A CASE STUDY IN TRAJECTORY OPTIMIZATION: PUTTING ON AN UNEVEN GREEN

ROBERT J. VANDERBEI

Operations Research and Financial Engineering
Princeton University
ORFE-00-4

Revised September 19, 2000

ABSTRACT. We study in this paper the problem of how to putt a golf ball on an uneven green so that it will arrive at the hole with minimal final speed. This problem serves as a good case study for trajectory optimization as it illustrates many of the issues that arise in trajectory optimization problems. This putting example is just one of a collection of case studies that is submitted to *Optimization and Engineering* under the title "Case Studies in Trajectory Optimization: Trains, Planes, and Other Pastimes". The purpose of these studies is to illustrate how recent advances in algorithms and modeling languages have made it easy to solve difficult optimization problems using off-the-shelf software.

1. INTRODUCTION

The problem of how to putt provides a simple framework for a discussion of trajectory optimization. The discussion in this paper is taken from a longer paper [9] on case studies in trajectory optimization. The purpose of these studies is to illustrate how recent advances in algorithms and modeling languages now make it easy to solve once difficult optimization problems using off-the-shelve software. A secondary goal is to show that it is nonetheless still possible to make subtle errors in a model which will render it (a) more difficult than it needs to be or (b) infeasible or, worse, (c) feasible but giving the wrong answer. In the past, trajectory optimization problems were thought to be difficult to solve and when failures occured it was unclear whether they were due to bad algorithms or bad models. Today, one can say that failures are most likely due simply to bad models.

We express our optimization models in the AMPL modeling language [4]. This language provides a common mechanism for conveying problems to codes to solve them. When solving problems we generally use two different solvers: (a) LOQO [7, 8, 10, 2], which implements an interior-point method for general nonlinear optimization and (b) SNOPT [5], which implements an active set strategy with a quasi-Newton method for the QP subproblem.

One of the lessons to be learned with the putting example is how easy it is to make a wrong model. With this in mind, we advise the interested golfer to read beyond the next section because the first model, right as it may appear, is wrong.

2. THE ALESSANDRINI MODEL

We begin with a discussion of the problem essentially as it appears in [1].

Given a golf ball sitting at rest on a putting green, the problem is to figure out how to hit the ball so that it will go into the cup. To make sure that it does not just skim over the cup and stop at some point far beyond, we try to have the ball arrive at the cup with the smallest speed possible.

The Normal Vector. We assume that the elevation of the green is given as $(x, y, z(x, y))$ and that its shape is given by $(x/a)^2 + (y/b)^2 \leq 1$. Two tangent vectors to the surface are provided by $(1, 0, \partial z/\partial x)$ and $(0, 1, \partial z/\partial y)$. By taking the cross product of these two vectors, we obtain an upward pointing normal vector to the surface:

$$(-\partial z/\partial x, -\partial z/\partial y, 1).$$

The normal force N exerted by the surface of the green on the golf ball must point in this direction and its magnitude must be such that the total force in this direction vanishes (to keep the ball rolling on the surface).

The Normal Force. Since the only forces that are not tangential to the green are the force of gravity and the normal force itself, we must have the projection of the force of gravity on the normal direction be exactly opposite to the magnitude of the normal force:

$$-mg(e_z \cdot N)/\|N\| = -\|N\|,$$

where m is mass of the ball, g is acceleration due to gravity, e_z is the unit vector pointing in the vertical direction, and of course N is proportional to the normal vector given above. From this relation, we get that

$$N_z = \frac{mg}{(\partial z/\partial x)^2 + (\partial z/\partial y)^2 + 1}$$

Date: September 19, 2000.

1991 *Mathematics Subject Classification.* Primary 65L10 Secondary 34B15.

Key words and phrases. trajectory optimization, optimal control, constrained optimization.

Research supported by NSF grant DMS-9870317, ONR grant N00014-98-1-0036.

and that

$$N_x = -\partial z/\partial x N_z \qquad N_y = -\partial z/\partial y N_z.$$

Friction. There is friction between the ball and the green. It is assumed to be proportional to the normal force and to point in a direction opposite to the velocity:

$$F = -\mu \|N\| \frac{v}{\|v\|}.$$

Equations of Motion. If we denote the trajectory by $u(t) = (x(t), y(t), z(t))$, then the equations of motion are

$$\begin{aligned} v &= \dot{u} \\ a &= \dot{v} \\ ma &= N + F - mge_z. \end{aligned} \tag{1}$$

Boundary Conditions. The initial and final positions are known,

$$u(0) = u_0 \qquad \text{and} \qquad u(T) = u_f,$$

but the time T at which the final position is reached is a variable.

