THE TIME ELAPSED FOR AN OBJECT SLIDING DOWN AN ARBITRARY PATH

Dr. Felino G. Pascual
Winona State University
Winona, Minnesota 55987
USA
fpascual@vax2.winona.msus.edu

I. Introduction

This presentation is about a project that I had given some of my students including Chris Hornisch and Muhammad Chowdhury from the past few years while teaching at Winona State University. After giving calculus and numerical analysis students a sequence of different small projects involving technology, it occurred to me that, out of a collection of these mini-projects, we could have one single project for numerical analysis students that would highlight different numerical techniques for finding approximate solutions for different classes of mathematical problems. The plan was for an exercise that would make the students work with modeling and take advantage of technology. I had wanted to include modeling as one of the tasks to help reinforce the notion that actual physical applications do provide a backdrop for certain mathematical ideas. On the whole, this is not a project I would have required had we not the computing tools that we have today.

For this project, we have used the problem of finding the elapsed time on slides as its setting. Haws and Kiser wrote a paper on their experimentations with their students on the cycloid and other curves in 1995 [Haws and Kiser]. They showed that the cycloid given by

(1) $x_c(\theta) = \rho(\theta - \sin\theta)$ and $y_c(\theta) = \rho(1 - \cos\theta)$

was the fastest frictionless slide from the origin to the point $(\pi,-2)$. In addition, they provided the analytical solution for the time elapsed for a parametrized curve. Haws and Kiser proceeded to discuss a family of "generalized" cycloids given by

(2) $x(\theta) = x_c(\theta) + \mu\rho(1 - \cos\theta)$ and $y(\theta) = y_c(\theta) + \mu\rho(\theta - \cos\theta)$

which were aimed at the problem with normal friction. Other families of curves that they considered were n^{th}-root curves given by $y = -2(x/\pi)^{1/n}$ a and parabolas.

What follows is the set up of the initial value problem for the trajectory of the sliding object on the curve with normal friction factored in. In addition we will point

out different numerical techniques that we used to complete different tasks or phases project.

II. The Set-up and the Solution

Let C be a smooth curve starting at the origin and ending at $(\pi,-2)$. The particle must follow the prescribed curve C which means no skipping or falling off the curve by the particle. In addition, there will be a normal component to acceleration. Hence, if the acceleration of the particle is $\boldsymbol{a} = (d^2x/dt^2, d^2y/dt^2)$, then $\boldsymbol{a} = a_{\boldsymbol{T}}\boldsymbol{T} + a_{\boldsymbol{N}}\boldsymbol{N}$.

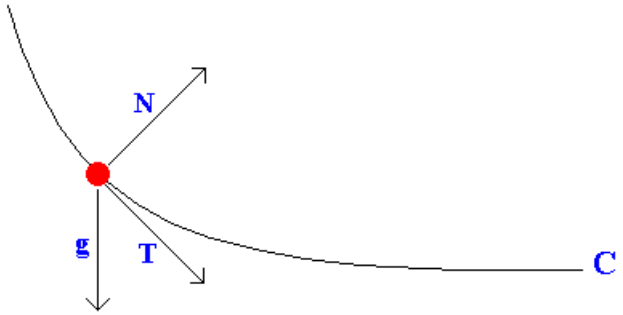

Fig. 2

Without friction, acceleration will be due to gravity only. Since the tangential component a of acceleration will be the scalar projection of gravitational acceleration $\boldsymbol{g}$, we have $\boldsymbol{a} = \boldsymbol{g} \cdot \boldsymbol{T}$. For simplicity, let $\boldsymbol{g} = (0,-1)$. If $\boldsymbol{T} = (1,dy/dx)/\sqrt{1 + (dy/dx)^2}$, then

$$(3) \qquad a_{\boldsymbol{T}} = \frac{-dy/dx}{\sqrt{1 + (dy/dx)^2}} .$$

Let

$$(4) \qquad v = \sqrt{(dx/dt)^2 + (dy/dt)^2} .$$

Then, $a = v^2/\rho$ where ρ = radius of curvature at the point (x,y) on C. Here,

$$(5) \qquad \rho = \frac{\left(1 + (dy/dx)^2\right)^{3/2}}{\left|d^2y/dx^2\right|} .$$

In [Haws and Kiser], friction was given as: $\boldsymbol{F}_{friction} = -\mu(\boldsymbol{F} \cdot \boldsymbol{n})\boldsymbol{T}$ where μ is a positive constant and $\boldsymbol{n}$ = unit normal to the curve such that the nonnegative angle θ between $\boldsymbol{n}$ and $\boldsymbol{g}$ is no bigger than $\pi/2$. With $\boldsymbol{F} = \boldsymbol{g}$, for a curve given by $f(x) = y$, we have

