Hydrodynamic Modeling

New Mexico

Supercomputing Challenge

Final Report

April 2, 2008

Team 3

Albuquerque Academy

Team Members:

Chad Bustard

Nathan Thai

Teacher:

Jim Mims

Project Mentor:

Bob Robey

Table of Contents

Executive Summary

We created a deep ocean wave model using two different spectrums, the Pierson-Moskowitz spectrum and the JONSWAP spectrum, which can be used to find wave period, length, height, speed, etc at different points in the ocean. We also considered the transformation that occurs when waves transition from deep water to shallow water, and we included this by using the TMA spectrum, a variation on the JONSWAP spectrum. We found that the models differ by small amounts because of their different variables, and we found the JONSWAP model, combined with the TMA spectrum, to be the most useful model in most occasions. Each spectrum was then applied to the stokes drift equation to give the net forward velocity of a particle, such as an oil droplet, on the waves. We hope to check the results for each spectrum and for the stokes drift equation against marine data before the expo in Los Alamos.

Problem Statement

To model particles on the ocean surface, two types of information are needed: 1) A model for the distribution of ocean surface waves describing wave speed, height, and direction, 2) an equation to govern the motion of particles on these surface waves. By using the Stoke's drift equation, applied to waves in deep water, it is possible to find the time-average drift velocity of a particle on a wave. Considering wind as the major factor in wave development, it is possible to use one of many standard models describing surface wave energy to obtain the desired characteristics of ocean waves. Shoaling effects can then be considered as a wave passes from deep water to shallow water, and the shallow water equations can predict the final stages of a wave as it breaks on the shore. Various models can be used for the deepwater stage of the waves, and it is simple to switch between these models at any time.If the particles are considered to be droplets of oil, each model can be checked against previous data taken from oil spills across the world to check the accuracy of the equations. An analysis of the deviations of each model from the actual data can tell which equations worked the best on average, and which equations worked the best according to the tested situation. For example, one model may give more accurate results for light wind speed than other models. It can also be found why each model deviated from experimental results based off of the simplifications we employed especially in the nature of the wind and oil.

Research

Linear Wave Theory

Terms:

Crest – highest point of a wave

Trough – lowest point of a wave

Wavelength (m) – distance between successive crests, written as L

Equilibrium Position - position halfway between the trough and the crest

Amplitude – distance from the equilibrium position to the crest

Period (s) – amount of time taken for two successive crests to pass a fixed point, written as T

Frequency (Hz)–number of wave cycles to pass a fixed point in a certain amount of time, written as f

Important Assumptions:

- Wave amplitude is zero, making the sea surface approximately a level plane
- Wave flow is 2D
- Coriolis force and viscosity (friction caused by the intermolecular forces between water molecules) are negligible

Equations:

k = wave number

$$k = 2\pi/L$$

w = wave frequency in radians/second (rad/s)

$$w = 2\pi f = 2\pi/T$$

Also,

$$w^2 = gk \tanh(kd)$$

where d is the ocean depth (m) and g is the acceleration due to gravity where g = 9.81 m/s. In deep water, it is assumed that d > L/4. Therefore, d >> L, so kd >> L and tanh(kd) = 1. This gives:

$$w^2 = gk.$$

c = wave speed, which is the speed at which one phase of a wave propagates (m/s).

$$c = L/T = w/k.$$

From the deep water assumption that d >> L

$$c = (g/k)^{1/2} = g/w$$

In reality, not all waves propagate at this speed. Instead, wave speed varies within a group of waves of similar speeds. The average speed of waves in this group can be approximated by the equation

$$c_g = c/2$$

where c_g is called the group speed.

$H_{1/3}$ = significant wave height, which is the average height of the tallest 1/3 of waves observed. In a narrow range of wave frequencies, this value is related to the standard deviation of sea surface displacement, $<\zeta^2>^{1/2}$.

$$H_{1/3} = 4 <\zeta^2>^{1/2}$$

$<\zeta^2>$ can be calculated by integrating the spectral energy density with respect to frequency in rad/s over an infinite interval.

$$<\zeta^2> = \int S(w)dw \text{ from } 0 \text{ to infinity}$$

S(w) is unique to each model used to calculate the wave spectra, which will be covered in the next pages.

Wave Spectra

Instead of a one to one relationship between wind speed and wave speed, wind develops a spectrum of waves with different energies, where the probability of observing a wave of a given energy depends on the wind speed. For example, a low energy wave would more commonly be found at low wind speeds than at high wind speeds. Over the years, multiple equations have been formulated to describe the energy distribution of waves. Each of these equations considers wind speed to be the main factor for developing waves. Stronger wind speeds cause the wind to blow over a larger area, causing the waves to get bigger and bigger.

One of the simplest and earliest models is that of the scientists Pierson and Moskowitz. The Pierson-Moskowitz model assumes that, over a long enough period of time, the waves in the sea will stop developing and the sea will come to a steady state. Using accelerometers on British ships in the North Atlantic in 1964, they measured wave speeds and wind speeds at times when the wind had been blowing for a long period of time over a large area of the ocean (when they believed the ocean to be fully developed).

Assuming the sea is at its fully developed state, however, can cause some problems. What about when waves are in the process of developing? In 1968 and 1969, scientists in the Joint North Sea Wave Observation Project(JONSWAP) found that the sea continues to develop from wave-wave interactions even when the wind has dissipated. They proposed an equation, again based off of observations,

including another variable: fetch, which is the distance over which a steady wind has been blowing. This model is widely used by people in the sea industry.

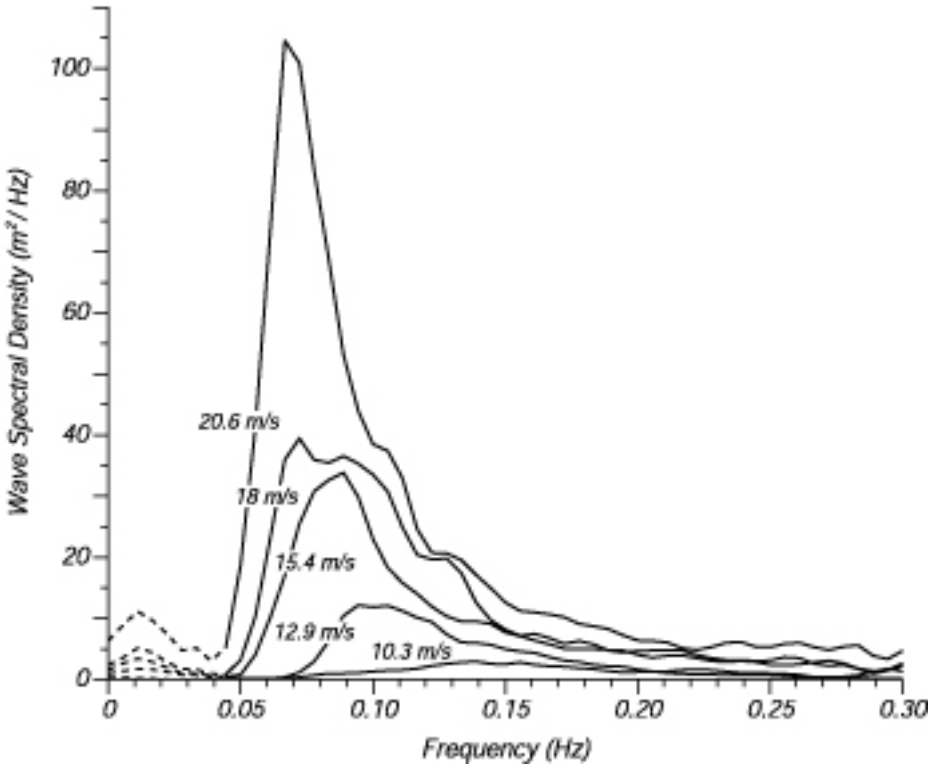
Both of these models, however, only apply to deep water situations. When a wave transitions into shallow water, the spectra changes. The TMA model in one often-used model that accounts for this.In our project, we experimented with all of these models.The equations for each model will now be presented.

Pierson-Moskowitz

S(w) is the wave spectral density as a function of frequency.

$$S(\omega) = \frac{\alpha g^2}{\omega^5} \exp\left[-\beta \left(\frac{\omega_0}{\omega}\right)^4\right]$$

where $\alpha = 8.1 \times 10^{-3}$, $\beta = 0.74$, $w_0 = g/U_{19.5}$ where $U_{19.5}$ is the wind speed at a height of 19.5 m above the sea surface, the height at which wind speed readings were taken on the ship used by Pierson and Moskowitz. Therefore, from the relationship between w_0 and the wind speed and the relationship between the frequency and the phase speed (c = g/w), we have an equation relating wave energy to wind speed. From the graph of spectral density vs. frequency, we can see that there is a peak frequency corresponding to a point on the graph which represents thehighest wave spectral density for a given wind speed. This peak frequency is the most common frequency of the waves developed by that particular wind speed.



For example, for a wind speed of 20.6 m/s, the peak frequency, f, is approximately 0.07 Hz and the highest spectral density is about 105 m^2/Hz. It is easy to see that differentiating s with respect to w and setting this expression equal to zero will yield the peak frequency, w, which is equal to $2\pi f$.

$$ds/dw = 0$$

$$w = 0.877g/U_{19.5} = w_p$$

where w_p is called the peak frequency.

Using the equation c = g/w, we find that

$$c_p = 1.14U_{19.5}$$

where c_p is the peak phase speed. We can also find the significant wave height:

$$\langle \zeta^2 \rangle = \int_0^\infty S(\omega)\, d\omega = 2.74 \times 10^{-3} \frac{(U_{19.5})^4}{g^2}$$

$H_{1/3} = 4 \langle \zeta^2 \rangle^{1/2} = .21(U_{19.5})^2 / g$

Where $\langle \zeta^2 \rangle^{1/2}$ is the standard deviation of sea surface displacement related to the wave energy by the equation

$E = p_w g \langle \zeta^2 \rangle$

where p_w is water density and E is energy

The significant wave height may not seem important right now, but it will be important in the Stokes drift equation.

JONSWAP

$$S(\omega) = \frac{\alpha g^2}{\omega^5} \exp\left[-\frac{5}{4}\left(\frac{\omega_p}{\omega}\right)^4\right] \gamma^r$$

where

$$r = \exp\left[-\frac{(\omega - \omega_p)^2}{2\,\sigma^2\,\omega_p^2}\right]$$

In this equation,

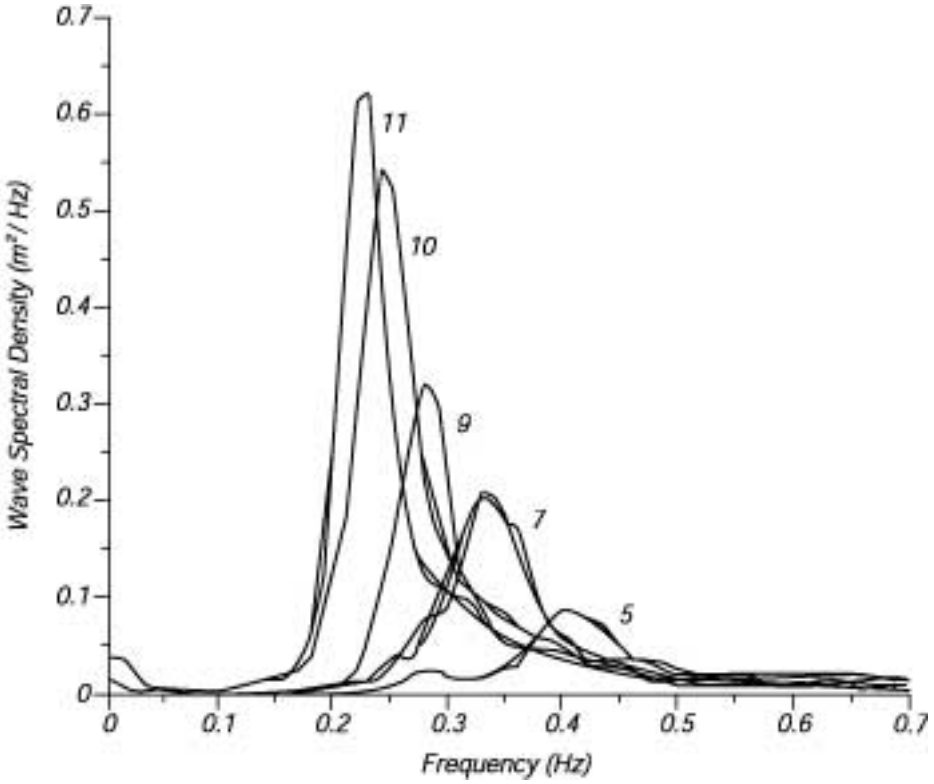
$$\alpha = 0.076 \left(\frac{U_{10}^2}{F\,g}\right)^{0.22}$$

Note that the wind speeds were taken at 10 m above the sea surface instead of 19.5 m. If the atmospheric boundary layer is assumed to have neutral stability for moist air flow over the water,