This problem can be cast as a (nonconvex) nonlinear optimization problem by discretizing the time interval $[0, T]$ into N small time segments and writing discrete approximations for the derivatives that appear in the model. There are many ways to do this. In this paper, we discuss two popular discretizations: midpoint discretization and trapezoidal discretization. We begin with the midpoint method. Letting `x[j]`, `y[j]`, and `z[j]` denote the positional coordinates at time jT/N, `j=0,1,...,N`, we define discrete approximations to the three components of velocity at the midpoint of each time interval as follows:

```
vx[j+0.5]=(x[j+1]-x[j])/(T/N),
vy[j+0.5]=(y[j+1]-y[j])/(T/N),
vz[j+0.5]=(z[j+1]-z[j])/(T/N),
```

`j=0,1,...,N-1`. Discrete approximations for acceleration are defined similarly:

```
ax[j] = (vx[j+0.5]-vx[j-0.5])/(T/N),
ay[j] = (vy[j+0.5]-vy[j-0.5])/(T/N),
az[j] = (vz[j+0.5]-vz[j-0.5])/(T/N),
```

`j=1,...,N-1`. The equations of motion given by (1), together with the boundary conditions, complete the constraints defining the model:

```
ax[j] = (Nx[j] + Fr_x[j])/m,
ay[j] = (Ny[j] + Fr_y[j])/m,
az[j] = (Nz[j] + Fr_z[j])/m - g.
```

Here, `Nx[j]`, `Ny[j]`, and `Nz[j]` are shorthand for

```
Nz[j] = m*g/(dzdx[j]^2 + dzdy[j]^2 + 1),
Nx[j] = -dzdx[j]*Nz[j],
Ny[j] = -dzdy[j]*Nz[j]
```

and `Fr_x[j]`, `Fr_y[j]`, and `Fr_x[j]` are shorthand for the three components of friction along the trajectory. Our first ampl model for this problem is shown in Figure 1. In this particular instance the shape of the green involves two rather flat, but slightly sloped, sections with a smooth ramp between them. The ball is initially on the lower section and the cup is on the higher section, a difficult putt similar to the one Tiger Woods faced on the 18th hole in the final round of the 2000 PGA Championship. The function $z(x, y)$ we use to define this ramp is

$$z(x, y) = -0.3 \arctan(y) + 0.05(x + y).$$

Neither LOQO nor SNOPT was able to solve the model shown in Figure 1. When this happens, it is natural to suspect that the problem is infeasible. Why should the model in Figure 1 be infeasible? Alessandrini was able to solve supposedly the same model (using a different elevation function for the green). We tried several different surfaces and they all fail with all codes *except* when the surface is planar (including, of course, tilted planar surfaces). Every optimizer we tried is able to solve such planar problems easily. This proved to be a good hint that something is wrong with the model.

After much pondering, it occured to us that z is being specified in two ways—once as an explicit function of x and y and a second time as the solution to a differential equation. Since the differential equation is computed by a somewhat crude discretization, it is entirely possible that the two specifications are enough different from each other to render the model infeasible. So, we tried two things:

(1) Removing from the model the explicit statement of how `z` depends on `x` and `y`. That is, we changed

```
var z{i in 0..n} = -0.3*atan(y[i])
                   + 0.05*(x[i]+y[i]);
```

to just

```
var z{i in 0..n};
```

(This, we later learned, is how Alessandrini formulated the problem.)

(2) Removing from the model the part of the differential equation that relates to the `z` component of the trajectory. That is, we removed the constraints `newt_z`, `zinit`, and `zfinal`.