(6) $$\boldsymbol{F}_{friction} = \mu \boldsymbol{g} \cdot (f'(x),-1)(1, f'(x))/\sqrt{1 + (f'(x))^2} = -\mu(1, f'(x))/\sqrt{1 + (f'(x))^2}\ .$$

For a parametric curve, $\boldsymbol{F}_{friction} = \mu \boldsymbol{g} \cdot \left(\frac{dy}{dr}, -\frac{dx}{dr}\right)\left(\frac{dx}{dr}, \frac{dy}{dr}\right)/\sqrt{(dx/dr)^2 + (dy/dr)^2} =$

$$= -\mu \frac{dx}{dr}\left(\frac{dx}{dr}, \frac{dy}{dr}\right)/\sqrt{(dx/dr)^2 + (dy/dr)^2}$$

Combining all of these results and choosing the unit normal $\boldsymbol{n}$ to equal $\left(\frac{dy}{dr}, -\frac{dx}{dr}\right)/\sqrt{(dx/dr)^2 + (dy/dr)^2}$, we have the following initial value problem with friction:

(7)
$$\begin{cases} x''(t) = \dfrac{-(dy/dx)}{1 + (dy/dx)^2} - \dfrac{(dy/dx)(d^2y/dx^2)}{(1 + (dy/dx)^2)^2}\left((x'(t))^2 + (y'(t))^2\right) - \mu \dfrac{(dx/dr)^2}{\sqrt{(dx/dr)^2 + (dy/dr)^2}} \\ y''(t) = \dfrac{-(dy/dx)^2}{1 + (dy/dx)^2} + \dfrac{(d^2y/dx^2)}{(1 + (dy/dx)^2)^2}\left((x'(t))^2 + (y'(t))^2\right) - \mu \dfrac{(dx/dr)\,(dy/dr)}{\sqrt{(dx/dr)^2 + (dy/dr)^2}} \end{cases}.$$

with the conditions that $x(0) = y(0) = x'(0) = y'(0) = 0$. As written above, other forms of damping such as air friction can be incorporated easily in the equations.

While Haws and Kiser provided a Mathematica code called "Race" for their simulations, I required the more advanced students to write their own code to solve for the elapsed time. The students could use the code for the mathematically formulated curve as a blueprint for the code for an arbitrarily drawn curve. For an arbitrary curve such as the one that might have been sketched using CorelDraw or MS Paintbrush or even one that had been drawn by hand, the students will have to find its formulation. Since it is virtually impossible if at all to find the exact mathematical formulation for such a curve, the students used approximate curves such as cubic or B-splines. They used a set of *anchor* or guide points which included the origin and the point $(\pi,-2)$ to generate the approximate spline curve. The current version of *Mathematica* has a feature that allows one to find the coordinates of the anchor points on a graph imported from other applications and, then, find interpolating curves. We decided to write our own code for cubic splines, however, because we wanted to be able to deal with different endpoint conditions such as the ones given in [Gerald and Wheatley]. The reader can refer to [Gerald and Wheatley] and to [Shikin and Plis] for detailed discussions of cubic splines and B-splines. We decided not to use B-splines for reasons that we will mention later. The following figure shows the anchor points A, B, C, D, E, and F along with a *natural* cubic spline and a B-spline "anchored" by the *6* points.

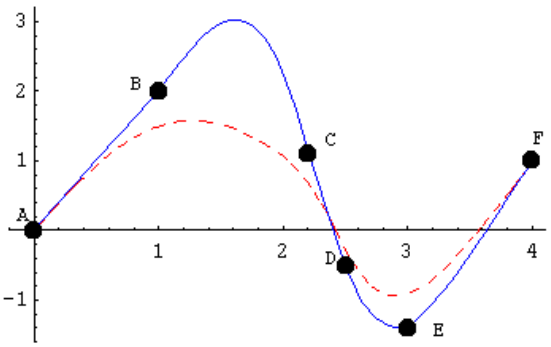

Fig. 2. A natural cubic spline (——) and a B-spline (- - - -)

A natural cubic spline is a spline where the second derivatives at the two endpoints A and F both equal 0.