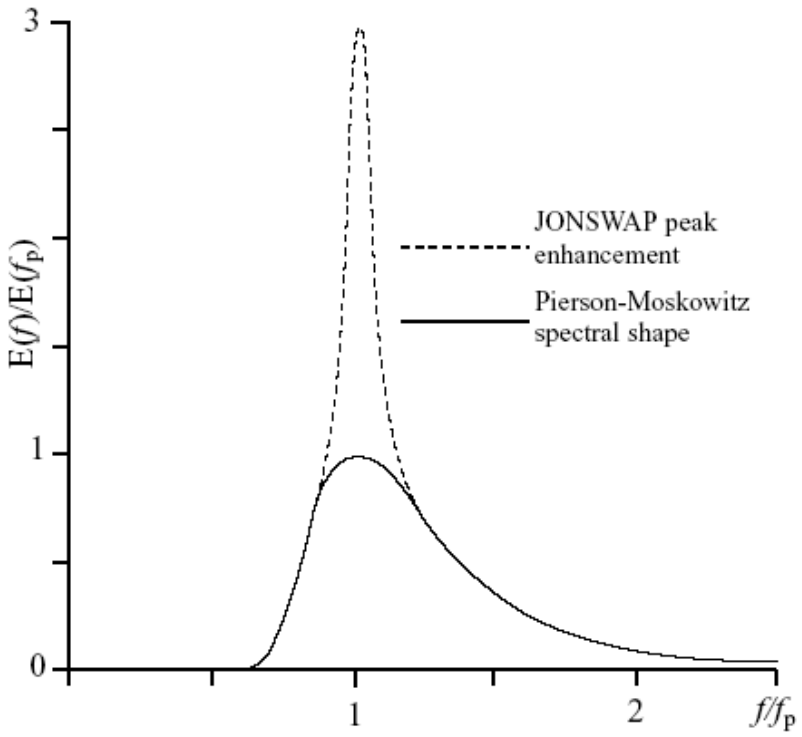
$$U_{19.5} = 1.026U_{10}$$

When the wind speed is inputted, it will be assumed that it is the wind speed at 10 m above the sea surface. For the Pierson-Moskowitz model, the input will be multiplied by 1.026 to give $U_{19.5}$.

Also, note the variable, F, which is fetch. This is a major difference between the Pierson-Moskowitz and JONSWAP equations. Unlike the Pierson-Moskowitz model, which needs only one inputted variable (wind speed), JONSWAP also requires an input of fetch. A graph of spectral density vs. frequency for given fetch values is given below according to the JONSWAP equation. It is obvious that the peak spectral density increases and the peak frequency decreases with increasing fetch. This means that a greater fetch causes a more narrow distribution of wave energies, and the peak frequency is more common among waves than if the fetch was smaller.



The JONSWAP spectral graphs are more narrow and more peaked than the Pierson-Moskowitz graphs.



γ describes the amount of peakedness of the spectrum. It is the ratio of the JONSWAP peak spectral density to the Pierson-Moskowitz peak spectral density. It actually ranges from about 1 to 6 with an average value of 3.3. Assuming that $\gamma = 3.3$ just simplifies the equation. γ^r is just a peak enhancement factor compared to the Pierson-Moskowitz equation.

$$\gamma = 3.3$$

$$w_p = 2.84\, g^{0.7}\, F^{-0.3}\, U_{10}^{-0.4}$$

$$\sigma = \begin{cases} 0.07 & \omega \leq \omega_p \\ 0.09 & \omega > \omega_p \end{cases}$$

Integrating S(w) yields

$$<\zeta^2> = 1.67 \times 10^{-7} (U_{10})^2 F / g$$

meaning that

$$H_{1/3} = .00163\ U_{10} (F/g)^{1/2}$$

TMA

$$S(w) = S(w)_{JONSWAP} \times f(w,d)$$

where $S(w)_{JONSWAP}$ is just a modified version of the JONSWAP spectral density equation with different values for γ and α, and f(w,d) is a frequency and depth dependent factor that reduces the peak density of the spectrum. This reduction comes about because, as waves enter shallow water, their energy dissipates as the waves get caught on the ocean floor.

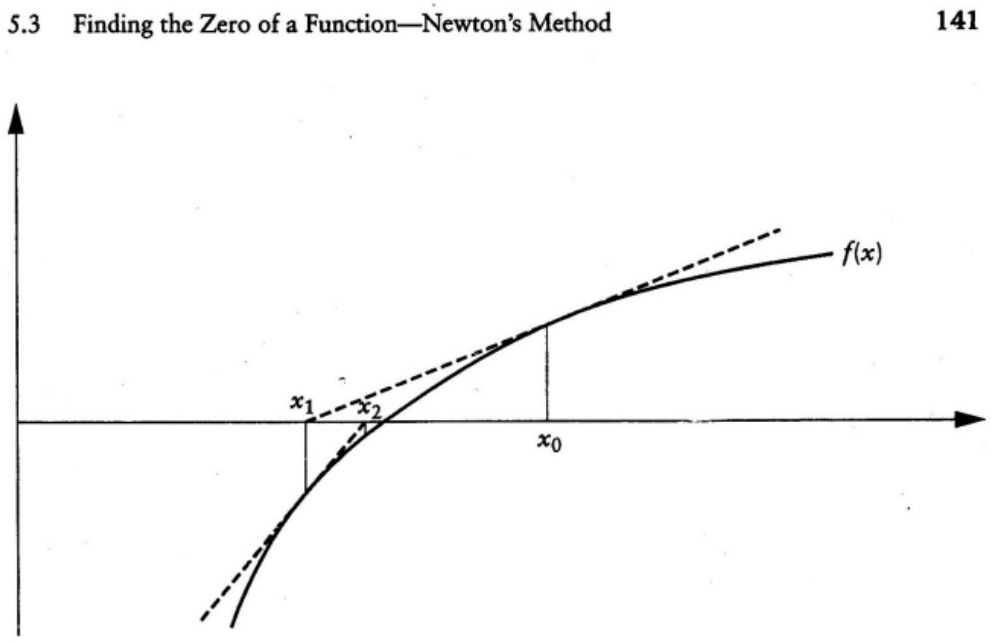
$$\alpha_{TMA} = \alpha_{JONSWAP} (L/L_0)^{0.49}$$

$$\gamma_{TMA} = \gamma_{JONSWAP} (L/L_0)^{0.39}$$

L_0 is the wavelength of the wave with peak frequency in deep water. L is the wavelength of the wave with the same peak frequency in water where d < L/4. These can be calculated by the relationship

$$\frac{L}{L_0} = \frac{c}{c_0} = \frac{\sin\alpha}{\sin\alpha_0} = \tanh\frac{2\pi d}{L}$$

Where d is the water depth. This equation can be solved for L by letting $L/L_0 - \tanh 2\pi d/L = 0$, with L_0 as a constant and L as the dependent variable, and solving for L at a certain d. This can be done by using the Maclaurin series for tanh x , expanded out to a certain number of terms, where $x = 2\pi d/L$, and then using a root finding algorithm to find L. Because this will need to be done tens of times each time our grid refreshes, we will need a fast algorithm. Therefore, we will choose Newton's method for finding the roots. This method basically takes tangent lines to the curve being evaluated, finds the root of the tangent line, takes the tangent line at the point on the curve with this x coordinate, and does this over and over again until it reaches an x coordinate that, when plugged into the curve equation, gives a value sufficiently close to zero. An illustration of the method is shown below.



Maclaurin series for tanh x:

$$\tanh x = x - \frac{1}{3}x^3 + \frac{2}{15}x^5 - \frac{17}{315}x^7 + \frac{62}{2835}x^9 + \dots$$

The code is shown in the jonswapequations class in Appendix A.

After we know L and L_0, we can find c.

$$c = Lc_0/L_0$$

Where $c_0 = gT/2\pi$

f(w,d) can be approximated within 4% accuracy as

$$f(w,d) = w_h^2/2 \qquad \text{for } wd <= 1$$

$$f(w,d) = 1 - \tfrac{1}{2}(2 - w_h)^2 \qquad \text{for } wd > 1$$

where $w_h = w(d/g)^{1/2}$ and d is the water depth

This model is used for the transition state between shallow water and deep water. As stated previously in the linear theory section, deep water approximations that d >> L are useful when d > L/4. Shallow water approximations are useful when d << L or when d < L/11. When L/11 < d < L/4, the TMA model is incorporated as part of the JONSWAP model. Even with the f(w,d) term, the w_p term remains in the equation and doesn't change from the JONSWAP value of w_p. Therefore, it can be seen that the peak frequency for the TMA spectrum is still w_p.

For calculating the significant wave height, f(w,d) presents some challenges. Up until now, we have integrated the spectral density function to find the significant wave height. Taking the integral of the TMA equation is very arduous, however, so

we use a different equation formulated by Young and Verhagen in 1996, which is now considered to be possibly the best equation for finding significant wave height as a function of depth and fetch. This equation was then re-analyzed in 2006 by Breugem and Holthuijsen, who then modified some of the exponents to give the more accurate equation below.

$$H_{1/3} = 0.24\ (\tanh(0.343d^{1.14})\tanh(4.14 \times 10^{-4}\ F^{0.79} / \tanh(0.343d^{1.14})))^{0.572}$$

αIn shallow water, kd <<1, so tanh (kd) = 1 and

$$w^2 = gk^2d$$

$$c = (gd)^{1/2}$$

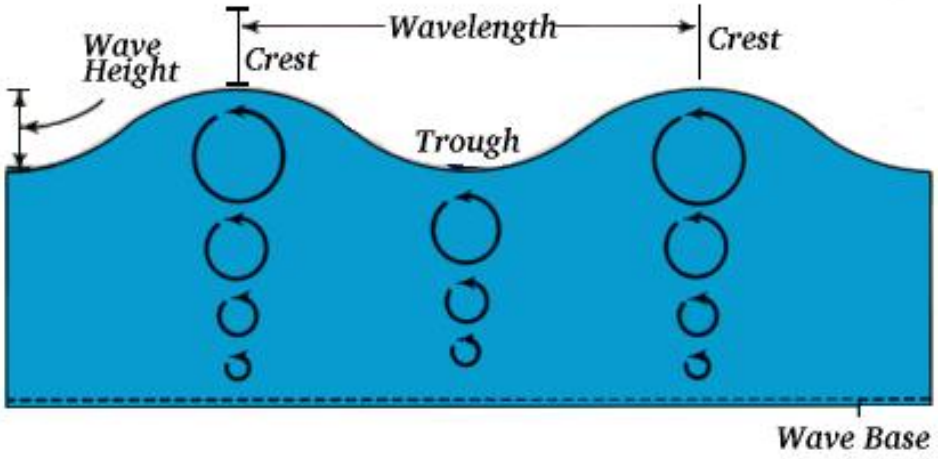
$$c_g = c$$

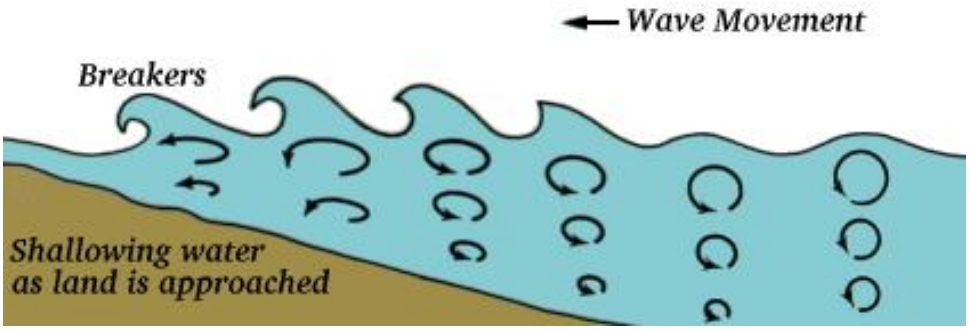
All other equations stated earlier are the same for deep water and shallow water. When d < L/11, these equations will be used in place of the equations for w^2, c, and c_g in deep water.

Each of the spectra outlined above are developed from observations but are put into equations using Fourier series. Fourier series use the concept that any function can be created from sine and cosine functions. The specifics on how the spectrum equations were created from this concept, however, are irrelevant to carrying out our project and will not be discussed.

Stokes Drift

The last idea that needs explaining is Stokes drift. By using Stokes drift, we assume that the particles on the waves are non-diffusive in the water.In deep water, particles on waves should undergo an elliptical orbit in the water. In shallow water, this elliptical shape breaks down, allowing the particles to travel forward over time. Eventually, as the waves get very close to the shore, they break.