The first of these changes produces a model that solves easily while the second one appears still to be infeasible. Hence, we seem to be on to something but more errors may be lurking. The trajectory found with the elevation constraint removed is shown in Figure 2. This trajectory looks almost right except that it seems to go airborne in the early part of the trajectory and then tunnel into the grass in the final stages. The ball is clearly not staying on the green but instead is flying through the air to the cup. This indicates that our differential equation for z is wrong. And, if it is wrong, then the equations for x and y ought to be wrong as well.

```
param g := 9.8; # acc due to gravity
param m := 0.01; # mass of a golf ball
param x0 :=  1; # coords of start pt
param y0 :=  2;
param xn :=  1; # coords of ending pt
param yn := -2;
param n := 50; # num of time points
param mu;

var T >= 0; # total time for the putt
var x{0..n};  # coords of the traj
var y{0..n};

var z   {i in 0..n}
   = -0.3*atan(y[i])+0.05*(x[i]+y[i]);
var dzdx{i in 0..n}
   = 0.05;
var dzdy{i in 0..n}
   = -0.3/(1+y[i]^2) + 0.05;

# v[i] denotes the deriv at midpt of
# the interval i(T/n) to (i+1)(T/n).
var vx{i in 0..n-1} = (x[i+1]-x[i])*n/T;
var vy{i in 0..n-1} = (y[i+1]-y[i])*n/T;
var vz{i in 0..n-1} = (z[i+1]-z[i])*n/T;

# a[i] denotes the accel at midpt of
# the interval (i-0.5)(T/n)
# to (i+0.5)(T/n), i.e. at i(T/n).
var ax{i in 1..n-1} = (vx[i]-vx[i-1])*n/T;
var ay{i in 1..n-1} = (vy[i]-vy[i-1])*n/T;
var az{i in 1..n-1} = (vz[i]-vz[i-1])*n/T;

var Nz{i in 1..n-1}
   = m*g/(dzdx[i]^2 + dzdy[i]^2 + 1);
var Nx{i in 1..n-1} = -dzdx[i]*Nz[i];
var Ny{i in 1..n-1} = -dzdy[i]*Nz[i];
var Nmag{i in 1..n-1}
   = m*g/sqrt(dzdx[i]^2 + dzdy[i]^2 + 1);

var vx_avg{i in 1..n-1}
   = (vx[i]+vx[i-1])/2;
var vy_avg{i in 1..n-1}
   = (vy[i]+vy[i-1])/2;
var vz_avg{i in 1..n-1}
   = (vz[i]+vz[i-1])/2;
```

```
var speed{i in 1..n-1}
   = sqrt(vx_avg[i]^2 + vy_avg[i]^2
                      + vz_avg[i]^2);

var Frx{i in 1..n-1}
   = -mu*Nmag[i]*vx_avg[i]/speed[i];
var Fry{i in 1..n-1}
   = -mu*Nmag[i]*vy_avg[i]/speed[i];
var Frz{i in 1..n-1}
   = -mu*Nmag[i]*vz_avg[i]/speed[i];

minimize finalspeed:
   vx[n-1]^2 + vy[n-1]^2;

s.t. newt_x {i in 1..n-1}:
   ax[i] = (Nx[i] + Frx[i])/m;
s.t. newt_y {i in 1..n-1}:
   ay[i] = (Ny[i] + Fry[i])/m;
s.t. newt_z {i in 1..n-1}:
   az[i] = (Nz[i] + Frz[i] - m*g)/m;

s.t. xinit: x[0] = x0;
s.t. yinit: y[0] = y0;
s.t. zinit: z[0]
   = -0.3*atan(y[0])+0.05*(x[0]+y[0]);

s.t. xfinal: x[n] = xn;
s.t. yfinal: y[n] = yn;
s.t. zfinal: z[n]
   = -0.3*atan(y[n])+0.05*(x[n]+y[n]);

s.t. onthegreen {i in 0..n}:
   x[i]^2 + y[i]^2 <= 16;

let T := 1.5;

let mu := 0.07;
let {i in 0..n}
   y[i] := (i/n)*yn + (1-i/n)*y0;
let {i in 0..n}
   x[i] := y[i]^2/2;

solve;
```

FIGURE 1. A first AMPL model for the putting problem. Note that the variable `v[i]` in the model is the same as `v[i+0.5]` in the text.

But what is wrong? The derivation was straightforward—how could it possibly be wrong?

