. Also, at the tracking station, draw a line pointing straight up, along the axis for the swivel motion. Now find the angle between the two lines, and subtract from $\pi/2$. That's the rock angle. The geometry can all be transformed into calculations. In place of drawing lines we write down vectors. Divide each vector by its length to get unit vectors. Take the dot product of these vectors to find the cosine of the angle between them. Take the inverse cosine and subtract from $\pi/2$. This allows us to calculate the rock angle at any time. The swivel angle is a little more complicated because we need a point of reference—say due north from the tracking station—but is similar in nature. If you go to the trouble to actually work out the formulas for the two angles, you will see why no one would want to try writing down their derivatives.

Object Oriented Calculus

An alternative is provided by object oriented programming. In this application, we will think of each function of time as an object. There are two kinds of these objects. First, the *basic* function objects, are directly programmed into the system. The functions for the satellite and tracking station positions are examples of basic function objects. We will call these objects ***Satellite*** and ***Tracker***. Second there are *composed* function objects—those that are created by combining or operating on other function objects. In the example above, the vector from the tracking station to the satellite is a composed function object, defined by subtracting two basic function objects: ***Satellite*** and ***Tracker***. Call this object the ***Line-of-Sight***. Here are two more composed function objects: the ***Squared-Distance*** and the ***Distance***. We define the ***Squared-Distance*** as the dot product of ***Line-of-Sight*** with itself, and we define the ***Distance*** as the square-root of ***Squared-Distance***. Ultimately, we can define composed function objects at each step along the way until we reach objects for the ***Swivel-Angle*** and the ***Rock-Angle***.

Now in object oriented programming, each object has some information, and they all communicate by passing messages back and forth. For our system, each function object knows how to compute its own value at any time, as well as the values of the first two derivatives. For the basic function objects, this information is programmed in. If I send a message to the ***Satellite*** and ask for its position and velocity at the time 3.7, it knows how to compute those results, and sends me the answers back. The composed objects work a little differently. Each composed object

has a list of *parts*, that is, the function objects that it was created from. For example, the ***Line-of-Sight*** knows that it was created from ***Satellite*** and ***Tracker***. When ***Line-of-Sight*** receives a message asking for its value and derivative at time 7.8, it turns around and asks each of its parts for the same information. To compute its own value, ***Line-of-Sight*** simply subtracts the values it receives from ***Satellite*** and ***Tracking-Station***. Similarly, to compute its own derivative it subtracts the derivatives it receives from ***Satellite*** and ***Tracking-Station***. Then it responds to the original message by sending back its own value and derivative. Notice that in all cases, what is sent in the messages and what is operated on are numerical values. There are no formulas in the message. However, there is a formula in the programming for ***Line-of-Sight***. Since it is a difference of two other function objects, ***Line-of-Sight*** is programmed to compute its derivative by subtracting the derivatives of its parts. The rule for the derivative of a difference is built in.

This system can be used to find the derivatives of very complicated functions like ***Swivel-Angle*** and ***Rock-Angle***. Send a message to ***Rock-Angle*** asking for its derivative at time 26.3. It knows that it was defined as the difference between two things ($\pi/2$ and ***Angle-from-Vertical***) so it sends them each messages asking for values and derivatives at 26.3, and then combines the results as it was programmed to do. One of the objects that ***Rock-Angle*** queries must send out a message of its own, to get the derivative information from its own parts. So the messages flow from ***Rock-Angle*** back through all the intermediate functions until the basic functions are reached. These know how to compute their own derivatives, and send the results forward. At each intermediate stage, the derivatives and values from the preceding stages are combined in the appropriate way, until ***Rock-Angle*** ultimately reaches the value for its derivative.

This entire system really depends on a fairly small number of operations. There are the usual operations of arithmetic: addition, subtraction, multiplication, and division, as well as some calculator style functions such as square-root, trigonometric functions and their inverses, exponential functions, and so on. These all operate on scalar function objects. Then there are vector operations of addition, subtraction, scalar multiplication, dot products and cross products. The rules of differentiation, including the chain rule, are built into the system. For example, when a new function object is defined as the square-root of an existing function object, the system knows that the derivative will be calculated by dividing the derivative of the existing function by twice the square-root of the function value, i.e.,

$$D\left(\sqrt{f(t)}\right) = \frac{f'(t)}{2\sqrt{f(t)}}$$

We build into the system the rules of differentiation for the few operations listed above. Then we can compute the derivatives of any composed function automatically, without ever working out the formulas for the derivatives. Now that is the way to do calculus in orbit.

Automatic Differentiation

The system described here uses object oriented programming, but it also uses a technique called automatic differentiation. If you would like to learn more about this subject, there are two excellent articles you can read. The first is by one of the fathers of the subject, Louis Rall. The article is: "The arithmetic of differentiation," *Mathematics Magazine*, volume 59 no. 5, pp. 275–282, December, 1986. The second was awarded the Pólya Prize for outstanding mathematical writing. It is; "Automatic differentiation and APL," by Richard D. Neidinger, *College Mathematics Journal*, volume 20 no. 3, pp. 238–251, May 1989. ■