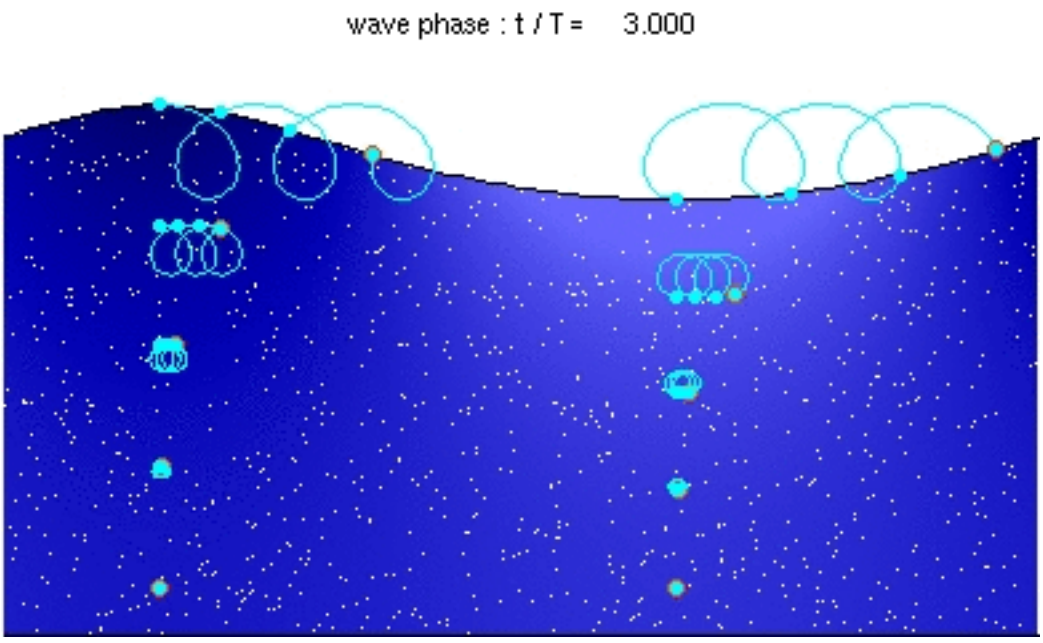




In reality, though, particles in deep water also have a net forward movement. This is not well explained, but the Stokes drift equation gives a mathematical description of the average net forward velocity.

$$\overline{u}_S \approx \omega\, k\, a^2\, \mathrm{e}^{2kz} = \frac{4\pi^2 a^2}{\lambda T}\, \mathrm{e}^{4\pi z/\lambda}.$$

where u_s is the net average particle velocity, a is the amplitude, which we will assume to be $H_{1/3}$, and λ is the wavelength, L. z is always negative, and the absolute value is the distance below the ocean surface. It can be seen that the velocity is highest at the surface where z = 0 and decreases as the distance below the surface increases. The picture below shows the actual trajectories of particles at different distances below the surface.



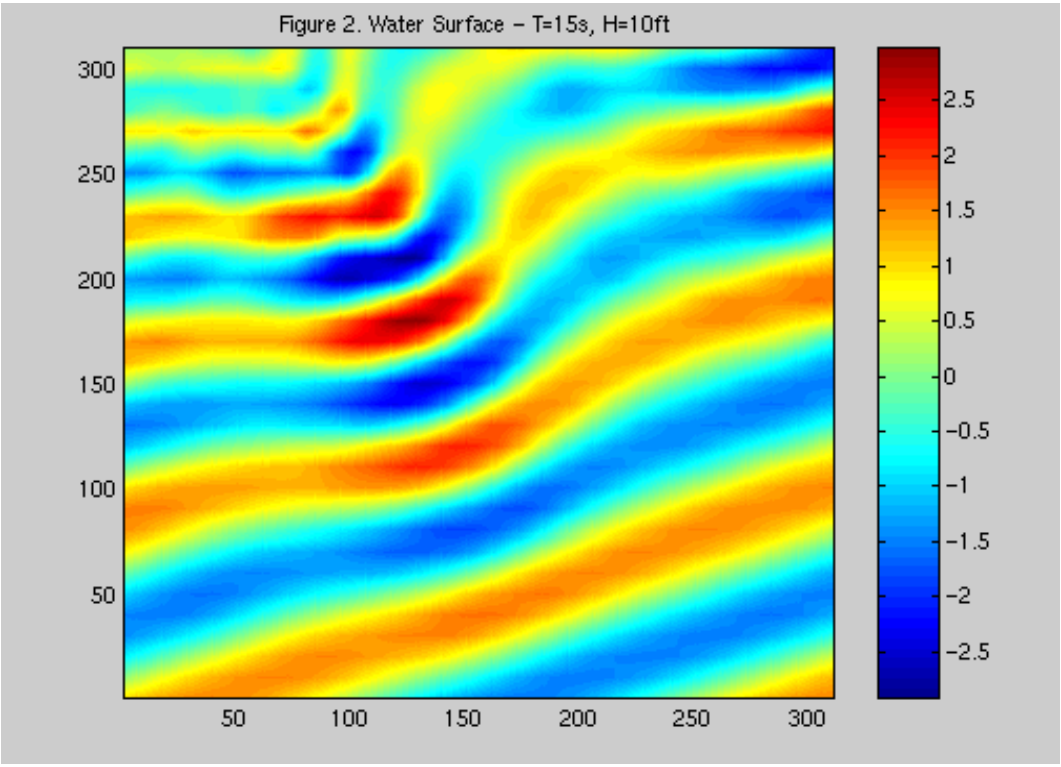
The other variables in the equation can be found from the wave spectra models and the linear theory of waves.

Analysis

The program gives a good model of ocean waves for randomly generated wind speed and angle. We considered that wind would most likely not exceed 5 m/s and that wind angle would probably range from 0 to 180 degrees. In a more realistic model, wind gusts over this 5 m/s boundary might occur, and wind direction may even reverse at times. This would only cause slight differences in the wave characteristics, though, and since it is basically impossible for us to account for this in our program, we believe that our program does a fair job. There are only a few things that could have possibly been considered in making our program that would have improved the output but would have taken too much time for us to finish by our deadline.

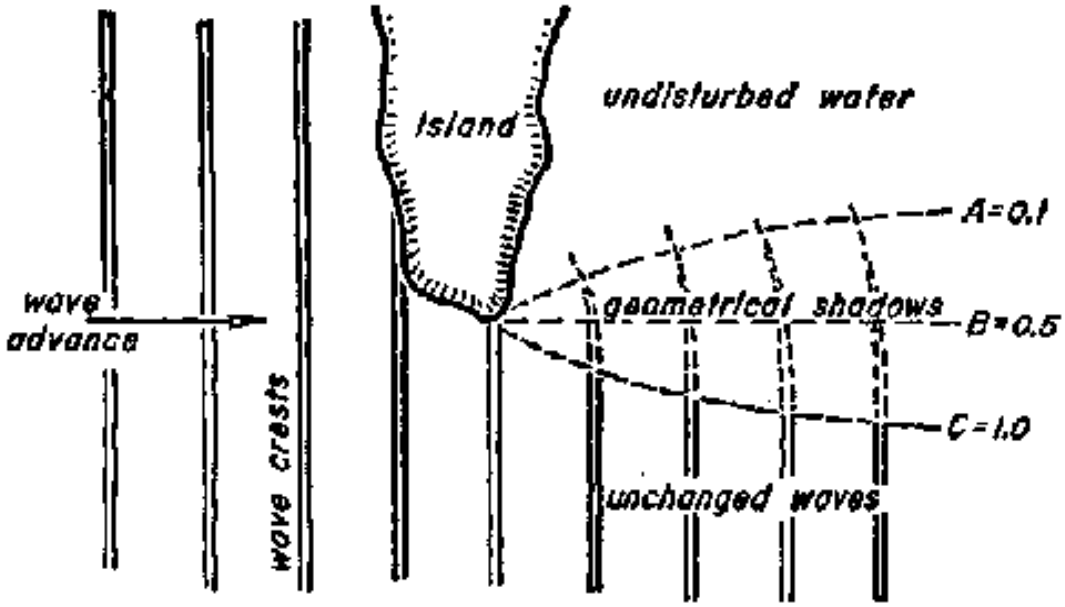
1) It has been shown that wind is the primary factor in *developing* waves, but non-linear wave-wave interactions are the greatest factors in keeping waves going and in determining wave direction. Our program makes the somewhat disturbing assumption that, at each coordinate, the wave direction is the same as the wind direction. This is not true. However, we were not able to account for this because of our limited time. The JONSWAP supposedly do a better job than the Pierson-Moskowitz equations at including these wave-wave interactions; however, since the model is only one-dimensional, we cannot account for the change in direction caused by these interactions. On a large scale, this probably wouldn't change the outcome very much, but it is worth considering.

2) Our program assumes that each wave, when approaching a shore, approaches perpendicular to the shore. This is usually not the case, especially in small gulfs or bays. When waves approach at an angle, refraction occurs when the front of the wave slows down and the back of the wave must kick out to the side, making a bow-shaped wave in shallow water and in the transition state between shallow and deep water. This naturally causes the direction to vary depending on the location in the wave group. We didn't account for this, however, since our program mainly considers deep water conditions, the effects of refraction are almost negligible but still worth noting.



3) Diffraction also occurs when waves approach a shore. When waves hit a peninsula or island for example, the water on the other side of the island in

the so-called “shadow zone” will behave erratically. We didn’t have time to research this and, therefore, didn’t include it. Once again, since our program mainly considers deep water conditions, these effects are almost negligible, but still worth noting.



4) Lastly, effective fetch plays a role when a wave encounters an oddly shaped shore. We consider there to be just one fetch for each coordinate, but there are actually more. Effective fetch accounts for these. This could play a role when a wave encounters a shore, but, as said before, the effects are almost negligible.

Overall, in choosing between the Pierson-Moskowitz model and the JONSWAP model, for this case where wind is randomly generated, it is safe to say that the ocean is not fully developed at most points and that the JONSWAP model would be the better choice. The fetch variable in the JONSWAP model is the main

reason for this difference, despite the fact that it could have been better implemented, especially in shallow water conditions. For a constant wind over a large area, the Pierson-Moskowitz model will also give good results.

For our oil model, it appears to give an accurate answer. In the example shown in Appendix M, you can see that the oil stays inside the grid and attempts to move in the direction of the wind if the boundaries allow it to.

Conclusions

In conclusion, our two models take random generations of wind speed and angle (also fetch and water depth in the case of the JONSWAP/TMA model) and output different characteristics of the waves such as period, length, speed, etc. With the incorporation of the TMA spectrum, we gain a much more accurate model than the Pierson-Moskowitz model for varying sea depth in a non-fully developed sea. Small factors such as effective fetch, refraction, etc could decrease the accuracy in shallow water cases, but for deep water, these models could prove very useful for people in the shipping or deep sea-fishing industries. For many purposes, the two models could be used interchangeably, but the slight differences caused by the ocean depth and fetch terms could make the JONSWAP model more useful.

For our Stokes drift model of oil on waves, it seems like our model is fairly accurate. The code works the way we want it to; however, we would like to check the accuracy of our program against marine data and past oil spill data before the expo in Los Alamos.

References

Internet Sources:

"Higher Order Stokes Theory." 10 Feb. 2008
<http://www.ocean.washington.edu/people/faculty/parsons/OCEAN549B/stokes-lect.pdf>.Stewart, Robert H. "Chapter 16-Ocean Waves." Dept. of Oceanography, Texas a & M University. 10 Feb. 2008

http://www.coastal.udel.edu/ngs/waves.html

<http://oceanworld.tamu.edu/resources/ocng_textbook/chapter16/chapter16_04.htm >.

http://www.oceanweather.com/data/

"Stokes Drift." Wikipedia. 10 Feb. 2008
<http://en.wikipedia.org/wiki/Stokes_drift>.

"The Java Game Development Tutorial." Java Boutique. 10 Feb. 2008
<http://javaboutique.internet.com/tutorials/Java_Game_Programming/>.

Book Sources:

Besset, Didier H. Object Oriented Implementation of Numerical Methods: an Introduction with Java and Smalltalk. San Francisco, California: Morgan Kaufmann, 2001.

Holthuijsen, Leo H. Waves in Oceanic and Coastal Waters. Cambridge UP, 2007. 29 Mar. 2008
<http://books.google.com/books?id=7tFUL2blHdoC&dq=waves+in+oceanic+and+coasta l+waters&source=gbs_summary_s&cad=0>.

Lambert, and Osborne. Fundamentals of Java. Thomson Course Technology, 2006

Massel, Stanislaw R. Ocean Surface Waves: Their Physics and Prediction. World Scientific, 1996. 27 Feb. 2008 <http://books.google.com/books?id=8sHp9ml7G6YC&dq=wave+refraction+ocean&source=gbs_summary_s&cad=0>.

Reeve, Dominic, and Andrew Chadwick. Coastal Engineering: Processes, Theory, and Design Practice. Taylor and Francis, 2004. 28 Mar. 2008 <http://books.google.com/books?id=3OaD5V5wVYsC&dq=coastal+engineering+processes&source=gbs_summary_s&cad=0>.

Acknowledgements

We would like to thank our instructor, Mr. Mims, for encouraging us to enter in the New Mexico Supercomputing Challenge and for continually giving us feedback on our presentations and our code. We would also like to acknowledge our mentor, Bob Robey, for his useful comments on our proposal and interim report. Most of all, we would like to thank everyone who makes the supercomputing challenge possible. This was our first year competing in the challenge, and we found it to be a very enjoyable, rewarding competition that has greatly encouraged us to learn more computer science.