3. THE CORRECT PUTTING MODEL

The key to understanding what is wrong with our implementation of the Alessandrini model is contained in the observation that

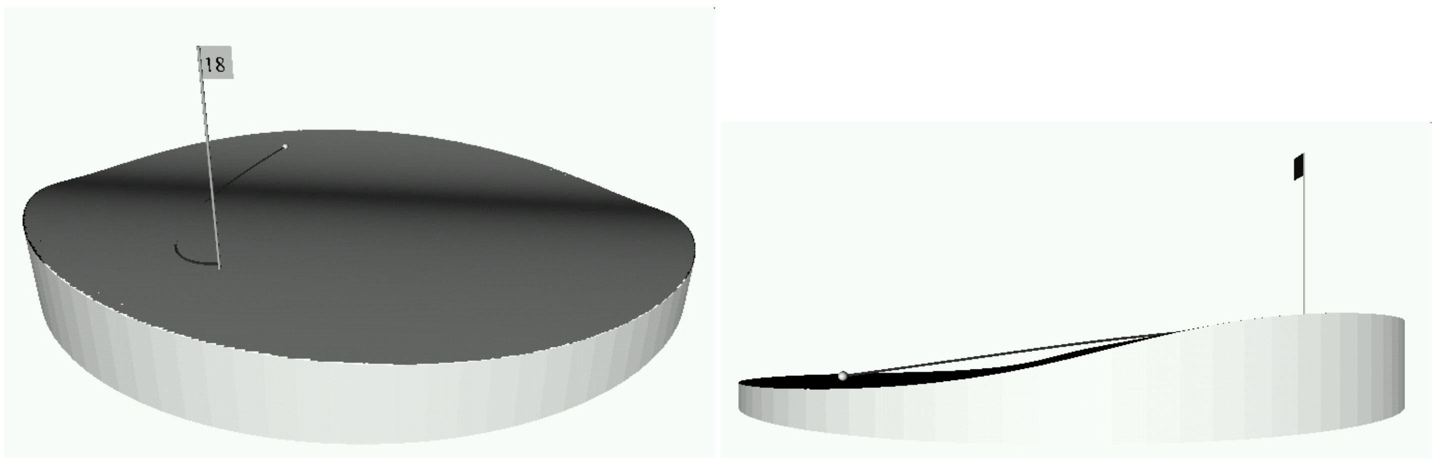

FIGURE 2. Two views of the trajectory obtained from the model in Figure 1 with the elevation constraint removed. *Note: For the online version of this paper, you can click on the figure to start a 3-D animation. In the animation, click on the flag to start the ball rolling.*

the model in Figure 1 is solvable when and only when the surface of the green is planar. This suggests that the derivation is only valid for that case. What is different when the surface is not planar? Well, if you drive a car over the crest of a hill you feel lighter than normal (pun intended), whereas if you speed through a valley you feel heavier. The weight that one feels is the magnitude of the normal force. Hence, the magnitude of the normal force is not constant when the surface has hills and valleys. As you go through a valley, the magnitude of the normal force must be greater than nominal in order to accelerate you along the arc defining the upward bending curve.

From this discussion, it is easy now to see that the magnitude of the normal force must be such that it compensates both for the pull of gravity and for the out-of-tangent-plane acceleration along the path:

$$\|N\| = mg\frac{e_z \cdot N}{\|N\|} + m\frac{a(t)\cdot N}{\|N\|}.$$

From this relation we can deduce that

$$N_z = m\frac{g - a_x(t)\frac{\partial z}{\partial x} - a_y(t)\frac{\partial z}{\partial y} + a_z(t)}{(\partial z/\partial x)^2 + (\partial z/\partial y)^2 + 1}.$$

Everything else in the previous derivation remains the same.

The complete correct model is shown in Figure 3. As shown in Figure 4, this trajectory does indeed follow the surface correctly (as it must given the model).

