Simple Three-D Raytrace Algorithm

Tom Murphy

April, 2011

1 Introduction

This document describes the mathematical formulation implemented in the raytracing program that we will use in our optical investigations. This particular algorithm performs the mathematical operations relevant to each surface encounter. A handling program does the book-keeping of initializing rays and propagating them through the system. The algorithm described here simply computes the intersection of a ray with a (generally) curved surface, computes the surface normal, and applies Snell's law to arrive at the new ray position and direction.

2 Defining the Surfaces

Most of the time we will use spherical surfaces, but let's allow aspherical surfaces as well—conic sections in particular. Following conventional practice, we will assume the optical axis of the system to be the z-axis.

A general form for a conic section oriented axially along the z-axis, whose vertex is at z_v and has radius R at that point, is:

$$(z - z_v)^2(K + 1) - 2R(z - z_v) + x^2 + y^2 = 0, \tag{1}$$

where K is the conic constant. A sphere has $K = 0$; $K > 0$ describes oblate ellipsoids; $-1 < K < 0$ describes prolate ellipsoids; $K = -1$ describes a paraboloid; and $K < -1$ describes hyperboloids (see Fig. 1). Note that the only place coordinates appear without the square also has an R. This means that flipping the sign of R also flips the curve along the z-axis. Positive R opens to the right, and negative R opens to the left. See the appendix for how to relate this to more familiar forms for the conic sections.

Since we will eventually want to describe surface normals for the application of Snell's Law, we go ahead and take the derivative here. Because of rotational symmetry, we can simplify our math for the moment by replacing $x^2 + y^2$ in Eq. 1 with the radial measure r^2. We will go after $\frac{\partial z}{\partial r}$, but could also work with $\frac{\partial r}{\partial z}$. To start, we will develop an expression for $z(r)$ by completing the square in $(z - z_v)$. To do this, we first divide Eq. 1 by $(K + 1)$, then add and subtract the constant that will complete the square.

$$(z - z_v)^2 - 2\frac{R}{K+1}(z - z_v) + \frac{r^2}{K+1} = 0$$

$$\left[(z - z_v)^2 - 2\frac{R}{K+1}(z - z_v) + \frac{R^2}{(K+1)^2}\right] - \frac{R^2}{(K+1)^2} + \frac{r^2}{K+1} = 0,$$

where the term in brackets is now a square, so that

$$\left[z - z_v - \frac{R}{K+1}\right]^2 = \frac{R^2}{(K+1)^2} - \frac{r^2}{K+1}. \tag{2}$$

Now it is straightforward to express z as a function of r:

$$z = z_v + \frac{R}{K+1} - \sqrt{\frac{R^2}{(K+1)^2} - \frac{r^2}{K+1}}, \tag{3}$$

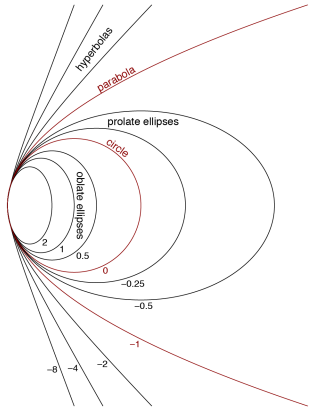

Figure 1: Various forms of conic section as a function of the conic constant, K.

where we have chosen the negative square root to place the curve at $z = z_v$ when $r = 0$. And finally we are in a position to evaluate derivatives:

$$z'(r) = \frac{r}{K+1}\left[\frac{R^2}{(K+1)^2} - \frac{r^2}{K+1}\right]^{-\frac{1}{2}}, \tag{4}$$

and while we're at it, we can check that the second derivative at the vertex (at $r = 0$) indeed gives us a radius of curvature of R. We simplify the derivative by noting that anything with a r out front will vanish in our evaluation at the vertex.

$$z''(r=0) = \frac{1}{K+1}\left[\frac{R^2}{(K+1)^2}\right]^{-\frac{1}{2}} = \frac{1}{R}$$

The predominance of the $K+1$ term in the denominators of these expressions indicate that we should take special care for the parabolic case when $K = -1$. For this, we are better off going back to Eq. 1 and re-deriving relationships in this simplified form.

3 Ray Intersection

We will define the ray segment, labeled by the subscript i as we must deal with multiple segments, as passing through the point $\vec{\rho_i} = (x_i,\, y_i,\, z_i)$, and traveling with unit vector direction $\hat{k_i} = (k_{ix},\, k_{iy},\, k_{iz})$. The default orientation will have the ray traveling from left-to-right ($k_z > 0$).