Appendices

Appendix A: jonswapequations

```
import java.util.Random;

public class jonswapequations

{

        private double waveinfo[][][];



        private int i;

        private int j;



        private double angle;



        private double f; // fetch

        private double u; // wind speed

        private double g; // gravity

        private double w; // angular frequency = 2 * PI * wave frequency

        private double T; // Period of the wave
```

```
        private double L; // Length of the wave

        private double c; // phase speed

        private double c2; // Group speed


        private double cshallow; // group and phase speed for shallow water, d <
L/11

        private double d; // water depth

        private double a; // wave height, calculated from the wind eqs

        // to be the significant wave height,

        // which is the average height of the largest 1/3 of waves in that spectrum

        private double aTMA; // wave height in TMA transition state between deep
water and shallow water

        private double cTMA; // phase speed in TMA transition state between deep
water and shallow water

        private double v; // Stokes drift velocity


        private double vshallow; // particle velocity in shallow water

        private double t;
```

```
private double totalt; // total time elapsed


private int maxx; // maximum number of cells in the x direction

private int maxy;// maximum number of cells in the y direction



public jonswapequations()

{

        t = 10800; // 10800s = 3 hours

        g = 9.81;

        i = 0;

        j = 0;


        maxy = 10000;

        maxx = 10000;


        totalt = 0;
```

```
}

/* randomly generates wind speed, wind angle, depth, and fetch for each
point (i,j)
*  then calculates period, length, height, etc. at these points depending on the
relationship
*  between d and L. The wave info is then placed in the waveinfo array.
**/
public void lengthandwavespeed()
{
        Random rand = new Random();

        thing = new double[maxy][maxx][10];
        for (i = 0; i < maxy; i++)
        {
                for(j = 0; j < maxx; j++)
                {
```

```
                    u = rand.nextInt(5);

                    angle = rand.nextInt(180);

                    d = rand.nextInt(200000);

                    f = rand.nextInt(1000);



                    waveinfo[i][j][0] = u;

                    waveinfo[i][j][1] = angle;

                    waveinfo[i][j][2] = d;

                    waveinfo[i][j][3] = f;

            }

}



for (i = 0; i < maxy; i++)

{

            for(j = 0; j < maxx; j++)

            {
```

```
                w = 2.84 * Math.pow(g, 0.7) *
Math.pow(waveinfo[i][j][3], -0.3) * Math.pow(waveinfo[i][j][0], -0.4);


                T = 2 * Math.PI / w;
                waveinfo[i][j][8] = T;


                if (waveinfo[i][j][2] > L/4)
                {
                    c = g/w;
                    c2 = c/2;
                    L = g * Math.pow(waveinfo[i][j][8], 2)/(2 *
Math.PI);
                    a = .00163 * waveinfo[i][j][0] *
Math.pow(waveinfo[i][j][3]/g, .5);
                    v = (4* Math.pow((Math.PI), 2) * Math.pow(a,
2))/(L * waveinfo[i][j][8]);
                    waveinfo[i][j][5] = v;
                    waveinfo[i][j][6] = a;
```

```
            waveinfo[i][j][9] = c2;



            waveinfo[i][j][7] = L;



}

else if(d<L/11)

{

            // amplitude doesn't apply

            // wavelength doesn't apply



            cshallow = Math.pow(d * g,.5);

            waveinfo[i][j][9] = cshallow;

            vshallow = cshallow;

            v = vshallow;



            waveinfo[i][j][5] = v;
```

```
}

else

{

        double[] coefficients = {-3.454,0,0,0,2.133333 *
Math.pow(Math.PI * waveinfo[i][j][2],4), 0, -4 * Math.pow(Math.PI *
waveinfo[i][j][2], 2) + (.159/(waveinfo[i][j][2] * L)), 0,1};

        NewtonZeroFinder zeroFinder2 = new
NewtonZeroFinder(new PolynomialFunction (coefficients),-1);


        zeroFinder2.evaluate();


        double result2  = zeroFinder2.getResult();


        cTMA = c * (result2/L);

        waveinfo[i][j][9]=cTMA;


        aTMA = 0.24 *
Math.pow(Math.tanh(0.343*Math.pow(waveinfo[i][j][2], 1.14)) *
```

```
Math.tanh(.000414 * Math.pow(waveinfo[i][j][3],0.79) / Math.tanh(0.343 *
Math.pow(waveinfo[i][j][2],1.14))),0.572);


                                    v = (4* Math.pow((Math.PI), 2) *
Math.pow(aTMA, 2))/(result2 * waveinfo[i][j][8]); // we'll just assume that the
same
                                    // stokes drift equation, which assumes depth to
be infinite, applies to this transition state, also.


                                    thing[i][j][5] = v;


                                    thing[i][j][6] = aTMA;


                                    L = result2;
                                    thing[i][j][7] = L;
                            }


                    }
```

```
            }

            }


        // These methods return different characteristics of the waves at a point
(x,y)

        public double getWindSpeed(int x, int y)

        {

            u = waveinfo[x][y][0];

            return u;

        }



        public double getFetch(int x, int y)

        {

            return waveinfo[x][y][3];

        }

        public double getDepth(int x, int y)

        {
```

```
        d = waveinfo[x][y][2];

        return d;

}

public double getAngle(int x, int y)

{



        angle = waveinfo[x][y][1];

        return angle;

}

public double getPeriod(int x, int y)

{

        T = waveinfo[x][y][8];

        return T;

}



public double getWaveLength(int x, int y)
```

```
        {
                return waveinfo[x][y][7];
        }
        public double getWaveSpeed(int x, int y)
        {
                return waveinfo[x][y][9];


        }
        public double getAmplitude(int x, int y)
        {
                return waveinfo[x][y][6];
        }
}
```

Appendix B: piersonequations

```
import java.util.Random;

public class piersonequations

{

        private double[][][] waveinfo;

        private int angle;

        private int maxx;

        private int maxy;

        private int u10; // wind speed at 10m

        private double u195; // wind speed at 19.5m

        private double g; // gravity

        private double w; // angular frequency = 2 * PI * wave frequency

        private double T; // Period of the wave

        private double L; // Length of the wave

        private double c; // phase speed

        private double c2; // group speed

        private double a; // wave height, calculated from the wind eqs
```

```
// to be the significant wave height,

// which is the average height of the largest 1/3 of waves in that spectrum

private double v; // stokes drift velocity



public piersonequations()

{

        g = 9.81;



        maxy = 10000;

        maxx = 10000;



        waveinfo = new double[maxy][maxx][10];

}



/* Randomly generates values for the wind speed and wind angle,

*  calculates wave period, length, frequency, amplitude, etc. at each point (i,j)
```

```
*  and stores the values in a 3d array

**/

public void equations()

{

        for (int i = 0; i < maxy; i++)

        {

                for(int j = 0; j < maxx; j++)

                {



                        Random rand = new Random();

                        u10 = rand.nextInt(5);

                        angle = rand.nextInt(180);

                        u195 = u10 * 1.026;



                        thing[i][j][0] = u10;

                        thing[i][j][1] = angle;

                        w = .877 * g / u195;
```

```
                    T = 2 * Math.PI / w;

                    L = g * Math.pow(T, 2)/(2 * Math.PI);

                    c = 1.14 * u195;

                    c2 = c/2;

                    a = .21 * Math.pow(u195,2) / g;

                    v = (4* Math.pow((Math.PI), 2) * Math.pow(a, 2))/(L *
T);



                    thing[i][j][2]= T;

                    thing[i][j][3]= L;

                    thing[i][j][4]= a;

                    thing[i][j][5]= v;

                    thing[i][j][6]= c2;

                }

            }

        }
```

```
// methods that return the wave info at a point (x,y)

public double getPeriod(int x, int y)

{

	return waveinfo[x][y][2];

}



public double getLength(int x, int y)

{

	return waveinfo[x][y][3];

}



public double getWindSpeed(int x, int y)

{

	return waveinfo[x][y][0];

}



public double getWaveSpeed(int x, int y)
```

```
        {
                return waveinfo[x][y][6];
        }

        public double getAmplitude(int x, int y)
        {
                return waveinfo[x][y][4];
        }
        public double getAngle(int x, int y)
        {
                return waveinfo[x][y][1];
        }
}
```

Appendix C: GUI

```
/*

 * NewJFrame.java

 *

 * Created on March 30, 2008, 6:29 PM

 */



/**

 *

 * @author  Nathan

 */

public class GUI extends javax.swing.JFrame {



    int y;
```

```
int x;



/** Creates new form NewJFrame */

public GUI() {

    initComponents();

}

/** This method is called from within the constructor to

 * initialize the form.

 * WARNING: Do NOT modify this code. The content of this method is

 * always regenerated by the Form Editor.

 */

// <editor-fold defaultstate="collapsed" desc=" Generated Code ">

private void initComponents() {

    jSeparator1 = new javax.swing.JSeparator();
```

```
WindSpeed = new javax.swing.JLabel();

WindSpeedBox = new javax.swing.JTextField();

WaterDepth = new javax.swing.JLabel();

WaterDepthBox = new javax.swing.JTextField();

WindAngle = new javax.swing.JLabel();

WindAngleBox = new javax.swing.JTextField();

Fetch = new javax.swing.JLabel();

FetchBox = new javax.swing.JTextField();

X = new javax.swing.JLabel();

XCoord = new javax.swing.JComboBox();

Y = new javax.swing.JLabel();

YCoord = new javax.swing.JComboBox();

GetJonSwapData = new javax.swing.JButton();
```

```
GetPiersonData = new javax.swing.JButton();

WaveSpeed = new javax.swing.JLabel();

WaveSpeedBox = new javax.swing.JTextField();

WaveHeight = new javax.swing.JLabel();

WaveHeightBox = new javax.swing.JTextField();

WavePeriod = new javax.swing.JLabel();

WavePeriodBox = new javax.swing.JTextField();

WaveLength = new javax.swing.JLabel();

WaveLengthBox = new javax.swing.JTextField();



setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);

setResizable(false);



WindSpeed.setHorizontalAlignment(javax.swing.SwingConstants.CENTER);
```

```
WindSpeed.setText("Wind Speed");

WindSpeed.setHorizontalTextPosition(javax.swing.SwingConstants.CENTER);



WaterDepth.setHorizontalAlignment(javax.swing.SwingConstants.CENTER);

WaterDepth.setText("Water Depth");



WindAngle.setText("Wind Angle");



Fetch.setHorizontalAlignment(javax.swing.SwingConstants.CENTER);

Fetch.setText("Fetch");



X.setFont(new java.awt.Font("Tahoma", 0, 12));

X.setHorizontalAlignment(javax.swing.SwingConstants.CENTER);
```

```
        X.setText("X");

        X.setHorizontalTextPosition(javax.swing.SwingConstants.CENTER);




        XCoord.setModel(new javax.swing.DefaultComboBoxModel(new String[] { "0",
"1", "2", "3", "4", "5", "6", "7", "8", "9", "10", "11", "12", "13", "14", "15", "16", "17",
"18", "19", "20", "21", "22", "23", "24", "25" }));

        XCoord.addActionListener(new java.awt.event.ActionListener() {

            public void actionPerformed(java.awt.event.ActionEvent evt) {

                XCoordActionPerformed(evt);

            }

        });




        Y.setFont(new java.awt.Font("Tahoma", 0, 12));

        Y.setHorizontalAlignment(javax.swing.SwingConstants.CENTER);

        Y.setText("Y");
```

```
        Y.setHorizontalTextPosition(javax.swing.SwingConstants.CENTER);



        YCoord.setModel(new javax.swing.DefaultComboBoxModel(new String[] { "0",
"1", "2", "3", "4", "5", "6", "7", "8", "9", "10", "11", "12", "13", "14", "15", "16", "17",
"18", "19", "20", "21", "22", "23", "24", "25" }));

        YCoord.addActionListener(new java.awt.event.ActionListener() {

            public void actionPerformed(java.awt.event.ActionEvent evt) {

                YCoordActionPerformed(evt);

            }

        });



        GetJonSwapData.setText("Get JonSwap Data");

        GetJonSwapData.addActionListener(new java.awt.event.ActionListener() {

            public void actionPerformed(java.awt.event.ActionEvent evt) {

                GetJonSwapDataActionPerformed(evt);
```

```
    }

});



GetPiersonData.setText("Get Pierson Data");

GetPiersonData.addActionListener(new java.awt.event.ActionListener() {

    public void actionPerformed(java.awt.event.ActionEvent evt) {

        GetPiersonDataActionPerformed(evt);

    }

});



WaveSpeed.setHorizontalAlignment(javax.swing.SwingConstants.CENTER);

WaveSpeed.setText("Wave Speed");

WaveSpeed.setHorizontalTextPosition(javax.swing.SwingConstants.CENTER);
```

```
        WaveHeight.setHorizontalAlignment(javax.swing.SwingConstants.CENTER);

        WaveHeight.setText("Wave Amplitude");

        WaveHeight.setHorizontalTextPosition(javax.swing.SwingConstants.CENTER);



        WavePeriod.setHorizontalAlignment(javax.swing.SwingConstants.CENTER);

        WavePeriod.setText("Wave Period");

        WavePeriod.setHorizontalTextPosition(javax.swing.SwingConstants.CENTER);



        WaveLength.setHorizontalAlignment(javax.swing.SwingConstants.CENTER);

        WaveLength.setText("Wave Length");

        WaveLength.setHorizontalTextPosition(javax.swing.SwingConstants.CENTER);



        javax.swing.GroupLayout layout = new
    javax.swing.GroupLayout(getContentPane());
```

```
        getContentPane().setLayout(layout);

        layout.setHorizontalGroup(

            layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)

            .addGroup(javax.swing.GroupLayout.Alignment.TRAILING,
layout.createSequentialGroup()

                .addContainerGap()

.addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADIN
G)

                    .addComponent(Fetch, javax.swing.GroupLayout.DEFAULT_SIZE, 51,
Short.MAX_VALUE)

                    .addComponent(FetchBox, javax.swing.GroupLayout.PREFERRED_SIZE,
51, javax.swing.GroupLayout.PREFERRED_SIZE))

.addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)

.addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADIN
G, false)

                    .addComponent(WindSpeedBox)
```

