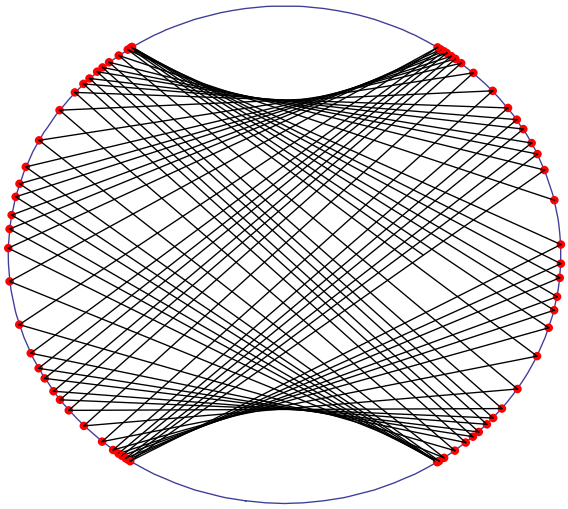
ACM 11: Billiard Ball Project

This is one possible topic for the ACM11 Mathematica project. Assigned December 7 2012. Due by 11:59pm December 14 2012.

Simulate the path of a billiard ball bouncing around inside a convex body, assuming elastic collisions.



It turns out that if you bounce a billiard ball around inside an ellipse, assuming elastic collisions, there are three possibilities: either the path of the ball passes through the foci of the ellipse, or it traces out a hyperbola or ellipse. See `http://www.maa.org/mathland/mathland_3_3.html`. In this project we'll develop a tool for tracing the paths of billiard balls inside of certain bodies.

We will use implicit equations to define the surface of our bodies, for example:

(i) the set of solutions to $x^2 + y^2 - 1 = 0$ is a circle

(ii) the set of solutions to $\left(\frac{x}{2}\right)^2 + \left(\frac{y}{3}\right)^2 - 1 = 0$ is an ellipse

(iii) and the set of solutions to $x^6 + y^4 - 1 = 0$ looks like a rectangle with rounded edges.

More generally, a function f can be used to define a body implicitly by stating that the level set $f(x) = 0$ represents the surface of the body, x such that $f(x) < 0$ are interior to the body, and $f(x) > 0$ means x is outside the body. Of course this definition only makes sense if f is such that the zero level set of f is a single closed figure. We will only consider the above three f, but the following method should work for a wide variety of f.

Our program will consist of two main functions: `rayBoundaryIntersection`, which finds the intersection of a path of the ball (which we'll call a ray) with the surface, and `reflectRay`, which will reflect a ray going toward the surface into a ray representing the path of the ball away from that point of the surface. Rays will be represented as a pair of pairs, where the first represents the origin of the ray and the second represents the positive direction of the ray: e.g. `{{1,0}, Normalize@{-Cos[Pi/3], Sin[Pi/5]}}`.

In `rayBoundaryIntersection` we will call `Solve` to find intersection points: those which satisfy both the implicit equation defining the surface, and the equation defining the ray.

Since the ball will be undergoing elastic collisions, the incoming angle of rays will be equal to their outgoing angles, where we measure the angles with respect to the tangent plane of the surface at the intersection points. The `reflectRay` function will capture this physics. To that end, we will need to know the local tangent and normal vectors at the intersection points; we will use a supplementary function `frenetFrame` to determine these from the implicit equation defining the surface.

Let's summarize how the program will work: the user will specify an implicit function defining the surface, an initial ray starting from a point inside the surface and pointing somewhere else inside the surface, and how many rays to cast (i.e. how many times to track the bouncing of the ball). Then we'll call `reflectRay` the appropriate number of times to determine the path the ball takes. Each time `reflectRay` is called it will itself call `rayBoundaryIntersection` to determine where the ball bounces. `reflectRay` will also be responsible for constructing the graphical representation of the path of the ball.

Here are detailed instructions:

(a) Define a function `implicitform` that takes x and y as coordinates and returns the value of the implicit function defining the surface. For example, for a standard circle, `implicitform[x_,y_] = Power[x,2] + Power[y,2] - 1`. Use `ContourPlot` with the options `Frame->False, AspectRatio->1` to define `boundarycurve`, a plot of the zero level set of the function. Note that you'll need to change the plot range for the ContourPlot when you change `implicitform`, to show the entire zero level set. While you're doing your coding, I recommend using the standard circle. For grading purposes, we will change `implicitform` to one of the above examples (and we'll change the plot range if necessary). We will also change the initial ray `curray` (see below) to a suitable choice.

(b) Define `graphqueue` as an empty list, and a function `addQueue` that takes a single argument and appends it to `graphqueue`. The idea is, that as we go along, we will add the `Line`s representing the ball's path between intersection points and the `Point`s representing the intersection points to `graphqueue`, and show `graphqueue` along with `boundarycurve` as our final visualization.

(c) Define `rayBoundaryIntersection` as a function which takes a ray as its argument, and returns the intersection point of that ray with the surface. Do this by solving the system of equations given by the fact that the intersection point must lie on the ray and must also satisfy `implicitform`. Two issues come to mind: one, you will get a trivial intersection point, because the starting point of the ray satisfies both these equations, also, you might get some non-real solutions. Filter out these two types of false intersection points. If you have more than one intersection point (you may, if the surface is nonconvex, for instance), return the one that is closest, in the positive direction, to the starting point of the ray– that is, the one that the ball would first encounter as it traveled along the ray.

(d) Define `frenetFrame` which takes a point (a pair of numbers) and returns a pair of pairs: the first pair being the tangent to the surface at the point, and the second being the inward pointing normal to the surface at the point. First, determine the normal using the fact that if $f = 0$ defines a body's surface and $f < 0$ defines its interior, then $-\nabla f$ defines the inward pointing normal. If the normal $n = (x, y)$, take the tangent to be $t = (-y, x)$; this will give the local tangent and normal basis a consistent orientation around the surface– i.e. the tangent vector will always point counterclockwise. Ensure, using `Normalize`, that the tangent and normal vectors are unit length.

(e) Define `reflectRay` as a function which accepts a ray as its argument. Inside `reflectRay`, first determine where that ray will intersect the surface, and add this point as a red `Point` to the graphics queue. Also add a `Line` connecting the origin of the ray to the intersection point to the graphics queue. Next find the tangent t and normal n at the intersection point, and return

the ray reflected about the intersection point as follows: make its origin the intersection point, and if the direction of the incoming ray was $at+bn$ (use `Dot` to determine a and b), the direction of the outgoing ray is $at - bn$.

(f) Put all of the above definitions in one cell (appropriately commented and spaced and formatted for readibility!). In the cell under it, we will do the visualization. First define the variable `curray` as a ray which starts somewhere inside the surface and points inside the surface; e.g., for any of the example surfaces given, `curray = {{1,0}, Normalize@{-Cos[Pi/3], Sin[Pi/5]}} //N` is a valid choice.

Make sure that the entries of `curray` are approximate numbers, not exact, as in the example `curray` given. The reason is that, if the starting ray contains exact numbers, Mathematica will try to find exact solutions each time `rayBoundaryIntersection` is called; the solutions will quickly get large and unwieldy, and Mathematica will essentially stall. If instead we use approximate numbers, Mathematica will avoid the propagation of large expressions and happily and quickly find inexact solutions.

Use a `For` loop to calculate `reflectRay[curray]` 70 times, each time overwriting `curray`. Finally, display the boundary curve and the contents of the graphics queue.

Document your program with comments and make sure that the TA will have no trouble reading your code; in particular, this entails spacing your code appropriately and using line breaks appropriately.