4. TRAPEZOIDAL DISCRETIZATION

The second common discretization technique is called the *trapezoidal method*. With this method, values for velocity and acceleration are defined at the same discrete times as for position; that is, at `jT/N`, `j=0,1,...,N`. Instead of giving a formula defining each component of velocity in terms of a difference of the corresponding component of position, we give constraints that say that the average value at two adjacent times is equal to the appropriate difference:

```
(vx[i]+vx[i-1])/2 = (x[i]-x[i-1])/(T/n);
(vy[i]+vy[i-1])/2 = (y[i]-y[i-1])/(T/n);
(vz[i]+vz[i-1])/2 = (z[i]-z[i-1])/(T/n);
```

Constraints that must be satisfied by the components of acceleration are similar:

```
(ax[i]+ax[i-1])/2 = (vx[i]-vx[i-1])/(T/n);
(ay[i]+ay[i-1])/2 = (vy[i]-vy[i-1])/(T/n);
(az[i]+az[i-1])/2 = (vz[i]-vz[i-1])/(T/n);
```

The AMPL model for the trapezoidal discretization using the correct formulation of the putting problem is shown in its entirety in Figure 5. Both SNOPT and LOQO solve this formulation of the problem but each takes about twice as long as when solving the corresponding midpoint discretization formulation. Furthermore, LOQO requires a slight relaxation in the stopping criteria (the infeasibility tolerance needs to be increased from its default of 10^{-6} to 2×10^{-5}).

The fact that LOQO requires a relaxation in the stopping rule suggests that something might be wrong with the model. John Betts [3] seems to have identified the issue. He points out that the speed of the ball as it arrives at the cup is zero and hence there is a singularity in the differential equation at the final time. Of course,

```
param g := 9.8; # acc due to gravity
param m := 0.01; # mass of a golf ball
param x0 :=  1; # coords of start pt
param y0 :=  2;
param xn :=  1; # coords of ending pt
param yn := -2;
param n := 50; # num of time points
param mu;

var T >= 0; # total time for the putt
var x{0..n};  # coords of the traj
var y{0..n};

var z   {i in 0..n}
   = -0.3*atan(y[i])+0.05*(x[i]+y[i]);
var dzdx{i in 0..n}
   = 0.05;
var dzdy{i in 0..n}
   = -0.3/(1+y[i]ˆ2) + 0.05;

# v[i] denotes the deriv at midpt of
# the interval i(T/n) to (i+1)(T/n).
var vx{i in 0..n-1} = (x[i+1]-x[i])*n/T;
var vy{i in 0..n-1} = (y[i+1]-y[i])*n/T;
var vz{i in 0..n-1} = (z[i+1]-z[i])*n/T;

# a[i] denotes the accel at the midpt of
# the interval (i-0.5)(T/n)
# to (i+0.5)(T/n), i.e. at i(T/n).
var ax{i in 1..n-1} = (vx[i]-vx[i-1])*n/T;
var ay{i in 1..n-1} = (vy[i]-vy[i-1])*n/T;
var az{i in 1..n-1} = (vz[i]-vz[i-1])*n/T;

var Nz{i in 1..n-1}
   = m*
     (g-ax[i]*dzdx[i]-ay[i]*dzdy[i]+az[i])
     /(dzdx[i]ˆ2 + dzdy[i]ˆ2 + 1);
var Nx{i in 1..n-1} = -dzdx[i]*Nz[i];
var Ny{i in 1..n-1} = -dzdy[i]*Nz[i];
var Nmag{i in 1..n-1}
   = m*
     (g-ax[i]*dzdx[i]-ay[i]*dzdy[i]+az[i])
     /sqrt(dzdx[i]ˆ2 + dzdy[i]ˆ2 + 1);
```

```
var vx_avg{i in 1..n-1} = (vx[i]+vx[i-1])/2;
var vy_avg{i in 1..n-1} = (vy[i]+vy[i-1])/2;
var vz_avg{i in 1..n-1} = (vz[i]+vz[i-1])/2;

var speed{i in 1..n-1}
   = sqrt(vx_avg[i]ˆ2 + vy_avg[i]ˆ2
                      + vz_avg[i]ˆ2);

var Frx{i in 1..n-1}
   = -mu*Nmag[i]*vx_avg[i]/speed[i];
var Fry{i in 1..n-1}
   = -mu*Nmag[i]*vy_avg[i]/speed[i];
var Frz{i in 1..n-1}
   = -mu*Nmag[i]*vz_avg[i]/speed[i];

minimize finalspeed:
   vx[n-1]ˆ2 + vy[n-1]ˆ2;

s.t. newt_x {i in 1..n-1}:
   ax[i] = (Nx[i] + Frx[i])/m;
s.t. newt_y {i in 1..n-1}:
   ay[i] = (Ny[i] + Fry[i])/m;

s.t. xinit: x[0] = x0;
s.t. yinit: y[0] = y0;

s.t. xfinal: x[n] = xn;
s.t. yfinal: y[n] = yn;

s.t. onthegreen {i in 0..n}:
   x[i]ˆ2 + y[i]ˆ2 <= 16;

let T := 1.5;

let mu := 0.07;
let {i in 0..n} y[i] := (i/n)*yn + (1-i/n)*y0;
let {i in 0..n} x[i] := y[i]ˆ2/2;

solve;
```