.addComponent(WindSpeed, javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE))

.addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)

.addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING, false)

.addComponent(WaterDepthBox)

.addComponent(WaterDepth, javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE))

.addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)

.addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING, false)

.addComponent(WindAngleBox)

.addComponent(WindAngle, javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE))

.addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)

.addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADIN
G, false)

.addComponent(WaveSpeedBox)

.addComponent(WaveSpeed, javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE))

.addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)

.addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADIN
G, false)

.addComponent(WaveHeightBox)

.addComponent(WaveHeight, javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE))

.addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)

.addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADIN
G, false)

.addComponent(WaveLengthBox)

.addComponent(WaveLength, javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE))

.addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)

.addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING, false)

.addComponent(WavePeriodBox)

.addComponent(WavePeriod, javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE))

.addGap(29, 29, 29)

.addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)

.addGroup(javax.swing.GroupLayout.Alignment.TRAILING, layout.createSequentialGroup()

.addComponent(X, javax.swing.GroupLayout.DEFAULT_SIZE, 37, Short.MAX_VALUE)

.addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)

```
                .addComponent(Y, javax.swing.GroupLayout.PREFERRED_SIZE, 37,
javax.swing.GroupLayout.PREFERRED_SIZE)

.addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED))

            .addGroup(layout.createSequentialGroup()

                .addComponent(XCoord, javax.swing.GroupLayout.PREFERRED_SIZE,
37, javax.swing.GroupLayout.PREFERRED_SIZE)

.addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)

                .addComponent(YCoord, javax.swing.GroupLayout.PREFERRED_SIZE,
37, javax.swing.GroupLayout.PREFERRED_SIZE)))

        .addGap(14, 14, 14)

.addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.TRAILI
NG)

            .addComponent(GetPiersonData,
javax.swing.GroupLayout.PREFERRED_SIZE, 132,
javax.swing.GroupLayout.PREFERRED_SIZE)

            .addComponent(GetJonSwapData,
javax.swing.GroupLayout.PREFERRED_SIZE, 132,
javax.swing.GroupLayout.PREFERRED_SIZE)))
```

```
        .addComponent(jSeparator1, javax.swing.GroupLayout.Alignment.TRAILING,
javax.swing.GroupLayout.DEFAULT_SIZE, 795, Short.MAX_VALUE)

    );

    layout.setVerticalGroup(

        layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)

        .addGroup(javax.swing.GroupLayout.Alignment.TRAILING,
layout.createSequentialGroup()

            .addContainerGap(464, Short.MAX_VALUE)

            .addComponent(jSeparator1, javax.swing.GroupLayout.PREFERRED_SIZE,
12, javax.swing.GroupLayout.PREFERRED_SIZE)

.addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)

.addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADIN
G)

.addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.BASELI
NE)

                .addComponent(WindSpeed)
```

```
                .addComponent(WaterDepth)

                .addComponent(WindAngle)

                .addComponent(WaveSpeed)

                .addComponent(Fetch))

            .addGroup(layout.createSequentialGroup()

.addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.BASELI
NE)

                .addComponent(WaveHeight)

                .addComponent(WaveLength)

                .addComponent(WavePeriod))

.addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)

.addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.BASELI
NE)

                .addComponent(WindSpeedBox,
javax.swing.GroupLayout.PREFERRED_SIZE,
```

```
javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.PREFERRED_SIZE)

                .addComponent(WaterDepthBox,
javax.swing.GroupLayout.PREFERRED_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.PREFERRED_SIZE)

                .addComponent(WindAngleBox,
javax.swing.GroupLayout.PREFERRED_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.PREFERRED_SIZE)

                .addComponent(FetchBox,
javax.swing.GroupLayout.PREFERRED_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.PREFERRED_SIZE)

                .addComponent(WaveSpeedBox,
javax.swing.GroupLayout.PREFERRED_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.PREFERRED_SIZE)

                .addComponent(WaveHeightBox,
javax.swing.GroupLayout.PREFERRED_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.PREFERRED_SIZE)

                .addComponent(WaveLengthBox,
javax.swing.GroupLayout.PREFERRED_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.PREFERRED_SIZE)

                .addComponent(WavePeriodBox,
javax.swing.GroupLayout.PREFERRED_SIZE,
```

javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.PREFERRED_SIZE)))

.addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADIN
G)

.addGroup(javax.swing.GroupLayout.Alignment.TRAILING,
layout.createSequentialGroup()

.addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.TRAILI
NG)

.addGroup(javax.swing.GroupLayout.Alignment.LEADING,
layout.createSequentialGroup()

.addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.BASELI
NE)

.addComponent(X)

.addComponent(Y))

.addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)

.addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.BASELI
NE)

```
                        .addComponent(XCoord,
javax.swing.GroupLayout.PREFERRED_SIZE, 20,
javax.swing.GroupLayout.PREFERRED_SIZE)


                        .addComponent(YCoord,
javax.swing.GroupLayout.PREFERRED_SIZE, 20,
javax.swing.GroupLayout.PREFERRED_SIZE)))


                    .addComponent(GetJonSwapData,
javax.swing.GroupLayout.Alignment.LEADING))


                .addGap(8, 8, 8))


            .addComponent(GetPiersonData,
javax.swing.GroupLayout.Alignment.TRAILING)))


        .addContainerGap())


    );


    pack();


}// </editor-fold>




private void GetPiersonDataActionPerformed(java.awt.event.ActionEvent evt) {


    piersonequations pWave = new piersonequations();
```

```
double angle = pWave.getAngle(x,y);

WindAngleBox.setText(angle+" Degrees");



double wiSpeed = pWave.getWindSpeed(x,y);

WindSpeedBox.setText(wiSpeed+"m/s");



double period = pWave.getPeriod(x,y);

WavePeriodBox.setText(period+"s");



double length = pWave.getLength(x,y);

WaveLengthBox.setText(length+"m");
```

```
    double waSpeed = pWave.getWaveSpeed(x,y);

    WaveSpeedBox.setText(waSpeed+"m/s");



    double amplitude = pWave.getAmplitude(x,y);

    WaveHeightBox.setText(amplitude+"m");



    FetchBox.setText("N/A");



    WaterDepthBox.setText("N/A");

}



private void XCoordActionPerformed(java.awt.event.ActionEvent evt) {

    x = XCoord.getSelectedIndex();

}
```

```
private void YCoordActionPerformed(java.awt.event.ActionEvent evt) {

    y = YCoord.getSelectedIndex();

}



private void GetJonSwapDataActionPerformed(java.awt.event.ActionEvent evt) {

    jonswapequations jsWave = new jonswapequations();

    jsWave.lengthandwavespeed();

    double fetch = jsWave.getFetch(x,y);

    FetchBox.setText(fetch+"m");



    double depth = jsWave.getDepth(x,y);

    WaterDepthBox.setText(depth+"m");
```

```
double angle = jsWave.getAngle(x,y);

WindAngleBox.setText(angle+" Degrees");



double wiSpeed = jsWave.getWindSpeed(x,y);

WindSpeedBox.setText(wiSpeed+"m/s");



double waSpeed = jsWave.getWaveSpeed(x,y);

WaveSpeedBox.setText(waSpeed+"m/s");



double period = jsWave.getPeriod(x,y);

WavePeriodBox.setText(period+"s");



double length = jsWave.getWaveLength(x,y);
```

```
    WaveLengthBox.setText(length+"m");



    double amplitude = jsWave.getAmplitude(x,y);

    WaveHeightBox.setText(amplitude+"m");



}



/**

 * @param args the command line arguments

 */

public static void main(String args[]) {

    java.awt.EventQueue.invokeLater(new Runnable() {

        public void run() {
```

```
            new GUI().setVisible(true);

        }

    });

}
```

```
// Variables declaration - do not modify

private javax.swing.JLabel Fetch;

private javax.swing.JTextField FetchBox;

private javax.swing.JButton GetJonSwapData;

private javax.swing.JButton GetPiersonData;

private javax.swing.JLabel WaterDepth;

private javax.swing.JTextField WaterDepthBox;
```

```
private javax.swing.JLabel WaveHeight;

private javax.swing.JTextField WaveHeightBox;

private javax.swing.JLabel WaveLength;

private javax.swing.JTextField WaveLengthBox;

private javax.swing.JLabel WavePeriod;

private javax.swing.JTextField WavePeriodBox;

private javax.swing.JLabel WaveSpeed;

private javax.swing.JTextField WaveSpeedBox;

private javax.swing.JLabel WindAngle;

private javax.swing.JTextField WindAngleBox;

private javax.swing.JLabel WindSpeed;

private javax.swing.JTextField WindSpeedBox;

private javax.swing.JLabel X;
```

```
    private javax.swing.JComboBox XCoord;

    private javax.swing.JLabel Y;

    private javax.swing.JComboBox YCoord;

    private javax.swing.JSeparator jSeparator1;

    // End of variables declaration


}
```