The ray segment's position (backward or forward) therefore satisfies

$$\vec{\rho} = \vec{\rho_i} + s\hat{k_i}, \tag{5}$$

where s is some pathlength of travel (negative for backwards propagation). The intersection of this ray segment with the optical curve of interest is given by the joint solution of Equation 1, in vector form, and Equation 5. Borrowing from Eq. 3, we have the component-by-component relationship

$$\begin{pmatrix} x_i + sk_{ix} \\ y_i + sk_{iy} \\ z_i + sk_{iz} \end{pmatrix} = \begin{pmatrix} x \\ y \\ z_v + \frac{R}{K+1} - \sqrt{\frac{R^2}{(K+1)^2} - \frac{x^2+y^2}{K+1}} \end{pmatrix}. \tag{6}$$

Now if we substitute expressions for x and y into the z-component equation, we produce a quadratic relationship in the only remaining unknown quantity, s. It is actually simpler—now that we have seen the concept fleshed out in

vector form—to go back to Eq. 1, and write

$$(x_i + sk_{ix})^2 + (y_i + sk_{iy})^2 - 2R(z_i + sk_{iz} - z_v) + (K+1)(z_i + sk_{iz} - z_v)^2 = 0, \tag{7}$$

for which we consolidate terms in s^2, s, and constants:

$$\begin{array}{rcl} s^2\left[k_{ix}^2 + k_{iy}^2 + (K+1)k_{iz}^2\right] & + \\ 2s\left[x_i k_{ix} + y_i k_{iy} - Rk_{iz} + (K+1)(z_i - z_v)k_{iz}\right] & + \\ x_i^2 + y_i^2 - 2R(z_i - z_v) + (K+1)(z_i - z_v)^2 & = & 0 \end{array}$$

If we now divide through by $\left[k_{ix}^2 + k_{iy}^2 + (K+1)k_{iz}^2\right]$ to put in the form $s^2 + 2bs + c = 0$, we have the familiar quadratic solution:

$$s = -b \pm \sqrt{b^2 - c}, \tag{8}$$

with

$$b = \frac{x_i k_{ix} + y_i k_{iy} + \left[(K+1)(z_i - z_v) - R\right] k_{iz}}{k_{ix}^2 + k_{iy}^2 + (K+1)k_{iz}^2}, \tag{9}$$

and

$$c = \frac{x_i^2 + y_i^2 + (K+1)(z_i - z_v)^2 - 2R(z_i - z_v)}{k_{ix}^2 + k_{iy}^2 + (K+1)k_{iz}^2}. \tag{10}$$

Now one simply inserts the solutions for b and c into Equation 8 to find the surface intersections. But notice there are two roots. This should be the case for a line intersecting a sphere, ellipsoid, or hyperboloid (recall there are two arcs): provided there is any non-tangent intersection there will be two. Which one do we want? For $K > -1$ and "forward" propagating rays ($k_{iz} > 0$), we want the most negative root when $R > 0$, and the most positive root when $R < 0$ (see Fig. 1 for visual sense of $R > 0$ case). For hyperboloids, we want the curve whose vertex is at z_v, and not the other curve. When $R > 0$, the other curve will be at more negative values of z, so we must flip the rule established above in the case where $K < -1$.

So the rule becomes:

- if $K > -1$, $s = -b - \frac{R}{|R|}\frac{k_{iz}}{|k_{iz}|}\sqrt{b^2 - c}$;
- if $K < -1$, $s = -b + \frac{R}{|R|}\frac{k_{iz}}{|k_{iz}|}\sqrt{b^2 - c}$,

where the signs of R and k_{iz} are used to pick which root to use. Inserting this solution for s into Eq. 5 gives the vector position of the intersection, $\vec{\rho}_{i+1}$.

Note that the math still works for $R \to \infty$ to represent a planar interface. However, a computer will have numerical roundoff problems, so it is best to detect $R > R_{max}$ and treat such cases as planar. The intersection solution in this case is a trivial calculation of Equation 5, with z_i set to the z-value of the plane.

Also note that in most cases, s should be positive. Provided we really intend light to travel in the direction of $\hat{k_i}$, we should see $s > 0$, which can be used as a diagnostic for the sanity of the ray path.