If we use $n+1$ anchor points with x-coordinates $x_1 \le x_2 \le \ldots \le x_{n+1}$, the cubic spline curve is given by

$$(8) \qquad y(x) = \sum_{k=1}^{n} y_k(x)$$

where $$y_k(x) = \begin{cases} a_k(x - x_k)^3 + b_k(x - x_k)^2 + c_k(x - x_k) + d_k & \text{if } x_k < x \le x_{k+1} \\ 0 & \text{otherwise} \end{cases}.$$

The code for finding the natural cubic spline for a set of anchor points is given below.

```
f[x_]:= Module[{a,i,j,k,b,s,t,equations},
    {Clear[a,b];
     a[1,1]=2.;
     a[1,2]=1.;
     a[d,d]=1.;

     j = 3;
     While[j<d+1,a[1,j]=0.;j++];

     j=1;
     While[j<d,a[d,j]=0.;j++];

     i=2;
     While[i<d,
       {j=1,
        While[j<d+1,
             {a[i,j]=If[i==j,4.,
                      If[i==j-1,1.,
                      If[i==j+1,1.,0.]]];};j++];};i++];
     Clear[b];
     b[xx_][n_Integer]:=If[n==1,0,
```

```
          If[n==d,0,6 xx[[n+1]]-12 xx[[n]]+6 xx[[n-1]]]];
     equations=Table[Sum[a[i,j] t[j],{j,1,d}]==b[x][i],
                 {i,1,d}];
     s=Solve[equations,Evaluate[Table[t[j],{j,i,d}]]];};
     Table[s[[1,k,2]],{k,1,d}]]
sy1=f[yy1];
```

We decided on two different approaches to solving the initial value problem:

- Mathematica's NDSolve feature
- a modified Euler Method for solving initial value problems.

I did not want the students to rely entirely on NDSolve because the focus at this point was on numerical techniques for finding solutions to initial value problems. Nonetheless, we used NDSolve for comparison. The code using NDSolve follows.

```
Clear[gl]
gl[r_]=ylc[n*r/Pi+1.0];
Clear[x,y,xxl,yyl,soln1,s]
grav=-1;
soln1=NDSolve[{x''[t]== grav*gl'[x[t]]/(1+gl'[x[t]]^2) -
          gl''[x[t]] gl'[x[t]](x'[t]^2+y'[t]^2)/(1+gl'[x[t]]^2)^2 - μ/√(1+gl'[x[t]]^2),
          y''[t]== grav*gl'[x[t]]^2/(1+gl'[x[t]]^2) +
          gl''[x[t]](x'[t]^2+y'[t]^2)/(1+gl'[x[t]]^2)^2 - μ gl'[x[t]]/√(1+gl'[x[t]]^2),
          x[0]==0.0,y[0]==0.0,
          x'[0]==0.0,y'[0]==0.0},
        {x[t],y[t]},
        {t,0.0,4}];
xxl[r_]:=soln1[[1,1,2]]/.t->r
```

The quantity `gl[]` is the cubic spline formulation of y in terms of x. The quantities `xxl[]` and `yyl[]` are the numerical solutions for $x(t)$ and $y(t)$. For the modified Euler approach (as with a higher order Runge-Kutta), we reduce the problem to a first-order initial value problem in the following manner. If

(9) $u''(t) = f(t,u,u')$,

then we introduce the variable v where $u' = v$. The differential equation becomes

(10) $v' = f(t,u,v)$ and $u' = v$.

Then, we can find the approximate solution using the following recursion: Let

(11) $u_{predictor} = u_k + v_k \, \Delta t$
$v_{predictor} = v_k + f(t_k , u_k , v_k) \, \Delta t$

Then,

(12) $u_{k+1} = u_k + 0.5(v_k + v_{predictor}) \, \Delta t$
$v_{k+1} = v_k + 0.5(f(t_k , u_k , v_k) + f(t_{k+1} , u_{predictor} , v_{predictor})) \, \Delta t$

The code for the Euler Method given in the Appendix can be easily tailored for a higher order Runge-Kutta approach for a higher degree of accuracy (The modified Euler Method is a *2*nd-order Runge-Kutta method.). In the code given in the Appendix, the variables `xxc` and `yyc` are the predictor values for *x* and *y*. The final values are given by `xxl` and `yyl`. The reader will note that we used "While" loops which allowed us to approximate the elapsed time in the *Mathematica* code without resorting to a root-finding technique. The "guard" statement insures that the execution of the loop stops as soon as $x > \pi$ and the elapsed time should be close to the value of *t* in the last iteration. The code has *3.15* as an approximation for π, but we can use better approximations for more accurate estimates of the elapsed time.

Finding the elapsed time from the NDSolve solution required a different approach. Since we have an actual curve as a solution, we can find the elapsed time using some root-finding technique. Taking advantage of only the continuity of the approximate solution, we used the Bisection Method after we encountered convergence problems when we used "FindRoot" or Newton's Method, especially with graphs imported from other applications. Because of the minimal requirements on the curve by the Bisection Method, it is guaranteed to converge to a particular value of *t*. We can solve for *t* using either of the following equations: $x(t) = 3.14$ or $y(t) = -2$.