Appendix D: DhbMath

```
import java.io.PrintStream;

/**

 * This class implements additional mathematical functions

 * and determines the parameters of the floating point representation.

 *

 * @author Didier H. Besset

 */

public final class DhbMath

{

        /**

         * Typical meaningful precision for numerical calculations.

         */

        static private double defaultNumericalPrecision = 0;
```

```
/**
 * Typical meaningful small number for numerical calculations.
 */
static private double smallNumber = 0;
/**
 * Radix used by floating-point numbers.
 */
static private int radix = 0;
/**
 * Largest positive value which, when added to 1.0, yields 0.
 */
static private double machinePrecision = 0;
/**
 * Largest positive value which, when subtracted to 1.0, yields 0.
```

```
 */

static private double negativeMachinePrecision = 0;

/**

 * Smallest number different from zero.

 */

static private double smallestNumber = 0;

/**

 * Largest possible number

 */

static private double largestNumber = 0;

/**

 * Largest argument for the exponential

 */
```

```
        static private double largestExponentialArgument = 0;

        /**

         * Values used to compute human readable scales.

         */

        private static final double scales[] = {1.25, 2, 2.5, 4, 5, 7.5, 8, 10};

        private static final double semiIntegerScales[] = {2, 2.5, 4, 5, 7.5, 8, 10};

        private static final double integerScales[] = {2, 4, 5, 8, 10};



private static void computeLargestNumber()

{

        double floatingRadix = getRadix();

        double fullMantissaNumber = 1.0d -

                                                floatingRadix *
getNegativeMachinePrecision();
```

```
        while ( !Double.isInfinite( fullMantissaNumber) )
        {
                largestNumber = fullMantissaNumber;
                fullMantissaNumber *= floatingRadix;
        }
}
private static void computeMachinePrecision()
{
        double floatingRadix = getRadix();
        double inverseRadix = 1.0d / floatingRadix;
        machinePrecision = 1.0d;
        double tmp = 1.0d + machinePrecision;
        while ( tmp - 1.0d != 0.0d)
        {
```

```
            machinePrecision *= inverseRadix;

            tmp = 1.0d + machinePrecision;

        }

}

private static void computeNegativeMachinePrecision()

{

        double floatingRadix = getRadix();

        double inverseRadix = 1.0d / floatingRadix;

        negativeMachinePrecision = 1.0d;

        double tmp = 1.0d - negativeMachinePrecision;

        while ( tmp - 1.0d != 0.0d)

        {

            negativeMachinePrecision *= inverseRadix;
```

```
                tmp = 1.0d - negativeMachinePrecision;

        }

}

private static void computeRadix()

{

        double a = 1.0d;

        double tmp1, tmp2;

        do { a += a;

                tmp1 = a + 1.0d;

                tmp2 = tmp1 - a;

               } while ( tmp2 - 1.0d != 0.0d);

        double b = 1.0d;

        while ( radix == 0)

        {
```

```
                b += b;

                tmp1 = a + b;

                radix = (int) ( tmp1 - a);

        }

}

private static void computeSmallestNumber()

{

        double floatingRadix = getRadix();

        double inverseRadix = 1.0d / floatingRadix;

        double fullMantissaNumber = 1.0d - floatingRadix *
getNegativeMachinePrecision();

        while ( fullMantissaNumber != 0.0d )

        {

                smallestNumber = fullMantissaNumber;
```

```
                fullMantissaNumber *= inverseRadix;

        }

}

public static double defaultNumericalPrecision()

{

        if ( defaultNumericalPrecision == 0 )

                defaultNumericalPrecision = Math.sqrt( getMachinePrecision());

        return defaultNumericalPrecision;

}

/**

 * @return boolean    true if the difference between a and b is

 * less than the default numerical precision

 * @param a double
```

```
 * @param b double

 */

public static boolean equal( double a, double b)

{

        return equal( a, b, defaultNumericalPrecision());

}

/**

 * @return boolean    true if the relative difference between a and b

 * is less than precision

 * @param a double

 * @param b double

 * @param precision double

 */

public static boolean equal( double a, double b, double precision)
```

```
{
        double norm = Math.max( Math.abs(a), Math.abs( b));
        return norm < precision || Math.abs( a - b) < precision * norm;
}
public static double getLargestExponentialArgument()
{
        if ( largestExponentialArgument == 0 )
                largestExponentialArgument = Math.log(getLargestNumber());
        return largestExponentialArgument;
}
/**
 * (c) Copyrights Didier BESSET, 1999, all rights reserved.
 */
```

```
public static double getLargestNumber()

{

        if ( largestNumber == 0 )

                computeLargestNumber();

        return largestNumber;

}

public static double getMachinePrecision()

{

        if ( machinePrecision == 0 )

                computeMachinePrecision();

        return machinePrecision;

}

public static double getNegativeMachinePrecision()

{
```

```
        if ( negativeMachinePrecision == 0 )

                computeNegativeMachinePrecision();

        return negativeMachinePrecision;

}

public static int getRadix()

{

        if ( radix == 0 )

                computeRadix();

        return radix;

}

public static double getSmallestNumber()

{

        if ( smallestNumber == 0 )
```

```
                computeSmallestNumber();

        return smallestNumber;

}

public static void printParameters( PrintStream printStream)

{

        printStream.println( "Floating-point machine parameters");

        printStream.println( "----------------------------------");

        printStream.println( " ");

        printStream.println( "radix = "+ getRadix());

        printStream.println( "Machine precision = "

                                                                        +
getMachinePrecision());

        printStream.println( "Negative machine precision = "

                                                                +
getNegativeMachinePrecision());
```

```
        printStream.println( "Smallest number = "+ getSmallestNumber());

        printStream.println( "Largest number = "+ getLargestNumber());

        return;

}

public static void reset()

{

        defaultNumericalPrecision = 0;

        smallNumber = 0;

        radix = 0;

        machinePrecision = 0;

        negativeMachinePrecision = 0;

        smallestNumber = 0;

        largestNumber = 0;

}
```

```
/**
 * This method returns the specified value rounded to
 * the nearest integer multiple of the specified scale.
 *
 * @param value number to be rounded
 * @param scale defining the rounding scale
 * @return rounded value
 */
public static double roundTo(  double value, double scale)
{
        return Math.round( value / scale) * scale;
}
        /**
```

```
        * Round the specified value upward to the next scale value.

        * @param the value to be rounded.

        * @param a fag specified whether integer scale are used, otherwise double
scale is used.

        * @return a number rounded upward to the next scale value.

        */

        public static double roundToScale( double value, boolean integerValued)

        {

                double[] scaleValues;

                int orderOfMagnitude = (int) Math.floor( Math.log( value) / Math.log(
10.0));

                if ( integerValued )

                {

                        orderOfMagnitude = Math.max( 1, orderOfMagnitude);

                        if ( orderOfMagnitude == 1)
```

```
                scaleValues = integerScales;

        else if ( orderOfMagnitude == 2)

                scaleValues = semiIntegerScales;

        else

                scaleValues = scales;

}

else

        scaleValues = scales;

double exponent = Math.pow( 10.0, orderOfMagnitude);

double rValue = value / exponent;

for ( int n = 0; n < scaleValues.length; n++)

{

        if ( rValue <= scaleValues[n])

                return scaleValues[n] * exponent;
```

```
                }

                return exponent;        // Should never reach here

        }

/**

 * (c) Copyrights Didier BESSET, 1999, all rights reserved.

 * @return double

 */

public static double smallNumber()

{

        if ( smallNumber == 0 )

                smallNumber = Math.sqrt( getSmallestNumber());

        return smallNumber;

}
```

```
}
```

Appendix E: FunctionalIterator

```
public abstract class FunctionalIterator extends IterativeProcess

{

        /**

         * Best approximation of the zero.

         */

        protected double result = Double.NaN;

        /**

         * Function for which the zero will be found.

         */

        protected OneVariableFunction f;

/**

 * Generic constructor.

 * @param func OneVariableFunction
```

```
 * @param start double

 */

public FunctionalIterator(OneVariableFunction func)

{

        setFunction( func);

}

/**

* Returns the result (assuming convergence has been attained).

*/

public double getResult( )

{

        return result;

}

/**
```

```
 * @return double

 * @param epsilon double

 */

public double relativePrecision( double epsilon)

{

        return relativePrecision( epsilon, Math.abs( result));

}

/**

 * @param func DhbInterfaces.OneVariableFunction

 */

public void setFunction( OneVariableFunction func)

{

        f = func;
```

```
}

/**

 * @param x double

 */

public void setInitialValue( double x)

{

        result = x;

}

}
```

Appendix F: FunctionDerivative

```
public final class FunctionDerivative implements OneVariableFunction

{

        /**

         * Function for which the derivative is computed.

         */

        private OneVariableFunction f;

        /**

         * Relative interval variation to compute derivative.

         */

         private double relativePrecision = 0.0001;




/**
```

```
* Constructor method.

* @param func the function for which the derivative is computed.

*/

public FunctionDerivative( OneVariableFunction func)

{

        this( func, 0.000001);

}

/**

* Constructor method.

* @param func the function for which the derivative is computed.

* @param precision the relative step used to compute the derivative.

*/

public FunctionDerivative( OneVariableFunction func, double precision)

{
```

```
        f = func;

        relativePrecision = precision;

}

/**

 * Returns the value of the function's derivative

 * for the specified variable value.

 */

public double value( double x)

{

        double x1 = x == 0 ? relativePrecision

                                        : x * ( 1 + relativePrecision);

        double x2 = 2 * x - x1;

        return (f.value(x1) - f.value(x2)) / (x1 - x2);
```

```
}


}
```

Appendix G: IterativeProcess

```
public abstract class IterativeProcess

{

        /**

         * Number of iterations performed.

         */

        private int iterations;

        /**

         * Maximum allowed number of iterations.

         */

        private int maximumIterations = 50;

        /**

         * Desired precision.

         */
```

```
        private double desiredPrecision = DhbMath.defaultNumericalPrecision();

        /**
         * Achieved precision.
         */
        private double precision;



/**
 * Generic constructor.
 */
public IterativeProcess() {
}
/**
* Performs the iterative process.
```

```
* Note: this method does not return anything because Java does not

* allow mixing double, int, or objects

*/

public void evaluate()

{

        iterations = 0;

        initializeIterations();

        while ( iterations++ < maximumIterations )

        {

                precision = evaluateIteration();

                if ( hasConverged() )

                        break;

        }
```

```
            finalizeIterations();

}

/**

* Evaluate the result of the current interation.

* @return the estimated precision of the result.

*/

abstract public double evaluateIteration();

/**

 * Perform eventual clean-up operations

 * (mustbe implement by subclass when needed).

 */

public void finalizeIterations ( )

{

}
```

```
/**
 * Returns the desired precision.
 */
public double getDesiredPrecision( )
{
        return desiredPrecision;
}
/**
 * Returns the number of iterations performed.
 */
public int getIterations()
{
        return iterations;
```

```
}

/**

 * Returns the maximum allowed number of iterations.

 */

public int getMaximumIterations( )

{

        return maximumIterations;

}

/**

 * Returns the attained precision.

 */

public double getPrecision()

{

        return precision;
```

```
}

/**

 * Check to see if the result has been attained.

 * @return boolean

 */

public boolean hasConverged()

{

        return precision < desiredPrecision;

}

/**

* Initializes internal parameters to start the iterative process.

*/

public void initializeIterations()
```

```
{

}

/**

 * @return double

 * @param epsilon double

 * @param x double

 */

public double relativePrecision( double epsilon, double x)

{

        return x > DhbMath.defaultNumericalPrecision()

                                                                                ?
epsilon / x: epsilon;

}

/**
```

```
 * Defines the desired precision.

 */

public void setDesiredPrecision( double prec )

                                                        throws
IllegalArgumentException

{

        if ( prec <= 0 )

                throw new IllegalArgumentException

                                                        ( "Non-positive precision:
"+prec);

        desiredPrecision = prec;

}

/**

 * Defines the maximum allowed number of iterations.

 */
```

```
public void setMaximumIterations( int maxIter)

                                                                    throws
IllegalArgumentException

{

        if ( maxIter < 1 )

                throw new IllegalArgumentException

                                                        ( "Non-positive maximum iteration:
"+maxIter);

        maximumIterations = maxIter;

}

}
```

Appendix H: NewtonZeroFinder

```
public class NewtonZeroFinder extends FunctionalIterator

{

        /**

         * Derivative of the function for which the zero will be found.

         */

        private OneVariableFunction df;

/**

* Constructor method.

* @param func the function for which the zero will be found.

* @param start the initial value for the search.

*/

public NewtonZeroFinder( OneVariableFunction func, double start)

{
```

```
        super( func);

        setStartingValue( start);

}

/**

* Constructor method.

* @param func the function for which the zero will be found.

* @param dFunc derivative of func.

* @param start the initial value for the search.

*/

public NewtonZeroFinder( OneVariableFunction func,

                                                OneVariableFunction dFunc,
double start)

                                                        throws
IllegalArgumentException

{
```

```
        this( func, start);

        setDerivative( dFunc);

}

/**

* Evaluate the result of the current interation.

* @return the estimated precision of the result.

*/

public double evaluateIteration()

{

        double delta = f.value( result) / df.value( result);

        result -= delta;

        return relativePrecision( Math.abs( delta));

}

/**
```

```
* Initializes internal parameters to start the iterative process.

* Assigns default derivative if necessary.

*/

public void initializeIterations()

{

        if ( df == null)

                df = new FunctionDerivative( f);

        if ( Double.isNaN( result) )

                result = 0;

        int n = 0;

        while ( DhbMath.equal( df.value( result), 0) )

        {

                if ( ++n > getMaximumIterations() )
```

```
                    break;

                result += Math.random();

            }

}

/**

 * (c) Copyrights Didier BESSET, 1999, all rights reserved.

 * @param dFunc DhbInterfaces.OneVariableFunction

 * @exception java.lang.IllegalArgumentException

 *                                          if the derivative is not accurate.

 */

public void setDerivative( OneVariableFunction dFunc)

                                                        throws
IllegalArgumentException

{
```

```
        df = new FunctionDerivative( f);

        if ( !DhbMath.equal( df.value( result), dFunc.value( result), 0.001) )

                throw new IllegalArgumentException

                                                ( "Supplied derative function is
inaccurate");

        df = dFunc;

}

/**

 * (c) Copyrights Didier BESSET, 1999, all rights reserved.

 */

public void setFunction( OneVariableFunction func)

{

        super.setFunction( func);

        df = null;
```

```
}

/**

* Defines the initial value for the search.

*/

public void setStartingValue( double start)

{

        result = start;

}

}
```

Appendix I: OneVariableFunction

```
public interface OneVariableFunction

{









/**

 * Returns the value of the function for the specified variable value.

 */

public double value( double x);

}
```

