

Every time I get on the air I hear variations in signal strength. It is so common it has an abbreviation in CW - QSB. I was curious if I could model some of the effects. I thought about how radio waves 'bounce' from the ion fields caused by the sun and formed by the earth's magnetic field. Since the signal variations are cyclic in nature I used a sine wave to model the ionosphere's lower surface.

How radio waves are reflected and refracted is a complex subject. I wanted to examine a simpler system with a sinusoidal ionosphere, in a two dimensional space, on a flat earth. I wrote a piece of software to simulate the effect of the ionosphere on radio waves. How non-linear is a reflection from a sine wave? The ionosphere is modeled with perfect reflectivity with the antenna sending out a pattern of rays which reflect from the ionosphere and land where they may.

I used Newton's method to solve for the intercept point between each transmitted ray and the sinusoidal ionosphere. With that point I calculated the the vector normal to the surface which let me use Snell's law to determine the reflected angle. This gave me the intercept point and the reflected angle. I solved the equation at ground level ( $y=0$ ) for the third point of my system.

Once I had the geometry worked out I stepped the phase to animate the system. You can see how much your signal bounces around even though it is a 2D model, on a flat earth. I added a user interface to control the altitude of the ionosphere, the amplitude, frequency, and phase of the sinusoid. I wanted the user to be able to change those variables as the system is running. Then I added limits, because the system explodes if pushed too hard. Newton's method works OK for rays between 25 and 60 degrees. Outside of those bounds I had to add traps for the failure modes. Limiting the range was simpler. It is surprising how much difference is caused by tiny input changes.

I wrote the code so it can be used on Windows, Linux, and Apple operating systems. With the latter two you need to install Wine, but that is a light tool which causes no problems with other applications. I learned to write windowing code by writing applications for my Macintosh. Then I moved to Windows and found the APIs easier to use. Now I write Win32 code and use Wine on my Linux box. Works great and lets me send applications to others without a problem.

## **User Interface**

The user interface consists of a few keys and the control panel. Use N, or T to toggle the normal or tangent indicators, or use Q to quit the application. You can left-click the display to close the app too.

Control the slider with your mouse, or more accurately with the arrow keys, Page Up, Page Down, Home, or End. Use the up and down arrow keys to single step a parameter. Home and End keys move you to the top and bottom of each slider quickly. Page Up and Page Down step from top to bottom in chunks. Use the tab key to step between sliders.

When the behavior becomes chaotic you have exceeded the capability of Newton's method. However, you have probably exceeded real world conditions at that point too.

## Deeper

The transmitting antenna is placed at the origin (0, 0). I generated a set of rays to test an area of space with a range of angles from 55 degrees down to 26 degrees, stepping by 2 degrees between each sample. Each ray starts as a unit length direction vector, a vector from (0, 0) outward along

$$ray(\xi) = \begin{bmatrix} \cos(\xi) \\ \sin(\xi) \end{bmatrix} \quad \text{Slope is calculated as} \quad \text{Slope} = \frac{\sin(\xi)}{\cos(\xi)}$$

Next, we determine where our ray hits the ionosphere function

$$f(x) = \text{Altitude} + \text{Amplitude} * \sin(\text{frequency} * (x + \text{phase}))$$

Newton's method incrementally finds the root of an equation. In this case I wanted to minimize the difference between ray(x) and f(x). We are looking for when the difference between one function and the other is zero instead of finding where a function crosses the x axis. It works as long as our two functions are continuously differentiable. That is why I started with a generic sine wave. It is complex enough to be useful while simple enough to be tractable.

Our search function  $h(x) = \sin(\xi) / \cos(\xi) * x - \text{Altitude} + \text{Amplitude} * \sin(\text{frequency} * (x + \text{phase}))$

with its derivative  $h'(x) = \sin(\xi) / \cos(\xi) - \text{Amplitude} * \text{frequency} * \cos(\text{frequency} * (x + \text{phase}))$

Start by feeding the root finder a guess. I chose a number somewhere in the middle of the range while setting my error a little less than 1.0. The function iteratively approaches the answer or it blows up. If you push the system too hard you will see the breaking point. Once the following algorithm finds where the difference is less than the acceptable error (close to zero) we have our intercept point.

```
for (int iter = 1; iter <= maxIter; iter++)
{
    diff = h(x0) / h'(x0);
    x1 = x0 - diff;                                // new guess
    if (fabs(diff) < allowedError)
    {
        return x1;
    }
    x0 = x1;
}

struct pt calc(float guess)
{
    float intercept;

    intercept = newton(guess);                      // calculate intercept point
    if (!converge) { a.x = -999; a.y = -999; return a; } // guard value
    a.x = intercept;                                // beginning of reflected ray
    a.y = altitude + A * sin(omega * (intercept + phase));

    return a;
}
```

Now, the next bit of calculus, find the derivative of the sine wave at the intercept point.