Here is the code for finding the elapsed time:

```
a=0.0;
b=3.9;
h[r_]:=xxl[r]-N[Pi]

i=0;
While[i<25&&h[a]h[b]<=0,
      c=.5(a+b);
       a=If[h[a]h[c]<=0,a,c];
       b=If[h[b]h[c]<=0,b,c];
       Print[c];i=i+1;]
```

The endpoints of the time interval over which we will search for the elapsed are given by *a* and *b*. The quantity `xxl[r]` happens to be the numerical solution for *x(t)* obtained by NDSolve. We are then solving for *t* in the equation $x(t) - \pi = 0$.

The following table gives the times for different curves using NDSolve and the Modified Euler and for the cases with or without friction.

Table I. The Elapsed Times for Different Curves Computed Using NDSolve and the Modified Euler Method.

	$\mu = 0.0$		$\mu = 0.001$		$\mu = 0.01$		$\mu = 0.1$	
	NDSolve	*Euler*	*NDSolve*	*Euler*	*NDSolve*	*Euler*	*NDSolve*	*Euler*
Cycloid	*3.14159*	*3.14168*	*3.14359*	*3.14368*	*3.16175*	*3.16178*	*3.35876*	*3.35151*
$y = -2x/\pi$ *Line*	*3.72419*	*3.72419*	*3.72767*	*3.72766*	*3.75936*	*3.75936*	*4.12834*	*4.12834*
$y = \frac{-2\sqrt{(2\pi-x)x}}{\pi}$ *Ellipse*	*3.15338*	*3.15338*	*3.15537*	*3.15334*	*3.17348*	*3.17345*	*3.37496*	*3.37491*
$y = \frac{2x(x-2\pi)}{\pi^2}$ *Parabola*	*3.27634*	*3.27633*	*3.27863*	*3.27863*	*3.29961*	*3.2996*	*3.53455*	*3.53454*
Spline Curve	*3.43536*	*3.33933*	*3.34171*	*3.34171*	*3.46578*	*3.36133*	*3.58028*	*3.58006*

Note that the time calculated by NDSolve for the frictionless ($\mu = 0$) spline curve is a slower time than does the same curve with friction at $\mu = 0.001$ which runs counter to what intuition tells us. The spline curve whose graph is given in the next figure is anchored by the points: *(0,0)*, *(0.15,-0.7)*, *(0.5,-1.5)*, *(1.0,-1.9)*, *(1.5,-1.95)*, *(2,-1.97)*, and *(π,-2)*.

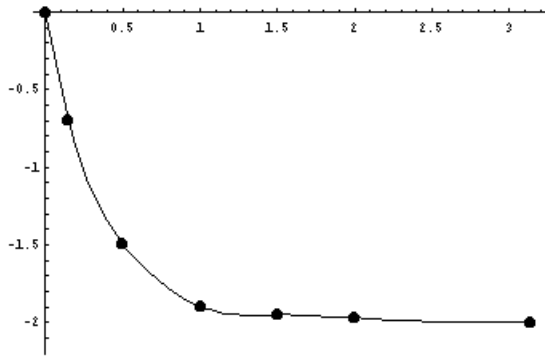

Fig. 3

III. The Student Experience

1. I had introduced the idea of working with mathematical slides to calculus students who had been exposed to the notion of continuity, smoothness and parametrizations. They were asked to draw their own curve, figure out the anchor points, find the cubic spline parametrization, and solve for the elapsed time. I had first-year calculus students use some of the built-in commands of *Mathematica.* However, by doing so, we obtained puzzling results and encountered occasional convergence problems. In an attempt to avoid some of these difficulties, some of the students and myself wrote our own code such as the one for Bisection.

2. I have had two students work on the full project separately. The modeling itself provided a fertile ground for discussion, and that it helped the students get a better grip on the basic ideas of vector projections, the notion of tangential and normal components of acceleration, parametrizations, and splines. In particular, the students learned some of the advantages and disadvantages of some of types of splines. Because B-splines lie within what is called the convex hull of the anchor points, we had believed that B-splines would not oscillate as much as cubic splines. Considering the nature of the problem, this is a desired quality for our approximating curves since we did not want bumps to appear in the curve out of nowhere. However, the students ran into the problem of converting to $f(x) = y$, so that they had to junk the idea of using B-splines. They learned that using B-splines, while easier to find and perhaps a more sound geometrical choice than cubic splines, do not provide an easy shift from a parametrization of the form $x = x(r)$ and $y = y(r)$ to the form $f(x) = y$. They ended up using cubic splines instead. Having mentioned this, I still would like to see a B-spline approach to the initial value problem with friction.