Appendix J: PolynomialFunction

```
import java.util.Vector;

import java.util.Enumeration;

public class PolynomialFunction implements OneVariableFunction

{

        /**

         * Polynomial coefficients.

         */

        private double[] coefficients;




/**

 * Constructor method.

 * @param coeffs polynomial coefficients.
```

```
 */

public PolynomialFunction( double[] coeffs)

{

        coefficients = coeffs;

}

/**

 *

 * @param r double   number added to the polynomial.

 * @return DhbFunctionEvaluation.PolynomialFunction

 */

public PolynomialFunction add( double r)

{

        int n = coefficients.length;

        double coef[] = new double[n];
```

```
        coef[0] = coefficients[0] + r;

        for ( int i = 1; i < n; i++)

                coef[i] = coefficients[i];

        return new PolynomialFunction( coef);

}

/**

 *

 * @param p DhbFunctionEvaluation.PolynomialFunction

 * @return DhbFunctionEvaluation.PolynomialFunction

 */

public PolynomialFunction add( PolynomialFunction p)

{

        int n = Math.max( p.degree(), degree()) + 1;
```

```
        double[] coef = new double[n];

        for ( int i = 0; i < n; i++ )

                coef[i] = coefficient(i) + p.coefficient(i);

        return new PolynomialFunction( coef);

}

/**

 * Returns the coefficient value at the desired position

 * @param n int          the position of the coefficient to be returned

 * @return double the coefficient value

 * @version 1.2

 */

public double coefficient( int n)

{

        return n < coefficients.length ? coefficients[n] : 0;
```

```
}

/**

 *

 * @param r double    a root of the polynomial (no check made).

 * @return PolynomialFunction the receiver divided by polynomial (x - r).

 */

public PolynomialFunction deflate( double r)

{

        int n = degree();

        double remainder = coefficients[n];

        double[] coef = new double[n];

        for ( int k = n - 1; k >= 0; k--)

        {
```

```
                coef[k] = remainder;

                remainder = remainder * r + coefficients[k];

        }

        return new PolynomialFunction( coef);

}

/**

 * Returns degree of this polynomial function

 * @return int degree of this polynomial function

 */

public int degree()

{

        return coefficients.length - 1;

}

/**
```

```
 * Returns the derivative of the receiver.

 * @return PolynomialFunction derivative of the receiver.

 */

public PolynomialFunction derivative()

{

        int n = degree();

        if ( n == 0 )

        {

                double coef[] = {0};

                return new PolynomialFunction( coef);

        }

        double coef[] = new double[n];

        for ( int i = 1; i <= n; i++)
```

```
                coef[i-1] = coefficients[i]*i;

        return new PolynomialFunction( coef);

}

/**

 *

 * @param r double

 * @return DhbFunctionEvaluation.PolynomialFunction

 */

public PolynomialFunction divide( double r)

{

        return multiply( 1 / r);

}

/**

 *
```

```
 * @param r double

 * @return DhbFunctionEvaluation.PolynomialFunction

 */

public PolynomialFunction divide( PolynomialFunction p)

{

        return divideWithRemainder(p)[0];

}

/**

 *

 * @param r double

 * @return DhbFunctionEvaluation.PolynomialFunction

 */

public PolynomialFunction[] divideWithRemainder( PolynomialFunction p)
```

```
{

        PolynomialFunction[] answer = new PolynomialFunction[2];

        int m = degree();

        int n = p.degree();

        if ( m < n )

        {

                double[] q = {0};

                answer[0] = new PolynomialFunction( q);

                answer[1] = p;

                return answer;

        }

        double[] quotient = new double[ m - n + 1];

        double[] coef = new double[ m + 1];

        for ( int k = 0; k <= m; k++ )
```

```
        coef[k] = coefficients[k];

double norm = 1 / p.coefficient( n);

for ( int k = m - n; k >= 0; k--)

{

        quotient[k] = coef[ n + k] * norm;

        for ( int j = n + k - 1; j >= k; j--)

                coef[j] -= quotient[k] * p.coefficient(j-k);

}

double[] remainder = new double[n];

for ( int k = 0; k < n; k++)

        remainder[k] = coef[k];

answer[0] = new PolynomialFunction( quotient);

answer[1] = new PolynomialFunction( remainder);
```

```
        return answer;

}

/**

 * Returns the integral of the receiver having the value 0 for X = 0.

 * @return PolynomialFunction integral of the receiver.

 */

public PolynomialFunction integral( )

{

        return integral( 0);

}

/**

 * Returns the integral of the receiver having the specified value for X = 0.

 * @param value double       value of the integral at x=0

 * @return PolynomialFunction integral of the receiver.
```

```
 */

public PolynomialFunction integral( double value)

{

        int n = coefficients.length + 1;

        double coef[] = new double[n];

        coef[0] = value;

        for ( int i = 1; i < n; i++)

                coef[i] = coefficients[i-1]/i;

        return new PolynomialFunction( coef);

}

/**

 *

 * @param r double
```

```
 * @return DhbFunctionEvaluation.PolynomialFunction

 */

public PolynomialFunction multiply( double r)

{

        int n = coefficients.length;

        double coef[] = new double[n];

        for ( int i = 0; i < n; i++)

                coef[i] = coefficients[i] * r;

        return new PolynomialFunction( coef);

}

/**

 *

 * @param p DhbFunctionEvaluation.PolynomialFunction

 * @return DhbFunctionEvaluation.PolynomialFunction
```

```
 */

public PolynomialFunction multiply( PolynomialFunction p)

{

        int n = p.degree() + degree();

        double[] coef = new double[n + 1];

        for ( int i = 0; i <= n; i++)

        {

                coef[i] = 0;

                for ( int k = 0; k <= i; k++)

                        coef[i] += p.coefficient(k) * coefficient(i-k);

        }

        return new PolynomialFunction( coef);

}
```

```
/**
 *
 * @return double[]
 */
public double[] roots()
{
        return roots( DhbMath.defaultNumericalPrecision());
}
/**
 *
 * @param desiredPrecision double
 * @return double[]
 */
public double[] roots( double desiredPrecision)
```

```
{

        PolynomialFunction dp = derivative();

        double start = 0;

        while ( Math.abs( dp.value( start)) < desiredPrecision )

                start = Math.random();

        PolynomialFunction p = this;

        NewtonZeroFinder rootFinder = new NewtonZeroFinder( this, dp, start);

        rootFinder.setDesiredPrecision( desiredPrecision);

        Vector rootCollection = new Vector( degree());

        while ( true)

        {

                rootFinder.evaluate();

                if ( !rootFinder.hasConverged() )
```

```
                break;

        double r = rootFinder.getResult();

        rootCollection.addElement( new Double( r));

        p = p.deflate( r);

        if ( p.degree() == 0 )

                break;

        rootFinder.setFunction( p);

        try { rootFinder.setDerivative( p.derivative());}

                        catch ( IllegalArgumentException e) {};

}

double[] roots = new double[ rootCollection.size()];

Enumeration e = rootCollection.elements();

int n = 0;

while ( e.hasMoreElements() )
```

```
        {
                roots[n++] = ( (Double) e.nextElement()).doubleValue();
        }
        return roots;
}
/**
 *
 * @param p DhbFunctionEvaluation.PolynomialFunction
 * @return DhbFunctionEvaluation.PolynomialFunction
 */
public PolynomialFunction subtract( double r)
{
        return add( -r);
```

```
}

/**

 *

 * @return DhbFunctionEvaluation.PolynomialFunction

 * @param p DhbFunctionEvaluation.PolynomialFunction

 */

public PolynomialFunction subtract( PolynomialFunction p)

{

        int n = Math.max( p.degree(), degree()) + 1;

        double[] coef = new double[n];

        for ( int i = 0; i < n; i++ )

                coef[i] = coefficient(i) - p.coefficient(i);

        return new PolynomialFunction( coef);

}
```

```
/**

* Returns a string representing the receiver

*/

public String toString()

{

        StringBuffer sb = new StringBuffer();

        boolean firstNonZeroCoefficientPrinted = false;

        for ( int n = 0; n < coefficients.length; n++)

        {

                if ( coefficients[n] != 0 )

                {

                        if ( firstNonZeroCoefficientPrinted)

                                sb.append( coefficients[n] > 0 ? " + " : " ");
```

```
                else

                        firstNonZeroCoefficientPrinted = true;

                if ( n == 0 || coefficients[n] != 1)

                        sb.append( Double.toString( coefficients[n]) );

                if ( n > 0 )

                        sb.append( " X^"+n);

            }

        }

        return sb.toString();

}

/**

 * Returns the value of the polynomial for the specified variable value.

 * @param x double   value at which the polynomial is evaluated

 * @return double polynomial value.
```

```
 */

public double value( double x)

{

        int n = coefficients.length;

        double answer = coefficients[--n];

        while ( n > 0 )

                answer = answer * x + coefficients[--n];

        return answer;

}

/**

 * Returns the value and the derivative of the polynomial

 * for the specified variable value in an array of two elements

 * @version 1.2
```

```
 * @param x double   value at which the polynomial is evaluated

 * @return double[0]   the value of the polynomial

 * @return double[1]   the derivative of the polynomial

 */

public double[] valueAndDerivative( double x)

{

        int n = coefficients.length;

        double[] answer = new double[2];

        answer[0] = coefficients[--n];

        answer[1] = 0;

        while ( n > 0 )

        {

                answer[1] = answer[1] * x + answer[0];

                answer[0] = answer[0] * x + coefficients[--n];
```

```
        }

        return answer;

}

}
```

Appendix J: DhbMath

```
import java.io.PrintStream;
/**
 * This class implements additional mathematical functions
 * and determines the parameters of the floating point representation.
 *
 * @author Didier H. Besset
 */
public final class DhbMath
{
	/**
	 * Typical meaningful precision for numerical calculations.
	 */
	static private double defaultNumericalPrecision = 0;
	/**
	 * Typical meaningful small number for numerical calculations.
	 */
	static private double smallNumber = 0;
	/**
	 * Radix used by floating-point numbers.
	 */
	static private int radix = 0;
	/**
	 * Largest positive value which, when added to 1.0, yields 0.
	 */
	static private double machinePrecision = 0;
	/**
	 * Largest positive value which, when subtracted to 1.0, yields
0.
	 */
	static private double negativeMachinePrecision = 0;
	/**
	 * Smallest number different from zero.
	 */
	static private double smallestNumber = 0;
	/**
	 * Largest possible number
	 */
	static private double largestNumber = 0;
	/**
	 * Largest argument for the exponential
	 */
	static private double largestExponentialArgument = 0;
	/**
	 * Values used to compute human readable scales.
	 */
	private static final double scales[] = {1.25, 2, 2.5, 4, 5, 7.5,
8, 10};
	private static final double semiIntegerScales[] = {2, 2.5, 4, 5,
7.5, 8, 10};
	private static final double integerScales[] = {2, 4, 5, 8, 10};

private static void computeLargestNumber()
{
	double floatingRadix = getRadix();
```

```
      double fullMantissaNumber = 1.0d -
                                        floatingRadix *
getNegativeMachinePrecision();
      while ( !Double.isInfinite( fullMantissaNumber) )
      {
            largestNumber = fullMantissaNumber;
            fullMantissaNumber *= floatingRadix;
      }
}
private static void computeMachinePrecision()
{
      double floatingRadix = getRadix();
      double inverseRadix = 1.0d / floatingRadix;
      machinePrecision = 1.0d;
      double tmp = 1.0d + machinePrecision;
      while ( tmp - 1.0d != 0.0d)
      {
            machinePrecision *= inverseRadix;
            tmp = 1.0d + machinePrecision;
      }
}
private static void computeNegativeMachinePrecision()
{
      double floatingRadix = getRadix();
      double inverseRadix = 1.0d / floatingRadix;
      negativeMachinePrecision = 1.0d;
      double tmp = 1.0d - negativeMachinePrecision;
      while ( tmp - 1.0d != 0.0d)
      {
            negativeMachinePrecision *= inverseRadix;
            tmp = 1.0d - negativeMachinePrecision;
      }
}
private static void computeRadix()
{
      double a = 1.0d;
      double tmp1, tmp2;
      do { a += a;
             tmp1 = a + 1.0d;
             tmp2 = tmp1 - a;
            } while ( tmp2 - 1.0d != 0.0d);
      double b = 1.0d;
      while ( radix == 0)
      {
            b += b;
            tmp1 = a + b;
            radix = (int) ( tmp1 - a);
      }
}
private static void computeSmallestNumber()
{
      double floatingRadix = getRadix();
      double inverseRadix = 1.0d / floatingRadix;
      double fullMantissaNumber = 1.0d - floatingRadix *
getNegativeMachinePrecision();
      while ( fullMantissaNumber != 0.0d )
      {
```

```
            smallestNumber = fullMantissaNumber;
            fullMantissaNumber *= inverseRadix;
      }
}
public static double defaultNumericalPrecision()
{
      if ( defaultNumericalPrecision == 0 )
            defaultNumericalPrecision = Math.sqrt(
getMachinePrecision());
      return defaultNumericalPrecision;
}
/**
 * @return boolean      true if the difference between a and b is
 * less than the default numerical precision
 * @param a double
 * @param b double
 */
public static boolean equal( double a, double b)
{
      return equal( a, b, defaultNumericalPrecision());
}
/**
 * @return boolean      true if the relative difference between a and b
 * is less than precision
 * @param a double
 * @param b double
 * @param precision double
 */
public static boolean equal( double a, double b, double precision)
{
      double norm = Math.max( Math.abs(a), Math.abs( b));
      return norm < precision || Math.abs( a - b) < precision * norm;
}
public static double getLargestExponentialArgument()
{
      if ( largestExponentialArgument == 0 )
            largestExponentialArgument = Math.log(getLargestNumber());
      return largestExponentialArgument;
}
/**
 * (c) Copyrights Didier BESSET, 1999, all rights reserved.
 */
public static double getLargestNumber()
{
      if ( largestNumber == 0 )
            computeLargestNumber();
      return largestNumber;
}
public static double getMachinePrecision()
{
      if ( machinePrecision == 0 )
            computeMachinePrecision();
      return machinePrecision;
}
public static double getNegativeMachinePrecision()
{
      if ( negativeMachinePrecision == 0 )
```

```
            computeNegativeMachinePrecision();
      return negativeMachinePrecision;
}
public static int getRadix()
{
      if ( radix == 0 )
            computeRadix();
      return radix;
}
public static double getSmallestNumber()
{
      if ( smallestNumber == 0 )
            computeSmallestNumber();
      return smallestNumber;
}
public static void printParameters( PrintStream printStream)
{
      printStream.println( "Floating-point machine parameters");
      printStream.println( "--------------------------------");
      printStream.println( " ");
      printStream.println( "radix = "+ getRadix());
      printStream.println( "Machine precision = "
                                                              +
getMachinePrecision());
      printStream.println( "Negative machine precision = "
                                                        +
getNegativeMachinePrecision());
      printStream.println( "Smallest number = "+ getSmallestNumber());
      printStream.println( "Largest number = "+ getLargestNumber());
      return;
}
public static void reset()
{
      defaultNumericalPrecision = 0;
      smallNumber = 0;
      radix = 0;
      machinePrecision = 0;
      negativeMachinePrecision = 0;
      smallestNumber = 0;
      largestNumber = 0;
}
/**
 * This method returns the specified value rounded to
 * the nearest integer multiple of the specified scale.
 *
 * @param value number to be rounded
 * @param scale defining the rounding scale
 * @return rounded value
 */
public static double roundTo(  double value, double scale)
{
      return Math.round( value / scale) * scale;
}
      /**
       * Round the specified value upward to the next scale value.
       * @param the value to be rounded.
```

```
       * @param a fag specified whether integer scale are used,
otherwise double scale is used.
       * @return a number rounded upward to the next scale value.
       */
      public static double roundToScale( double value, boolean
integerValued)
      {
            double[] scaleValues;
            int orderOfMagnitude = (int) Math.floor( Math.log( value) /
Math.log( 10.0));
            if ( integerValued )
            {
                  orderOfMagnitude = Math.max( 1, orderOfMagnitude);
                  if ( orderOfMagnitude == 1)
                        scaleValues = integerScales;
                  else if ( orderOfMagnitude == 2)
                        scaleValues = semiIntegerScales;
                  else
                        scaleValues = scales;
            }
            else
                  scaleValues = scales;
            double exponent = Math.pow( 10.0, orderOfMagnitude);
            double rValue = value / exponent;
            for ( int n = 0; n < scaleValues.length; n++)
            {
                  if ( rValue <= scaleValues[n])
                        return scaleValues[n] * exponent;
            }
            return exponent;  // Should never reach here
      }
/**
 * (c) Copyrights Didier BESSET, 1999, all rights reserved.
 * @return double
 */
public static double smallNumber()
{
      if ( smallNumber == 0 )
            smallNumber = Math.sqrt( getSmallestNumber());
      return smallNumber;
}
}
```

