# Main Model

```
library(ggplot2)
library(reshape2)
library(grid)
library(gridExtra)
setwd("/Users/Nick/mysisModeling") #Doesnt actually do anything in chunks.
```

---

# Check for the existance of needed datafiles.

If the files are not present, then run their corresponding scripts and generate them.

The corresponding R Markdown scripts associated with these files can be found on RPub:

- Solar Data Cleaning
- Thermocline Model
- Mysocline Depth Model

```
setwd("/Users/Nick/mysisModeling") #Locate ourselves in the computer.

#Check for solar data
if(!file.exists("data/light_day_moon_hour.csv")){ source("dataGen/solarData.r") }

#Check for thermocline data
if(!file.exists("data/Depth_Thermocline_Hour.csv")){ source("dataGen/thermoclineLevels.r") }

#Mysocline data
if(!file.exists("data/mysocline_hour.csv")){ source("dataGen/depthModel.r") }

#Mysocline data
if(!file.exists("data/FoodAvail_Hour.csv")){ source("dataGen/foodAvailability.r") }

#Read in the data now.
depthData = read.csv("data/mysocline_hour.csv")$x
foodCurve = read.csv("data/FoodAvail_hour.csv")
foodAvail = foodCurve$foodAvail
foodVar   = foodCurve$variability
```

# Class and method declarations:

I have chosen to use the S4 object oriented programing style. The details are minor but more info can be found here.

## Set up the mysis class:

This basically is a digital mysis that simplistically holds a couple of peices of information and upon which the model will act.

```
setClass("mysis",
         representation(
           energy    = "numeric",    # The energy reserves are a numeric value
           migrating = "logical",    # If mysis is migrating is a logical value
           alive     = "logical",    # Alive still?
           depth     = "numeric"),
         prototype(
           energy    = 0,            # The default instantiated mysis starts with zero energy...
           migrating = FALSE,        # and not migrating...
           alive     = TRUE,         # alive
           depth     = 100),         # and at the bottom.
         )

#Set the show method, basically how we want the program to display the info about the mysis on calling.
setMethod("show", "mysis",
          function(object){
            if (object@alive){
              print(object@energy)
              print(object@migrating)
              print(object@depth)
            } else {
              print("dead")
            }
          })
```

**Real quick we need to make a model for decisions to migrate based upon condition:**

The form of this model is a logistic curve:

$$f(x) = \frac{1}{1 + e^{-k(x-m)}}$$

Where $k$ is an indicator of steepness, and $m$ is the midepoint of the curve.

```
migrationProb = function(condition){ #I am choosing to leave the random roll outside of script for continuity, could
be changed
  m = 120 #value of curve midpoint
  k = 0.03 #steepness of curve
  x = condition

  dist = 1/(1 + exp(-k* (x - m)))
  return(1 - dist)
}
```

## The decision tree model:

This is a stochastic model, meaning it utilizes randomness to simulate the real world.

The function of the decision tree is such:

- Draw random number to decide if migrating:
    - If the number drawn is above the ratio needed to migrate (food ratio) then the mysis migrates.
    - After migration is decided, another draw is done to see if the mysis is killed by predation.

```
rewardUnits = .67
migrationCost = 20       #How many energy units the mysis use up migrating

setGeneric( "nextTime", function(object, ...){standardGeneric("nextTime")})
setMethod("nextTime","mysis",
          function(object, foodAvail, foodVar, time){       #Takes in the mysis object and the ratio of food quality
at a given time.

            #Run the draws:
            migrationDraw   = runif(1) #random number between 0 and 1, this will be used for the migration decision
            migrationDraw_2 = runif(1) #draw for second migration decision, this time compared to logistic curve
            predationDraw   = runif(1) #" " to see if killed by predation
            #Lets set some constants real quick:
            migrationRisk = 0  #We don't really need to simulate predation.
            stayRisk      = 0
#             migrationRisk = 0.0001   #Chance of being eaten if migrating
#             stayRisk      = 0.00001  #Chance of being eaten if they stay on the bottom
#
            #Energy rewards:
            migrationReward = rnorm(1, (rewardUnits*(1 + foodAvail)), foodVar)
            stayReward      = migrationReward * .2
            threshold = 0.2  #Let's make sure that the benthic reward can never drop below a threshold.
            if(stayReward < threshold){
              stayReward = threshold
            }
```

```
  sunset  = 18 #hardcode sunset, fill this with real data later.
  sunrise = 7
  conditionCurve = migrationProb(object@energy)

  if (object@energy <= 0){ #did the mysis starve?
    object@alive = FALSE
    object@energy = 0
  }
  #if the mysis is in good condition use standard migration chance, if bad use 1 - chance
  if ((time %% 24 == sunset + 1)&&(object@alive)){
    if (migrationDraw < foodAvail) {
      object@migrating = migrationDraw_2 > conditionCurve
      object@energy = object@energy - migrationCost
    } else {
       object@migrating = FALSE
    }
  } else if (time %% 24 == sunrise){ #bring them back down at sunrise.
    object@migrating = FALSE
  }

  #Here is the decision tree:
  if (object@alive){ #If the mysis is alive let's run the decision tree
    if (object@migrating){
      object@depth = depthData[time] #Grab the mysocline limit at this hour

      if (predationDraw > migrationRisk){ #The mysis evades predation
        object@energy = object@energy + migrationReward
      } else { #Mysis is eaten
        object@alive = FALSE
      }
    } else { #The mysis didn't migrate
      object@depth = 100

      if (predationDraw > stayRisk){ #The mysis evades predation
        object@energy = object@energy +  stayReward
      } else { #Mysis is eaten
        object@alive = FALSE
      }
    }
  }
  object} #Return the mysis
)
```

## Testing it:

Now that the model is all set up can run it over a small subset of the eventual numbers (~10,000 mysids over 8,700 hours) and

see what results we get.

```
mysids = NULL
numOfMysids = 15
mysisNames = paste("mysis",(seq(1,numOfMysids)), sep = "")

for (i in 1:numOfMysids){
  initialenergy = rnorm(1,150,25) #draw initial energy from normal dist centered at 30 with stdev of 5.
  mysids = c(mysids, new("mysis", energy = initialenergy) )
}
```

… and now we can run the model on those mysids:

```
migrations = NULL #Initialize some dataframes to keep track of the migrations and conditions values for the mysids.
conditions = NULL
alive      = NULL
hours = 1:(24*365) #150 days
for (mysid in mysids){ #loop through the mysis
  migration = NULL
  condition = NULL
  for (i in hours){
    mysid    = nextTime(mysid, foodAvail[i], foodVar[i] , i)
    migration = c(migration, mysid@depth)
    condition = c(condition, mysid@energy)
    if (i == length(hours)) { alive = c(alive, mysid@alive)}
  }
  migrations = cbind(migrations, migration) # add to the object of migrations.
  conditions = cbind(conditions, condition)
}
```

# What do we have?

Let's plot this small run to see what kind of behavior we are getting.

```
setwd("/Users/Nick/mysisModeling")
migrations = cbind(hours,migrations)

#Make the dataframe for the migrations
migrations_df = as.data.frame(migrations)
names(migrations_df) = c("hours", mysisNames)
migrations_plot = melt(migrations_df, id = 'hours')

#Render the plot for the migrations
# a = ggplot(migrations_plot, aes(x = hours, y = -value, group = variable)) + geom_line(aes(color = variable)) +
#   labs(title = "Mysis migration patterns for first 100 days of year", y = "Depth below surface", x = "") + theme(le
gend.position="none")

#Assemble the dataframe for condition
conditions_df = as.data.frame(cbind(hours, conditions))
names(conditions_df) = c("hours", mysisNames)
conditions_plot = melt(conditions_df, id = 'hours')

#Find portion who died
mysidsWhoDied = tail(conditions_df, 1)[1:numOfMysids+1] <= 0
mortalityRate = sum(mysidsWhoDied)/numOfMysids

subtitleText = paste(numOfMysids, "mysids, Reward:", rewardUnits, "Migration Cost:", migrationCost, ", Mortality Rate
:",mortalityRate)

#Draw condition plot
b = ggplot(conditions_plot, aes(x = hours, y = value, group= variable)) + stat_smooth(aes(color = variable),method =
"gam", formula = y ~ s(x, bs = "cs")) +
  labs(title = "blah!", y = "energy units") + theme(legend.position="none") +  ggtitle(bquote(atop(.("Mysis Condition
 Over Year"), atop(italic(.(subtitleText)), ""))))

b
ggsave(file = paste("figures/condition", rewardUnits, "_", migrationCost,".pdf", sep = ""))
# #Plot both migrations and condition on top of eachother.
# Plot = arrangeGrob(a,b,ncol=1, main = paste("Migration and Condition,", "Mortality Rate:", mortalityRate) )
# Plot
```