For this case we have  $f(x) = \text{Altitude} + \text{Amplitude} * \sin(\text{frequency} * (x + \text{phase}))$

and  $f'(x) = \text{Amplitude} * \text{frequency} * \cos(\text{frequency} * (x + \text{phase}))$

gives us our tangent.

I parameterized the derivative as  $P' = \begin{bmatrix} 1 \\ \text{Amplitude} * \text{frequency} * \cos(\text{frequency} * (a.x + \text{phase})) \end{bmatrix}$

We can obtain the normal to the curve by rotating the tangent vector counter-clockwise. You could apply a 2D rotation matrix, or do it manually by swapping coefficients and negating the x member of the vector, which is equivalent.

$$N = \begin{bmatrix} -\text{Amplitude} * \text{frequency} * \cos(\text{frequency} * (a.x + \text{phase})) \\ 1 \end{bmatrix}$$

Once I had the normal I could calculate its angle with the arctangent function.

$$na = \arctan(1, -\text{Amplitude} * \text{frequency} * \cos(\text{frequency} * (a.x + \text{phase})))$$

Snell's law tells me the angle of incidence equals the angle of reflection. In this case finding those angles is a little different since I used the normal to the curve. The normal vector supplies my reflection reference. Here is the algorithm I used to determine the reflected angle from the ray angle and the normal angle:

$$\delta = na - ra \quad \text{where } ra \text{ is the ray angle we started with.}$$

The reflected ray angle is calculated with  $rra = na + 180^\circ + \delta$

With the intercept point and the reflected angle I can build a line. I needed to find the other end point. Simply find ground level, conveniently located at  $y=0$ .

Using the point slope form of a line  $(y - y_1) = m * (x - x_1)$

Where the slope is determined by the reflected angle,  $m = \sin(rra) / \cos(rra)$

We know our intercept point  $\begin{bmatrix} a.x \\ a.y \end{bmatrix}$  so  $(y - a.y) = m * (x - a.x)$

At  $y=0$  we get  $-a.y = m * (x - a.x)$

Solve for x to get  $x = a.x - a.y / m$

Draw our reflected ray from  $\begin{bmatrix} a.x \\ a.y \end{bmatrix}$  to  $\begin{bmatrix} a.x - a.y / m \\ 0 \end{bmatrix}$

## What is displayed

Start with our antenna located at the origin (0, 0). Radiation is represented by sample rays ranging from 55 degrees to 26 degrees above horizontal with a 2 degree spacing. I changed the color of every 5<sup>th</sup> ray to help keep track of the action. The bottom of the ionosphere is represented by a blue sinewave whose parameters are set by the user.

You can control the altitude of the ionosphere layer, plus the sine wave's amplitude, frequency, and phase. The latter is used as a means to animate the system. I wanted to 'see' what happens to the signal as I change the settings. Can I replicate the slow speed changes in signal strength or the fast flutter which chops dahs into dits? When I saw a caustic appear in the system I knew I had solved the problem correctly. While sine waves can't focus light waves into a beam, they can concentrate them somewhat. Notice the light and dark patterns cast on the bottom of a sunlit pool.

I mentioned using N or T to toggle the normal or tangent vectors, but did not mention that the normal vector is on at start up. The hedgehog indicators display how the normal vector varies at even the smallest amplitudes and slowest frequencies. If I toggle them off I lose their sensitivity. Pressing T, while the display window has focus, shuts off the normal vectors and turns on their associated tangent vectors. Unless the amplitude is high this effect is rather boring. But I needed them along the way for debugging purposes so I left them in as legacy bits.

If you will notice the origin is offset from the left side of the display, as well as a little above its lower boundary. This, too, is from the debugging stages. I needed to know where ground level was located. You can resize the window but that is not attached to the underlying code. You just get a larger area of blank. I could fix the borders in place or modify the code in a zillion places to allow for scaling. My usual method for modifying settings is to use the compiler :)

## Why this way?

From observation we notice QSB effects signals rhythmically. Cyclical effects can be modeled as sinusoids, or a mixture of them. Newton's method requires a continuously differentiable function so a sine wave was my first choice. I could mix in more sine waves for a complex wave form but they may exceed the ability of Newton's method.