For the case of the parabola, when $K = -1$, the denominator in Eqs. 9 and 10 will be zero for the common case of light traveling parallel to the z-axis. In this case, we are better off going straight to Eqs. 1 and 5 to find:

$$s_{\text{parabola}} = \frac{x_i^2 + y_i^2 - 2R(z_i - z_v)}{2Rk_{iz}}.$$

4 Surface Normal

Now that we have established the coordinates of the ray intersection, we must determine the angle of the surface normal at this location. For the general case, we can use Equation 4, re-expressed as:

$$z'(r_{i+1}) = \frac{r_{i+1}}{\sqrt{R^2 - (K+1)r_{i+1}^2}}\frac{R}{|R|},$$

where the last factor accommodates different signs for the radius, and the $i+1$ subscript reminds us that this is the point of intersection. In a two-dimensional world of r and z coordinates, the components of the surface normal vector would have a ratio dictated by the rise-over-run ratio z'. At the vertex, where $z' = 0$, the vector should be strictly in the z-direction. So we could generalize the normal vector as lying in the direction $(r,\, z) = (1, 1/z')$. Bringing this argument into three dimensions, we get a (normalized) surface normal:

$$\hat{n} = \frac{1}{\sqrt{1-(K+1)+\frac{R^2}{r_{i+1}^2}}} \begin{pmatrix} \frac{x_{i+1}}{r_{i+1}} \\ \frac{y_{i+1}}{r_{i+1}} \\ -\frac{R}{|R|}\frac{\sqrt{R^2-(K+1)r_{i+1}^2}}{r_{i+1}} \end{pmatrix} = \frac{1}{\sqrt{R^2-Kr_{i+1}^2}} \begin{pmatrix} x_{i+1} \\ y_{i+1} \\ -\frac{R}{|R|}\sqrt{R^2-(K+1)r_{i+1}^2} \end{pmatrix}. \tag{11}$$

This works for all values of K, including the parabolic case.

5 Snell's Law and the Final Ray

How do we apply Snell's law in three dimensions? First, we note that the incident ray vector, $\hat{k}_i$, and the surface normal, $\hat{n}$, define a plane. The only time they do not is if the two are parallel (or anti-parallel), in which case the refraction is trivial: no change in angle. We *could* construct a vector from the cross-product on $\hat{k}_i$ and $\hat{n}$, then rotate our incoming $\hat{k}_i$ through some angle about this vector to get the resulting $\hat{k}_{i+1}$. But we can perhaps keep it simple enough by taking the part of $\hat{n}$ that is perpendicular to $\hat{k}_i$ and then normalizing it. We will call this vector $\hat{\ell}$:

$$\hat{\ell} = \mathrm{norm}(\hat{n} - (\hat{k}_i \cdot \hat{n})\hat{k}_i). \tag{12}$$

It is easy to convince yourself that $\hat{\ell}$ is perpendicular to $\hat{k}_i$ by computing $\hat{\ell} \cdot \hat{k}_i$.

We define the incident angle by

$$\cos\theta_i = |\hat{n} \cdot \hat{k}_i|,$$

from which we compute the (always positive in this convention):

$$\sin\theta_i = \sqrt{1-(\hat{n} \cdot \hat{k}_i)^2}, \tag{13}$$

so that Snell's law,

$$n_i \sin\theta_i = n_{i+1} \sin\theta_{i+1}, \tag{14}$$

is readily applied to give the new angle relative to the normal. We then note that the angle change in $\hat{k}$ is the difference between the incoming and outgoing angles relative to the surface normal:

$$\Delta\theta = \theta_{i+1} - \theta_i. \tag{15}$$

So in order to rotate $\hat{k}_i$ by $\Delta\theta$—knowing that $\hat{k}_i$ and $\hat{\ell}$ form an orthogonal vector plane in the plane of incidence (since $\hat{\ell}$ was constructed out of $\hat{n}$ and $\hat{k}_i$, which together *define* the plane of incidence)—we simply construct:

$$\hat{k}_{i+1} = \hat{k}_i \cos\Delta\theta \pm \hat{\ell} \sin\Delta\theta. \tag{16}$$

The only trick is to know which sign to pick to get the story straight. To get it straight myself, I drew 16 different pictures depicting all combinations of slope, which side of the interface $\hat{n}$ falls on, which side has larger refractive index, and whether the refractive indices have the same sign or not (which can be used to affect reflection). You are welcome to repeat this exercise yourself, but I ended up with the following formulation that works:

$$\hat{k}_{i+1} = \frac{n_i n_{i+1}}{|n_i n_{i+1}|}\left[\hat{k}_i \cos\Delta\theta - \frac{\hat{k}_i \cdot \hat{n}}{|\hat{k}_i \cdot \hat{n}|}\hat{\ell} \sin\Delta\theta\right], \tag{17}$$

where the fraction pieces control the signs, flipping the k-vector around if the refractive index changes sign (reflection), and getting the cross-wise direction right depending on which side of the interface the $\hat{n}$ vector happens to point relative to the incoming $\hat{k}$.