# Food Availability Model

*Nick Strayer*

*March 12, 2015*

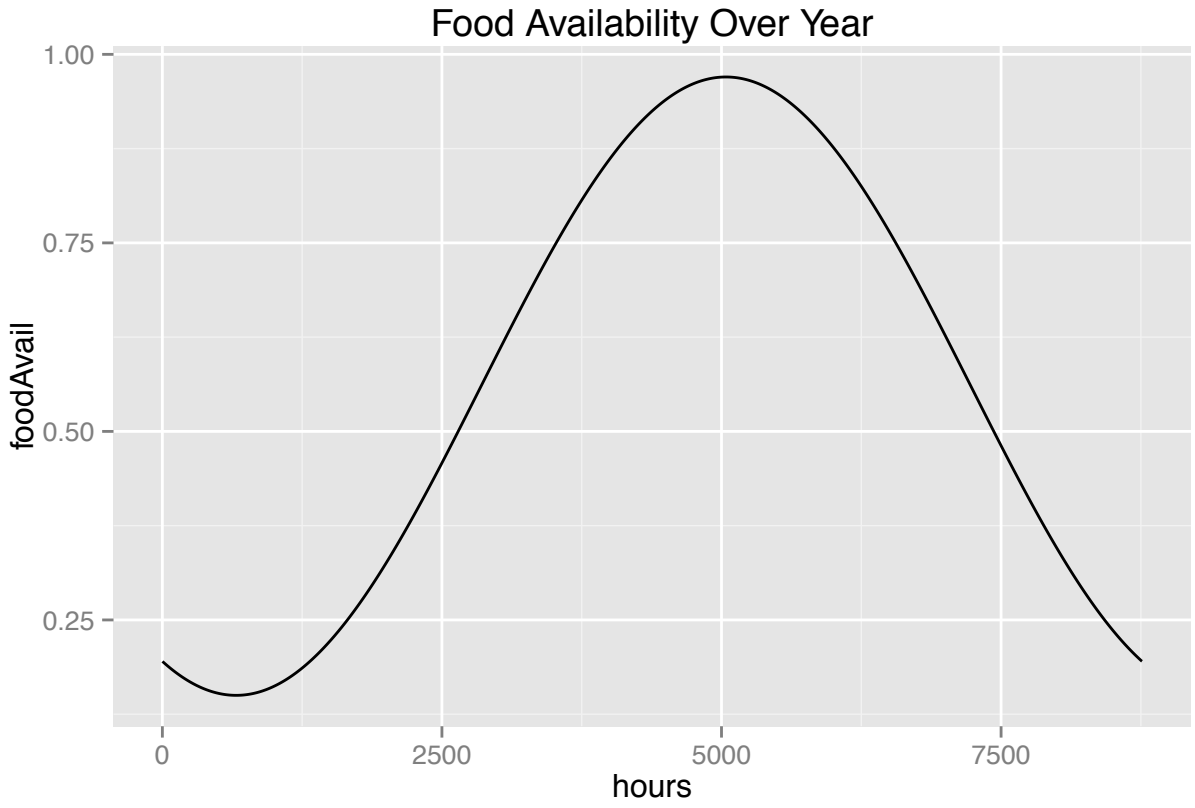
```
library(ggplot2)
# Code for a simple food availability model based upon the assumptions of higher pelagic food quality
# in the summer.
```

### Parameters We Set:

```
#First we set up some user defined variables.
min     = 0.15 #minimum food availability value
max     = 0.97 #max
highDay = 210*24 #Number of days into the year that the max food availability is (226 is aug 15)
august  = 5856 #hours into the year
```

### The actual model

```
#Now we generate the others
scaler    = (max - min)/2 #range divided by two
heightAdj = (max + min)/2 #average value
hours      = 1:(365*24) #Set up a hour vector to loop over
foodAvail = NULL

#Quick loop to generate the data.
for (hour in hours){
  foodAvail = c(foodAvail, scaler * cos( (1/(365*24))*2*pi * (hour - highDay)) + heightAdj)
}

qplot(hours,foodAvail, geom = "line", main = "Food Availability Over Year")
```