3. Once the modeling of the problem was accomplished, the rest was writing up the computer code. While having some experience in computer programming helped, I did not think that it was essential for this project. Once given a pseudocode as a blueprint, the students were able to write the corresponding *Mathematica* code. The main point was that the students had to have a good understanding of the methods in order to write up the code correctly. They did encounter minor difficulties in setting up and applying the code especially for NDSolve and Euler's method. Indeterminates such as $0/0$ appeared in the differential equations for the cycloid, for example. Most of these were merely a nuisance and were easily overcome with a little preliminary simplification of the differential equations.

4. My original plan was for freshman calculus students to work with Euler's method and Bisection. The first-year students did use a first-order Euler's method that uses only the tangent to solve for the approximate solution. Working with the set-up of the initial value problem gave me the idea that this might be a worthy project for more advanced students.

5. Finally, I would not have asked students to work on the project with so many iterations had there not been technology to help us. While we might be able to accomplish the task with ordinary scientific calculators (I have not tried it myself.), the time it might have taken would have been better devoted for other more meaningful tasks.

IV. References

1. Curtis F. Gerald, and Patrick O. Wheatley, , *Applied Numerical Analysis*, 6th edition, Addison-Wesley, 1999.Haws,
2. LaDawn Haws and Terry Kiser, Exploring the Brachistochrone Problem, *The American Mathematical Monthly*, Vol. 102, No. 4, April 1995.
3. Leonard I. Holder, James DeFranza, and Jay M. Pasachoff, *Calculus*, 2nd ed., Brooks/Cole Publishing Company, 1994.
4. Eugene V. Shikin and Alexander L. Plis, *Handbook on Splines for the User*, CRC Press, 1995.

V. Appendix

```
Clear[fl,gl]
fl[r_]=r;
gl[r_]=y1c[n*r/Pi+1];
Clear[x,y,xxl,yyl,dx,dy,du,dv,uul,vvl,tx,ty]
gl[r_]=-2/Pi*Sqrt[Pi^2-(r-Pi)^2];
tx[r_]:=If[r==0,0,1/Sqrt[1+gl'[r]^2]]
ty[r_]:=If[r==0,-1,gl'[r]/Sqrt[1+gl'[r]^2]]
grav=-1;

dx[u_,v_,x_,y_]:=u;
dy[u_,v_,x_,y_]:=v;
du[u_,v_,x_,y_]:= grav*gl'[x]/(1+gl'[x]^2) - gl''[x] gl'[x](u^2+v^2)/(1+gl'[x]^2)^2 -
                                                                    μ tx[x];
dv[u_,v_,x_,y_]:= grav*gl'[x]^2/(1+gl'[x]^2) + gl''[x](u^2+v^2)/(1+gl'[x]^2)^2 -
                                                                    μ ty[x];


xxl[0]=0.0;yyl[0]=0.0;
uul[0]=0.0;vvl[0]=0.0;
step=0.01;
i=1;
```

```
While[i<=4.7/step&&xxl[i-1]<3.15,
     {xxc=xxl[i-1]+
              step dx[uul[i-1],vvl[i-1],xxl[i-1],yyl[i-1]];
      yyc=yyl[i-1]+
              step dy[uul[i-1],vvl[i-1],xxl[i-1],yyl[i-1]];
      uuc=uul[i-1]+
              step du[uul[i-1],vvl[i-1],xxl[i-1],yyl[i-1]];
      vvc=vvl[i-1]+
              step dv[uul[i-1],vvl[i-1],xxl[i-1],yyl[i-1]];
      xxl=xxl[i-1]+
              0.5 step (dx[uul[i-1],vvl[i-1,
                          xxl[i-1],yyl[i-1]]+
                        dx[uuc,vvc,xxc,yyc]);
      yyl=yyl[i-1]+
              0.5 step (dy[uul[i-1],vvl[i-1,
                          xxl[i-1],yyl[i-1]]+
                        dy[uuc,vvc,xxc,yyc]);
      uul=uul[i-1]+
              0.5 step (du[uul[i-1],vvl[i-1,
                          xxl[i-1],yyl[i-1]]+
                        du[uuc,vvc,xxc,yyc]);
      vvl=vvl[i-1]+
              0.5 step (dv[uul[i-1],vvl[i-1,
                          xxl[i-1],yyl[i-1]]+
                        dv[uuc,vvc,xxc,yyc]);
     Print[{i*step,xxl[i],yyl[i]}];
     i=i+1;
```