6 Notes on Subtleties

Each intersection is computed without reference to whether the designated "pass-through" point is to the left or right of the surface. An example of how this can be bad is if a bi-convex (positive) lens is too thin, such that the spheres occur in the wrong order far from the optical axis. The ray will blithely intersect them in this wrong order, dutifully following Snell's Law, producing an unphysical ray path. A warning in the algorithm ($s < 0$) can alert you to any ray whose x-intersection jumps backward from one surface to the next.

An example of where this might be beneficial is if you want to define your initial ray at the center of the lens rather than far away, there is no penalty. For example, if I want a ray whose angle to the optical axis is 0.1 radians, and I want this ray to encounter the first lens (at $z = 0$, say) at a height of 5 mm, then I can specify the initial position as (5.0, 0.0, 0.0) with an initial direction of (0.1, 0.0, 1.0), rather than starting the ray at (-5.0, 0.0, -100.0) with the same initial vector. The math is simplified. Likewise, if I want the original ray to head for a specified point to the right of the optic (in the absence of the optic), then I simply specify the ray through that point. The calculation will "backtrack" to the appropriate first-surface intersection, with a warning.

7 Computer Code and Example

The following listing is written in Python, and just passes one ray through the optical system. A more sophisticated handling program is far more useful, but larger than is warranted for this space.

```
#!/usr/bin/env python

from math import *
import sys
import numpy

def vmag(vect):
  return sqrt(numpy.dot(vect,vect))

def vnorm(vect):
  return vect/vmag(vect)

def ray_step_3d(ray_pos, ray_direc, surface):

  # rename ray_pos components for clarity
  x0 = ray_pos[0]
  y0 = ray_pos[1]
  z0 = ray_pos[2]
  init_pos = numpy.array([x0,y0,z0],dtype='d')

  # rename ray_direc components for clarity, and normalize
  kx0 = ray_direc[0]
  ky0 = ray_direc[1]
  kz0 = ray_direc[2]
  k0 = vnorm(numpy.array([kx0,ky0,kz0],dtype='d'))

  # rename surface params for clarity
  n0 = surface[0]
  z_v = surface[1]
  R = surface[2]
  K = surface[3]
  n_new = surface[4]

  # establish sign flippy dealies
  Rsign = R/fabs(R)
  Ksign = 1.0
  if (K < -1.0):
    Ksign = -1.0
```

```
  direc_sign = kz0/fabs(kz0)

  # solve intersection
  denom = kx0*kx0 + ky0*ky0
  if (K == -1.0 and denom == 0.0):              # parabolic case, straight in
    s = (x0*x0 + y0*y0 - 2*R*(z0 - z_v))/(2*R*kz0)
  else:
    denom = kx0*kx0 + ky0*ky0 + (K+1)*kz0*kz0
    b = (x0*kx0 + y0*ky0 + ((K+1)*(z0-z_v) - R)*kz0)/denom
    c = (x0*x0 + y0*y0 + (K+1)*(z0*z0-2*z0*z_v+z_v*z_v) - 2*R*(z0-z_v))/denom
    s = -b - direc_sign*Rsign*Ksign*sqrt(b*b - c)

  if (fabs(R) > 1.0e10):        # assume exactly planar
    s = (z_v - z0)/kz0

  if (s < 0.0):
    print "WARNING: ray jumped backwards!"

  # ray/surface intersection position
  new_pos = init_pos + s*k0
  xi = new_pos[0]
  yi = new_pos[1]
  zi = new_pos[2]
  ri2 = xi*xi + yi*yi

  # surface normal
  nhat = vnorm(numpy.array([xi,yi,-Rsign*sqrt(R*R - (K+1)*ri2)],dtype='d'))

  if (fabs(R) > 1.0e10):        # planar case
    nhat = numpy.array([0.0,0.0,1.0],dtype='d')

  # establish delta-theta
  ndotk = numpy.dot(nhat,k0)
  sin_thet_in = sqrt(1.0 - ndotk*ndotk)