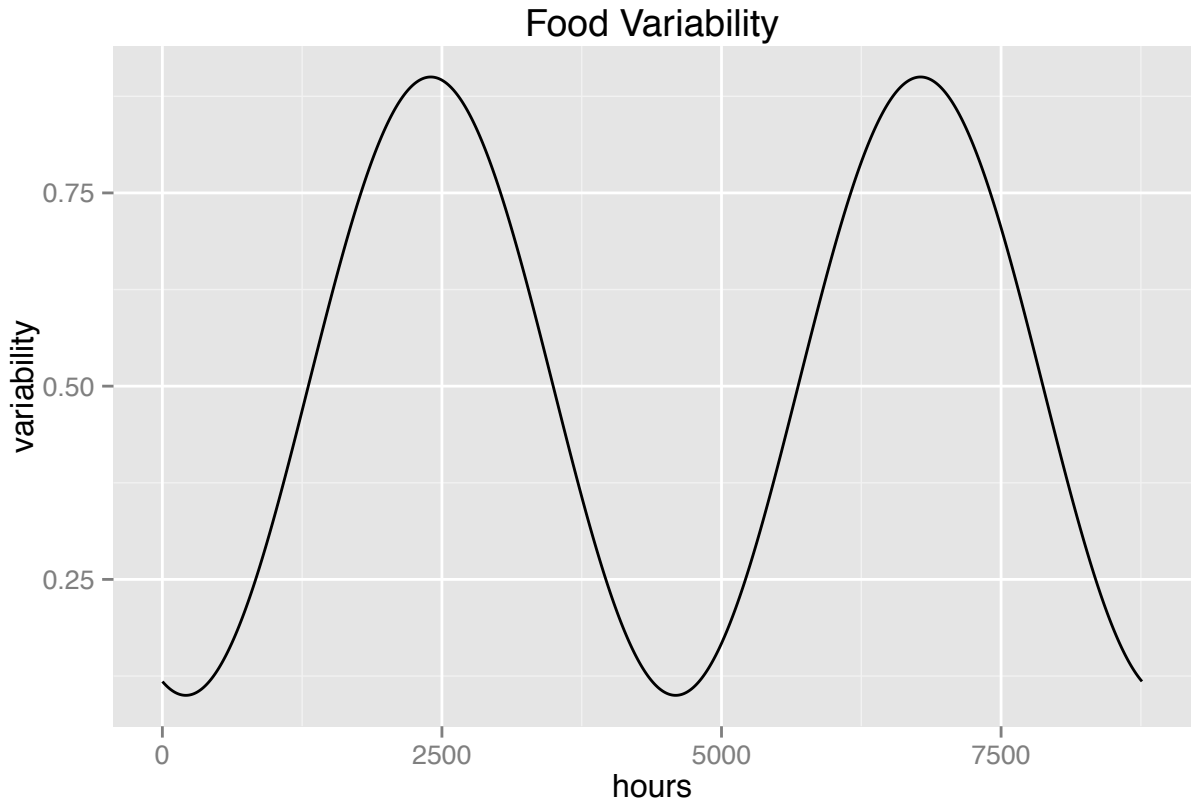

```
setwd("/Users/Nick/mysisModeling/paperMaterials/figures")
ggsave(filename = "pres_foodAvailability.pdf", width = 6, height = 2.5)
```

### The variability curve

Here we set up another curve which represents the variablity of the food distribution at any time.

The value of the curve $f(hour)$ = variability of distribution.

```
min     = 0.1 #minimum food availability value
max     = 0.9 #max
highDay = 100*24
scaler    = (max - min)/2 #range divided by two
heightAdj = (max + min)/2 #average value
variability = NULL

#Quick loop to generate the data.
for (hour in hours){
  variability = c(variability, scaler * cos( (1/(365*24))*4*pi * (hour - highDay)) + heightAdj)
}

qplot(hours,variability, geom = "line", main = "Food Variability")
```