Appendix K: stokesdriver

```
public class stokesdriver
{
	public static void main(String[] args)
	{
		//piersonequations pierson = new piersonequations();
		// pierson.getstokes();

		 jonswapequations jonswap = new jonswapequations();
		 jonswap.lengthandwavespeed();
	}

}
```

Appendix L: jonswapequations (oil program)

```
import java.util.Scanner;
import java.util.Random;
public class jonswapequations
{
	Scanner in;
	private double waveinfo[][][];
	private double a0,a1,a2,a3,a4;
	private int i;
	private int j;

	private double[][][] again;

	private double angle;
	private double f; // fetch
	private double u; // wind speed
	private double g; // gravity
	private double w; // angular frequency = 2 * PI * wave frequency
	private double T; // Period of the wave
	private double L; // Length of the wave
	private double c; // phase speed
	private double c2; // group speed
	private double cTMA;

	private double cshallow; // group and phase speed for shallow
water, d < L/11
	private double d; // water depth
	private double a; // wave height, calculated from the wind eqs
	// to be the significant wave height,
	// which is the average height of the largest 1/3 of waves in
that spectrum
	private double aTMA; // wave height from new equations

	private double v; // Stokes drift velocity

	private double vshallow;
	private double t;
	private double totalt;
	private double distance;
	private int maxx; // maximum number of cells in the x direction
	private int maxy;// maximum number of cells in the y direction
	private int z;


	public jonswapequations()
	{
		in = new Scanner(System.in);

		g = 9.81;
		i = 0;
		j = 0;

		t = 500000; // time in seconds

		c = g/w;
```

```
		c2 = c/2;

		System.out.println("Enter maximum x distance");
		maxx = (int)in.nextDouble() / 1000;

		System.out.println("Enter maximum y distance");
		maxy = (int)in.nextDouble() / 1000;

		totalt = 0;
	}



	public void lengthandwavespeed()
	{
		Random rand = new Random();

		waveinfo = new double[maxy][maxx][8];
		for (i = 0; i < maxy; i++)
		{
			for(j = 0; j < maxx; j++)
			{

				if (totalt <50000)
				{
					u = rand.nextInt(5);
					angle = rand.nextInt(180);
					d = rand.nextInt(200000);
					f = rand.nextInt(1000);
					z = rand.nextInt(2);

					if (angle > 0 && angle < 22.5)
						angle = 0;
					else if (angle > 22.5 && angle < 67.5)
						angle = 45;
					else if (angle > 67.5 && angle < 112.5)
						angle = 90;
					else if (angle > 112.5 && angle < 157.5)
						angle = 135;
					else if (angle > 157.5 && angle < 202.5)
						angle = 180;
					else if (angle > 202.5 && angle < 247.5)
						angle = 225;
					else if (angle > 247.5 && angle < 292.5)
						angle = 270;
					else if (angle > 292.5 && angle < 337.5)
						angle = 315;
					else if(angle>337.5 && angle <=360)
						angle = 0;

					waveinfo[i][j][0] = u;
					waveinfo[i][j][1] = angle;
					waveinfo[i][j][2] = d;
					waveinfo[i][j][3] = f;
					waveinfo[i][j][4] = z; // has oil = 0, no
oil = 1
```

```
                        }
                        else
                        {
                              waveinfo[i][j][0] = a0;
                              waveinfo[i][j][1] = a1;
                              waveinfo[i][j][2] = a2;
                              waveinfo[i][j][3] = a3;
                              waveinfo[i][j][4] = a4;
                        }

                  }
            }

            for (i = 0; i < maxy; i++)
            {
                  for(j = 0; j < maxx; j++)
                  {
                        w = 2.84 * Math.pow(g, 0.7) *
Math.pow(waveinfo[i][j][3], -0.3) * Math.pow(waveinfo[i][j][0], -0.4);

                        T = 2 * Math.PI / w;

                        if (waveinfo[i][j][2] > L/4)
                        {
                              L = g * Math.pow(T, 2)/(2 * Math.PI); //
T will change over time as w changes, so L will, also
                              a = .00163 * waveinfo[i][j][0] *
Math.pow(waveinfo[i][j][3]/g, .5);
                              v = (4* Math.pow((Math.PI), 2) *
Math.pow(a, 2))/(L * T);

                              waveinfo[i][j][5] = v;
                              waveinfo[i][j][6] = a;
                              waveinfo[i][j][7] = L;

                        }
                        else if(d<L/11)
                        {
                              // amplitude doesn't apply
                              // wavelength doesn't apply

                              cshallow = Math.pow(d * g,.5);
                              vshallow = cshallow;
                              v = vshallow;
                              waveinfo[i][j][5] = v;
                        }
                        else
                        {
                              double[] coefficients = {-
3.454,0,0,0,2.133333 * Math.pow(Math.PI * waveinfo[i][j][2],4), 0, -4 *
Math.pow(Math.PI * waveinfo[i][j][2], 2) + (.159/(waveinfo[i][j][2] *
L)), 0,1};
                              NewtonZeroFinder zeroFinder2 = new
NewtonZeroFinder(new PolynomialFunction (coefficients),-1);

                              zeroFinder2.evaluate();
```

```
                              double result2  =
zeroFinder2.getResult();

                              cTMA = c * (result2/L);

                              aTMA = 0.24 *
Math.pow(Math.tanh(0.343*Math.pow(waveinfo[i][j][2], 1.14)) *
Math.tanh(.000414 * Math.pow(waveinfo[i][j][3],0.79) / Math.tanh(0.343
* Math.pow(waveinfo[i][j][2],1.14))),0.572);

                              v = (4* Math.pow((Math.PI), 2) *
Math.pow(aTMA, 2))/(result2 * T); // we'll just assume that the same
                              // stokes drift equation, which assumes
depth to be infinite, applies to this transition state, also.

                              waveinfo[i][j][5] = v;

                              waveinfo[i][j][6] = aTMA;

                              L = result2;
                              waveinfo[i][j][7] = L;

                        }

                  }
            }

            for (i = 0; i < maxy; i++)
            {
                  for(j = 0; j < maxx; j++)
                  {
                        if (waveinfo[i][j][4] == 0)
                        {
                              System.out.println("x: " + j*1000 + " y:
" + i*1000);
                              System.out.println(waveinfo[i][j][5]);
                              System.out.println(waveinfo[i][j][1]);
                        }
                  }
            }
            System.out.println();

            again = new double[maxy][maxx][8];
            for (i = 0; i < maxy; i++)
            {
                  for(j = 0; j < maxx; j++)
                  {
                        again[i][j][0]= waveinfo[i][j][0];
                        again[i][j][1]= waveinfo[i][j][1];
                        again[i][j][2]= waveinfo[i][j][2];
                        again[i][j][3]= waveinfo[i][j][3];
                        again[i][j][4]= 1;
                  }
            }

            for (i = 0; i < maxy; i++)
            {
```

```
                  for(j = 0; j < maxx; j++)
                  {
                        if (waveinfo[i][j][4] == 0)
                        {

                              distance = waveinfo[i][j][5] * t;  //
distance in meters

                              if (angle == 0 && distance > 100 && i+1
<maxy)
                              {
                                    again[i+1][j][4] = 0;
                                    again[i][j][4] = 1;
                              }
                              else if (angle == 45 && distance >
100*Math.sqrt(2) && i+1 < maxy && j+1 <maxx)
                              {
                                    again[i+1][j+1][4] = 0;
                                    waveinfo[i][j][4] = 1;
                              }
                              else if (angle == 90 && distance > 100 &&
j+1 < maxx)
                              {
                                    again[i][j+1][4] = 0;
                                    again[i][j][4] = 1;
                              }
                              else if (angle == 135 && distance >
100*Math.sqrt(2) && i-1>=0 && j+1<maxx)
                              {
                                    again[i-1][j+1][4] = 0;
                                    again[i][j][4] = 1;
                              }
                              else if (angle == 180 && distance > 100
&& i-1>=0)
                              {
                                    again[i-1][j][4] = 0;
                                    again[i][j][4] = 1;
                              }
                              else if (angle == 225 && distance >
100*Math.sqrt(2) && i-1>=0 && j-1>=0)
                              {
                                    again[i-1][j-1][4] = 0;
                                    again[i][j][4] = 1;
                              }
                              else if (angle == 270 && distance > 100
&& j-1>=0)
                              {
                                    again[i][j-1][4] = 0;
                                    again[i][j][4] = 1;
                              }
                              else
                              {
                                    if (distance > 100*Math.sqrt(2) &&
i+1<maxy && j-1>=0)
                                    {
                                          again[i+1][j-1][4] = 0;
                                          again[i][j][4] = 1;
```

```
                                    }
                                    else
                                          again[i][j][4] = 0;
                              }
                        }
                  }
            }

            for (i = 0; i < maxy; i++)
            {
                  for(j = 0; j < maxx; j++)
                  {
                        if (again[i][j][4] == 0)
                        {
                              System.out.println("x: " + j*1000 + " y:
" + i*1000);

                              System.out.println(waveinfo[i][j][5]);
                              System.out.println(waveinfo[i][j][1]);
                        }
                  }
            }
            for (i = 0; i < maxy; i++)
            {
                  for(j = 0; j < maxx; j++)
                  {
                        a0 = again[i][j][0];
                        a1 = again[i][j][1];
                        a2 = again[i][j][2];
                        a3 = again[i][j][3];
                        a4 = again[i][j][4];
                  }
            }
            System.out.println();
            totalt = totalt + t;
            while(totalt < 100000)
            {
            lengthandwavespeed();
            }
      }
}
```

Appendix M: Screenshot


```
Jonswapequations class output :

Enter maximum x distance
2000
Enter maximum y distance
2000
x: 1000 y: 1000
2.6293099201400797E-4
45.0

x: 1000 y: 1000
2.6293099201400797E-4
45.0
```

As you can see, oil starts out at the point (1000,1000), the upper right coordinate inside the inputted boundaries. Because the angle of the wind at that point is 45 degrees, the oil should move up 1000m in both the x and y direction. Since this would be out of the boundaries, the oil doesn't move, and the program stops.