  thet_in = asin(sin_thet_in)
  thet_out = asin(n0*sin_thet_in/n_new)
  dthet = thet_out - thet_in

  # establish l-vector perp to k-vect and in plane of incidence
  lvect = nhat - ndotk*k0
  if (vmag(lvect) > 1.0e-6):
    lhat = vnorm(lvect)
  else:
    lhat = numpy.array([0.0,0.0,0.0],dtype='d')

  # get sign flips
  dnsign = n0*n_new/fabs(n0*n_new)
  knsign = -ndotk/fabs(ndotk)

  k_new = dnsign*(cos(dthet)*k0 + knsign*sin(dthet)*lhat)

  return (new_pos,k_new)

#main handling program
narg = len(sys.argv)

x = []
y = []
z = []
kx = []
ky = []
```

```
kz = []
n = []
z_vert = [0.0]
R = [0.0]
K = [0.0]
surf_params = [0.0]*5


screen_pos=0.0

if (narg > 7):                  # must have at least these four
  filename = sys.argv[1]
  x.append(float(sys.argv[2]))
  y.append(float(sys.argv[3]))
  z.append(float(sys.argv[4]))
  kx.append(float(sys.argv[5]))
  ky.append(float(sys.argv[6]))
  kz.append(float(sys.argv[7]))
else:
  print "Must supply lens_file_name x0, y0, z0, kx0, ky0, kz0 arguments"
  sys.exit()

if (narg > 8):          # optionally, put a screen somewhere
  screen_pos = float(sys.argv[8])

lens_file = open(filename,'r'); # grab lens surface parameters
n_surf = int(lens_file.readline().strip())      # number of surfaces (1st line)
n.append(float(lens_file.readline().strip()))   # initial refr. index (2ndline)
current_z = 0.0;
for i in range(n_surf):                 # and n_surf additional lines...
  # read in lens file and verify results
  line = lens_file.readline()
  Slist = line.split()
  n.append(float(Slist[0]))
  z_vert.append(float(Slist[1]))
  R.append(float(Slist[2]))
  K.append(float(Slist[3]))
  current_z += z_vert[i+1]
  print "Surface %d has n = %f, z_vert = %f, radius = %g, K = %f" % \
         (i+1,n[i+1],current_z,R[i+1],K[i+1])

lens_file.close()

print "Ray 1 has x, k = (%f,%f,%f), (%f,%f,%f)" % \
        (x[0],y[0],z[0],kx[0],ky[0],kz[0])

current_z = 0.0
for i in range(n_surf):         # now propagate surface-at-a-time
                                # begin ray propagation for loop
  # populate surface parameters array
  current_z += z_vert[i+1]
  surf_params[0] = n[i]
  surf_params[1] = current_z
  surf_params[2] = R[i+1]
  surf_params[3] = K[i+1]
  surf_params[4] = n[i+1]

  # populate in_ray array for +y ray
  ray_pos = numpy.array([x[i],y[i],z[i]],dtype='d')
  ray_direc = vnorm(numpy.array([kx[i],ky[i],kz[i]],dtype='d'))

  # carry out calculation
  out_pos,out_direc = ray_step_3d(ray_pos,ray_direc,surf_params)
```

```
    # stow out_ray into approp. arrays
    x.append(out_pos[0])
    y.append(out_pos[1])
    z.append(out_pos[2])
    kx.append(out_direc[0])
    ky.append(out_direc[1])
    kz.append(out_direc[2])

    print "Ray %d has x, k = (%f,%f,%f), (%f,%f,%f)" % \
          (i+2,x[i+1],y[i+1],z[i+1],kx[i+1],ky[i+1],kz[i+1])

# compute intercepts
s_screen = (screen_pos - z[n_surf])/kz[n_surf]
x_screen = x[n_surf] + s_screen*kx[n_surf]
y_screen = y[n_surf] + s_screen*ky[n_surf]

s_zy = -x[n_surf]/kx[n_surf]
y_zy = y[n_surf] + s_zy*ky[n_surf]
z_zy = z[n_surf] + s_zy*kz[n_surf]

s_zx = -y[n_surf]/ky[n_surf]
x_zx = x[n_surf] + s_zx*kx[n_surf]
z_zx = z[n_surf] + s_zx*kz[n_surf]

print "Ray intercepts screen at (%f, %f, %f)" % (x_screen,y_screen,screen_pos)
print "Ray intercepts z-y plane at (%f, %f, %f)" % (0.0,y_zy,z_zy)
print "Ray intercepts z-x plane at (%f, %f, %f)" % (x_zx,0.0,z_zx)
```