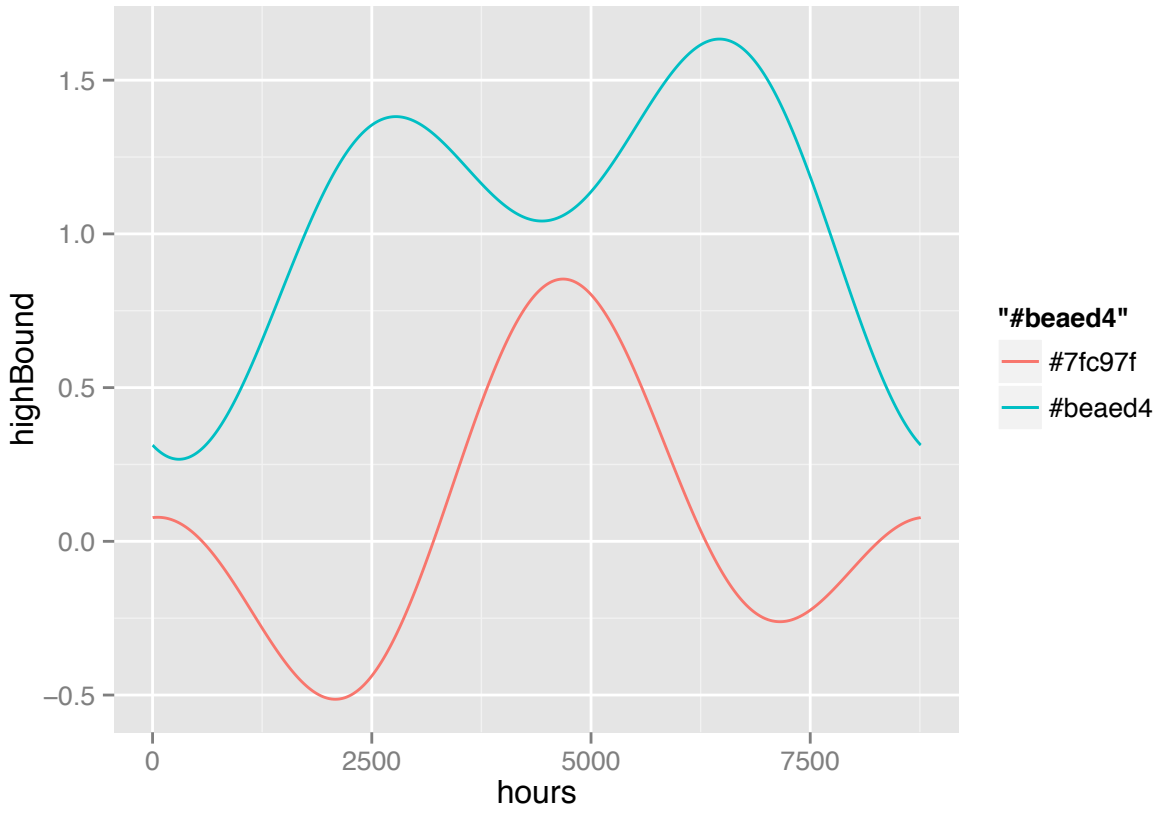



```
setwd("/Users/Nick/mysisModeling/paperMaterials/figures")
ggsave(filename = "pres_foodVariability.pdf", width = 6, height = 2.5)

avail_var_df = as.data.frame(cbind(foodAvail, variability))
```

### Combined curves:

We will now generate the bounding curves by adding and subtracting the variablity from the main curve.

```
highBound = NULL
lowBound  = NULL

for (hour in hours){
  highBound = c(highBound, (foodAvail[hour] + variability[hour]))
  lowBound = c(lowBound, (foodAvail[hour] - variability[hour]))
}

distributionDf = as.data.frame(cbind(hours, highBound, lowBound))

library(ggplot2)

ggplot(distributionDf, aes(hours)) +
  geom_line(aes(y = highBound, colour = "#beaed4")) +
  geom_line(aes(y = lowBound, colour = "#7fc97f"))
```




### Save to file

```
setwd("/Users/Nick/mysisModeling")
#write.csv(distributionDf, "data/FoodAvail_Hour.csv", row.names=FALSE) #The bounded model  #
#write.csv(avail_var_df, "data/FoodAvail_Hour.csv", row.names=FALSE) #The plain side curve.
```