FIGURE 3. A second, and this time correct, AMPL model for the putting problem. This version is very similar to before—the main difference is in the definitions of `Nz` and `Nmag`.

a numerical approximation might never experience the singularity exactly but it still can feel the effect. For the problem at hand, at the optimal solution LOQO has `speed[n] = 2.6e-6` and SNOPT has `speed[n] = 1.7e-6`. These values are not zero but they are getting close and one could imagine that numerical issues related to the singularity of the differential equation are beginning to enter in here. To test this, we changed the optimization objective from minimizing the final speed to minimizing the deviation of the final speed from some small prescribed value. In particular, we tried `(vx[n]ˆ2 + vy[n]ˆ2 - 0.25)ˆ2`. With this objective function, both solvers are able to find a solution in a much more robust fashion (i.e., using fewer iterations and being successful over a wider range of choice of some of the other parameters

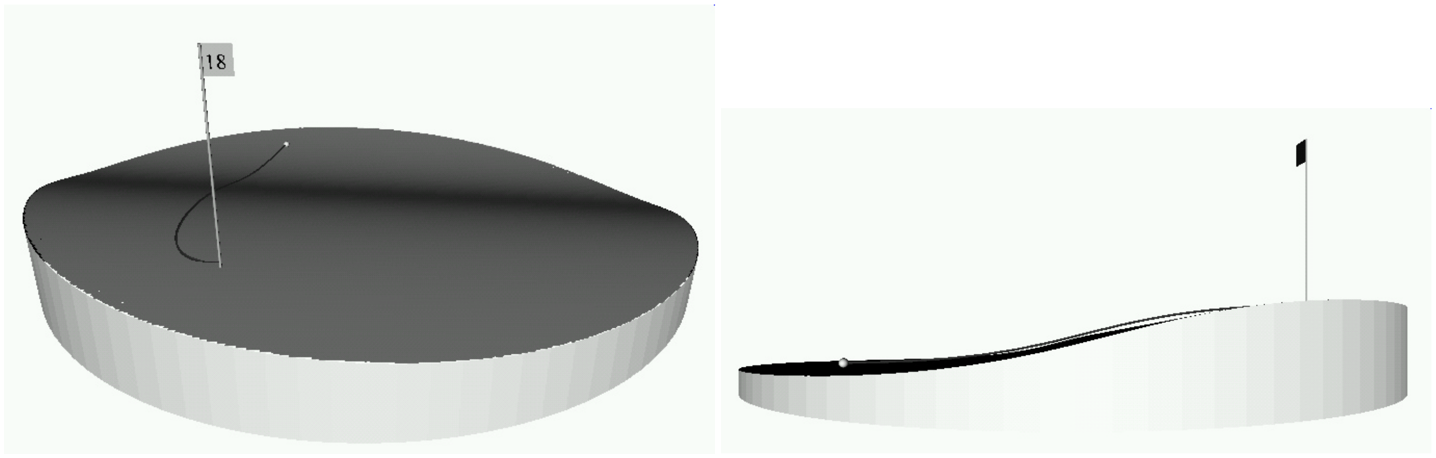

FIGURE 4. Two views of the trajectory from the correct model shown in Figure 3. Note how the trajectory follows the contour of the green.

in the problem). Our local golf expert (aka John Mulvey) indicates that this is the objective function used by real golfers anyway. He says that a real golfer does not want the ball to arrive at the cup with too little speed because then small imperfections in the green can have rather large unpredictable effects in those last few inches near the cup.

It is interesting to note that the midpoint rule is "less" bothered by the singularity issue. The reason is that the final speed in that model is the average final speed over the last time interval. This number is small but not as small as the final speed in the trapezoidal rule. For example, LOQO gets a final speed of `7e-3` with this discretization, which is a few orders of magnitude larger than it got with the trapezoidal rule.