I imagine the ionosphere as a bubble controlled by electrical and magnetic fields. The earth's field keeps the ionosphere 'inflated' so it is mostly spherical with height variations due to sun angle. When the ionosphere is not quiescent, ripples are induced by solar wind. The sun forces the ionosphere to move, while the earth's field tries to keep the ionosphere mostly spherical. The interactions between these two effects cause rhythmic height disturbances. Any time I hear cyclical in a description I think of applying cyclic tools like sinusoids.

I devised a system of vector and matrix tools which work well for 2, 3, or higher dimensional systems. I could use the same tools to work on a 3D system of sinusoidal sheets. I don't think I am making my hammer fit any problem, but rather finding the first order approximation of a problem.

Another assumption I fixed in place was perfect reflectivity. That is just a first order guess. I expect the reflection is angle dependent. If the ray hits the ionosphere at a steep angle it may just pass right through. We know the ionosphere is frequency dependent which gives us the MUF. I expect the "interface" is much more complex than a simple mirror surface. But I do expect it to behave refractively. What we call the ionosphere is a layer where the density of the electric field increases. When light hits a difference between one refractive index and another the light is 'bent'. Radio waves are just light waves at another frequency. They should react the same way. Above a critical angle the ray will pierce the ionosphere until it hits a denser field where it will refract. So the theory must include rays which are partially reflected along with those which are fully reflected. If I added another layer on top of the first I could create a ducting effect. Now, how do we know that the conditions at one location are the same as another? The ionosphere is effected by the earth's EM field while it lives in the sun's much larger EM field. There are many factors to consider to create a realistic model. My attempt is only a level zero simulation. Just enough to get started, but enough to show how non-linear the system is with perfect reflections, using a sine wave shaped surface.

## **Future Work**

The rays you see in the simulation are just part of an antenna radiation pattern. Later, I could use different colors to indicate the strongest lobes of the pattern. For now I am using the density of the sample rays to indicate how strong the signal is at any given location. If there are more lines aimed there the signal is stronger. If they are spread out the signal is weaker. I changed the color of every fifth ray to differentiate them. Otherwise you lose the pattern in a sea of orange rays.

Another option is to use a color palette based on power levels. Start at the antenna with full power then attenuate the signal by path length and interaction with the ionosphere. Each ray will show the power level at each pixel of its length. Instead of drawing each line from two points, with one color pen, I will draw them pixel by pixel along the way. Takes a little more code but the effect should be very nice. Once that is working create a starting power pattern for any given antenna type. The output will still be colored rays but they will show the characteristic take off angles of each antenna.

Now examine how the radio waves interact with the ionosphere a little more thoroughly. Use a refractive frame of reference instead of the initial reflective one. I know more about the theory of light wave refraction than I do about radio waves. Time to hit the books and dig out more equations to model. EM field theory must have a term for refractive index :) When the radio beam hits the surface of the ionosphere how much of it passes through, how much of it is bent into the layer, and how much of it is reflected? Does it pierce, bend, burrow, or reflect? I expect them all so will apportion power to each path separately. How do those numbers change with angle of incidence?

## A few tools from linear algebra

```
struct pt plus(struct pt a, struct pt b)
{
    struct pt c;
    c.x = a.x + b.x;
    c.y = a.y + b.y;
    return c;
}
```

```
struct pt minus(struct pt a, struct pt b)
{
    struct pt c;
    c.x = a.x - b.x;
    c.y = a.y - b.y;
    return c;
}
```

```
struct pt scale(struct pt a, float b)
{
    struct pt c;
    c.x = a.x * b;
    c.y = a.y * b;
    return c;
}
```

```
struct pt div(struct pt a, float b)
{
    struct pt temp;
    temp.x = a.x / b;
    temp.y = a.y / b;
    return temp;
}
```

```
float dot(struct pt a, struct pt b)
{
    return a.x * b.x + a.y * b.y;
}
```

```
float dist(struct pt a)           // length of vector
{
    return sqrt(dot(a, a));
}
```

```
struct pt norm(struct pt a)       // normalize to unit vector
{
    return div(a, dist(a));
}
```

```
// g is the normal vector, normalized, scaled by 80, and added to p
s = plus(p, scale( norm( g ), 80 ) ) ;
// s is the vector of the scaled normal of the curve from the current point p
```