7.1 Example Input File

The input file for a beam expander looks like:

```
4
1.0
1.5 -1.5 -100.0 0.0
1.0  3.0  1.0e20 0.0
1.5 198.5 1.0e20 0.0
1.0 4.0 -201.55 0.0
```

The first line indicates the number of surfaces. The second line indicates the initial refractive index. Each following line represents a surface. For each, the fields are as follows:

- the refractive index that one passes into by traversing the surface
- The z-displacement from the last vertex (initialized to zero)
- the radius of curvature: positive means center of curvature is to the right
- the K-constant describing the shape

In this example, the vertices lie at z-values of -1.5, 1.5, 200.0, and 204.0, respectively (the numbers in the file represent separation from the last vertex). The first lens is a negative lens, and the second a positive lens. Surfaces are either spherical (all have $K = 0.0$) or planar (huge R).

7.2 Example Output

Example output of the code looks like the following:

```
./one_3d.py beam_expander 3.0 4.0 -10.0 0.0 0.0 1.0 300.0

Surface 1 has n = 1.500000, z_vert = -1.500000, radius = -100, K = 0.000000
Surface 2 has n = 1.000000, z_vert = 1.500000, radius = 1e+20, K = 0.000000
Surface 3 has n = 1.500000, z_vert = 200.000000, radius = 1e+20, K = 0.000000
Surface 4 has n = 1.000000, z_vert = 204.000000, radius = -201.55, K = 0.000000
Ray 1 has x, k = (3.000000,4.000000,-10.000000), (0.000000,0.000000,1.000000)
Ray 2 has x, k = (3.000000,4.000000,-1.625078), (0.010008,0.013344,0.999861)
Ray 3 has x, k = (3.031281,4.041708,1.500000), (0.015013,0.020017,0.999687)
Ray 4 has x, k = (6.012199,8.016265,200.000000), (0.010008,0.013344,0.999861)
Ray 5 has x, k = (6.049712,8.066282,203.747637), (-0.000008,-0.000011,1.000000)
Ray intercepts screen at (6.048942, 8.065256, 300.000000)
Ray intercepts z-y plane at (0.000000, -0.000000, 756938.486428)
Ray intercepts z-x plane at (0.000000, 0.000000, 756938.486428)
```

The ray starts at (3, 4, -10), initially horizontal. A screen has been placed at $z = 300$. The final ray leaves the lens at (6.049712, 8.066282, 203.747637) at an angle of -0.000014 radians (combining x and y tangents in quadrature). This ray will cross the z-axis at $z = 756938.5$ (far away: nearly level).

Appendix: Relating to Familiar Conic Forms

Starting with Eq. 1:

$$(z - z_v)^2(K + 1) - 2R(z - z_v) + r^2 = 0,$$

where r is the radial offset from the axis, and completing the square as we did before for Eq. 2, we end up with

$$\left[z - z_v - \frac{R}{K+1}\right]^2 + \frac{r^2}{K+1} = \frac{R^2}{(K+1)^2}.$$

Now for the sake of two-dimensional familiarity, we will rename the term in brackets x, which is just a shift of the coordinate origin along the z-axis, followed by a renaming of the z-axis to the x-axis. Additionally, we will re-name the cross direction y for 2-d familiarity. This leaves:

$$x^2 + \frac{y^2}{K+1} = \frac{R^2}{(K+1)^2}.$$

Dividing by the right-hand side, we get:

$$\frac{x^2(K+1)^2}{R^2} + \frac{y^2(K+1)}{R^2} = 1.$$

Now if we define constants a and b according to:

$$a^2 \equiv \frac{R^2}{(K+1)^2},$$

$$b^2 \equiv a^2(K+1),$$

we can re-express in a very familiar form:

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1. \tag{18}$$

When $K = 0$, for a circle, $a = b = R$, and we can re-express Eq. 18 as the more familiar $x^2 + y^2 = R^2$. For ellipses, Eq. 18 is already in the most familiar Cartesian form with semi-major and semi-minor axes a and b. Note that in the case of ellipses, $K + 1$ is always positive. In familiar terms, the eccentricity, e, of an ellipse satisfies: $a^2(1 - e^2) = b^2$, from which we learn that $K = -e^2$. For hyperbolae, $K + 1 < 0$, so that a^2 and b^2 differ in sign. If we associate $\alpha^2 = a^2$ and $\beta^2 = -b^2$, we could re-write Eq. 18 in the more familiar form for hyperbolae in Cartesian coordinates. The asymptotic slope is then β/α.