5. LESSONS

(1) It is deceptively easy to formulate a problem incorrectly.
(2) Incorrect formulations are surprisingly likely to be infeasible.
(3) Infeasibility is especially hard for nonlinear solvers to detect reliably.
(4) In the early days of optimization, a nonconvex problem with 10 or more variables was considered exceedingly hard to solve. In its most compact form, the problem here only really has 2 decision variables: the x and y components of the initial velocity vector that the putter imparts to the golf ball. After giving the ball its initial kick, the rest is determined by physics. One could formulate the problem this way. There would be just two decision variables and there would be a fairly complicated integrator function that would determine if the trajectory actually arrives at the hole and, if it does, the speed at which it arrives there. Using this integrator function as a "black box", one could make an optimization problem with just two variables. However, with modern optimization technology it is easy to incorporate the physics into the optimization model as we have done here and get a much larger model but one that is not any more difficult to solve. In fact, by expressing both the optimization part of the model and the physics in the same place and using the same "language" provides a level of model control that was totally lacking before. For example, if the physics is wrong, as it was in our first attempt, then the optimization problem is likely to be infeasible. If the physics and the optimization are separated from each other it is especially hard to identify what (or who!) is at fault. By having them together, it is easy to print out variables, trajectories, dual variables, etc. and all of this information can be useful in figuring out what is wrong with a model.
(5) It wasn't mentioned in the discussion above, but one of the lessons in this example is how important it is to give an initial solution that is close to the optimal solution. For example, the optimal value of T is close to 2 in the examples above. We initialized T to be 1.5. Both LOQO and SNOPT find the right solution for any value of T between 1 and 3 but outside this range the solvers start to get into trouble. For example, neither of the solvers was able to solve the problem when initialized with $T = 5$.

```
param g := 9.8; # acc due to gravity
param m := 0.01; # mass of a golf ball
param x0 :=  1; # coords of start pt
param y0 :=  2;
param xn :=  1; # coords of ending pt
param yn := -2;
param n := 50; # num of time points
param mu;

var T >= 0; # total time for the putt
var x{0..n};  # coords of the traj
var y{0..n};

var z   {i in 0..n}
   = -0.3*atan(y[i])+0.05*(x[i]+y[i]);
var dzdx{i in 0..n}
   = 0.05;
var dzdy{i in 0..n}
   = -0.3/(1+y[i]ˆ2) + 0.05;

var vx{i in 0..n};
var vy{i in 0..n};
var vz{i in 0..n};

var ax{i in 0..n};
var ay{i in 0..n};
var az{i in 0..n};

var Nz{i in 0..n}
   = m*
     (g-ax[i]*dzdx[i]-ay[i]*dzdy[i]+az[i])
     /(dzdx[i]ˆ2 + dzdy[i]ˆ2 + 1);
var Nx{i in 0..n} = -dzdx[i]*Nz[i];
var Ny{i in 0..n} = -dzdy[i]*Nz[i];
var Nmag{i in 0..n}
   = m*
     (g-ax[i]*dzdx[i]-ay[i]*dzdy[i]+az[i])
     /sqrt(dzdx[i]ˆ2 + dzdy[i]ˆ2 + 1);

var speed{i in 0..n}
   = sqrt(vx[i]ˆ2 + vy[i]ˆ2 + vz[i]ˆ2);
```

```
var Frx{i in 0..n}
   = -mu*Nmag[i]*vx[i]/speed[i];
var Fry{i in 0..n}
   = -mu*Nmag[i]*vy[i]/speed[i];
var Frz{i in 0..n}
   = -mu*Nmag[i]*vz[i]/speed[i];

minimize finalspeed: vx[n]ˆ2 + vy[n]ˆ2;

s.t. vx_def {i in 1..n}:
   (vx[i]+vx[i-1])/2=(x[i]-x[i-1])/(T/n);
s.t. vy_def {i in 1..n}:
   (vy[i]+vy[i-1])/2=(y[i]-y[i-1])/(T/n);
s.t. vz_def {i in 1..n}:
   (vz[i]+vz[i-1])/2=(z[i]-z[i-1])/(T/n);

s.t. ax_def {i in 1..n}:
   (ax[i]+ax[i-1])/2=(vx[i]-vx[i-1])/(T/n);
s.t. ay_def {i in 1..n}:
   (ay[i]+ay[i-1])/2=(vy[i]-vy[i-1])/(T/n);
s.t. az_def {i in 1..n}:
   (az[i]+az[i-1])/2=(vz[i]-vz[i-1])/(T/n);

s.t. newt_x {i in 0..n}: ax[i] = (Nx[i] + Frx[i])/m;
s.t. newt_y {i in 0..n}: ay[i] = (Ny[i] + Fry[i])/m;

s.t. xinit: x[0] = x0;
s.t. yinit: y[0] = y0;
s.t. xfinal: x[n] = xn;
s.t. yfinal: y[n] = yn;

s.t. onthegreen {i in 0..n}:
   x[i]ˆ2 + y[i]ˆ2 <= 16;

let T := 1.5;

let {i in 0..n} x[i] := (i/n)*xn + (1-i/n)*x0;
let {i in 0..n} y[i] := (i/n)*yn + (1-i/n)*y0;
let {i in 0..n} vx[i] := (xn-x0)/T;
let {i in 0..n} vy[i] := (yn-y0)/T;

let mu := 0.07;

solve;
```

FIGURE 5. The correct putting model with a trapezoidal discretization. Note how positions, velocities, and accelerations are all defined over the same index set.

6. FINAL REMARKS

We have considered just one trajectory optimization problem—the putting problem. With this problem a number of issues came up that needed to be resolved. It turns out that these same issues are common in trajectory optimization problems. Hence, this example serves as a good prototype for trajectory optimization in general.

Finally, note that in preparing this case study we contacted Stephen Alessandrini to ask him about the fact that his model was incorrect. It turns out that when he derived the equations for his model his interest was in the planar case. It was only at the final stages of writing that he added a nonplanar example and he didn't realize the equations didn't apply. Interestingly, it was this last example that caught the eye of others, see for example [3], and for a time the incorrect model propogated unchecked.

This problem is not purely an academic exercise. See [6] for a description of a system in which putting trajectories were used for real-time animation during television coverage.

REFERENCES

[1] S.M. Alessandrini. A motivational example for the numerical solution of two-point boundary-value problems. *SIAM Review*, 37(3):423–427, 1995. 1

[2] H.Y. Benson, D.F. Shanno, and R.J. Vanderbei. Interior-Point Methods for Nonconvex Nonlinear Programming: Jamming and Comparative Numerical Testing. Technical Report ORFE-00-2, Dept. of Operations Research and Financial Engineering, Princeton University, Princeton NJ, 2000. 1

[3] J.T. Betts. *Practical Methods for Optimal Control using Nonlinear Programming*. SIAM, Philadelphia, PA, 2000. 4, 8

[4] R. Fourer, D.M. Gay, and B.W. Kernighan. *AMPL: A Modeling Language for Mathematical Programming*. Scientific Press, 1993. 1

[5] P.E. Gill, W. Murray, and M.A. Saunders. User's guide for SNOPT 5.3: A Fortran package for large-scale nonlinear programming. Technical report, Systems Optimization Laboratory, Stanford University, Stanford, CA, 1997. 1

[6] W.E. Lorensen and B. Yamrom. Golf green visualization. *IEEE Computer Graphics Appl.*, 12:35–44, 1992. 8

[7] R.J. Vanderbei. LOQO: An interior point code for quadratic programming. *Optimization Methods and Software*, 12:451–484, 1999. 1

[8] R.J. Vanderbei. LOQO user's manual—version 3.10. *Optimization Methods and Software*, 12:485–514, 1999. 1

[9] R.J. Vanderbei. Interior-Point Methods for Nonconvex Nonlinear Programming: Jamming and Comparative Numerical Testing. Technical Report ORFE-00-3, Dept. of Operations Research and Financial Engineering, Princeton University, Princeton NJ, 2000. 1

[10] R.J. Vanderbei and D.F. Shanno. An interior-point algorithm for nonconvex nonlinear programming. *Computational Optimization and Applications*, 13:231–252, 1999. 1

ROBERT J. VANDERBEI, PRINCETON UNIVERSITY, PRINCETON, NJ