# Thermocline Model

*Nick Strayer*

*March 8, 2015*

---

## Define the model.

In this case we will be using a logistic curve to model the rise and fall of the thermocline

```
thermoclineDist = function(t){
  maxThermDepth = 40
  if (t < (2190*2)) { #Winter + Spring
    n1 = .003
    n2 = -5
    x = t
    dist = (maxThermDepth)/(1 + exp(-(n2 + n1*x)))
  } else { #Summer+ fall
    n1 = .005
    n2 = -8
    x = t - (2190*2)
    dist = (-maxThermDepth)/(1 + exp(-(n2 + n1*x))) + maxThermDepth
  }
  return(dist)
}
```

## Generate data

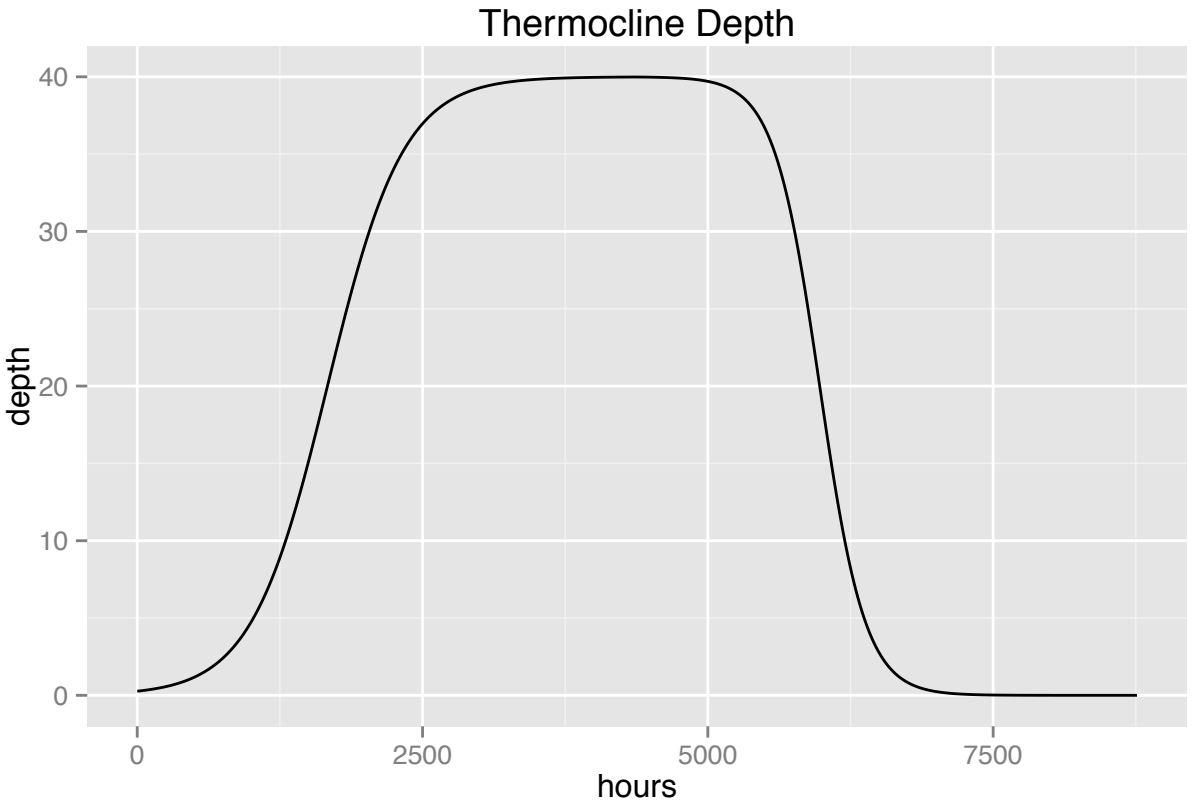
Now we initialize an hour list and run the thermocline function over it to get our data.

```
hours  = 0:(365*24)
dist = NULL

for (t in hours){ dist = c(dist, thermoclineDist(t)) }
```

## Plot and save

Now let's check out the results and save to a csv file.

```
data = cbind(hours, dist) # Wrap the data.

thermocline = as.data.frame(data)
ggplot(thermocline, aes(x = hours, y = dist)) + geom_line() +
  labs(title = "Thermocline Depth", y = "depth")
```




```
ggsave(filename = "../paperMaterials/figures/pres_thermoclineDepth.pdf", width = 6, height = 2.5)
#write.csv(data, "/Users/Nick/mysisModeling/data/Depth_Thermocline_Hour.csv", row.names=FALSE)
```

# Solar Data

*Nick Strayer*

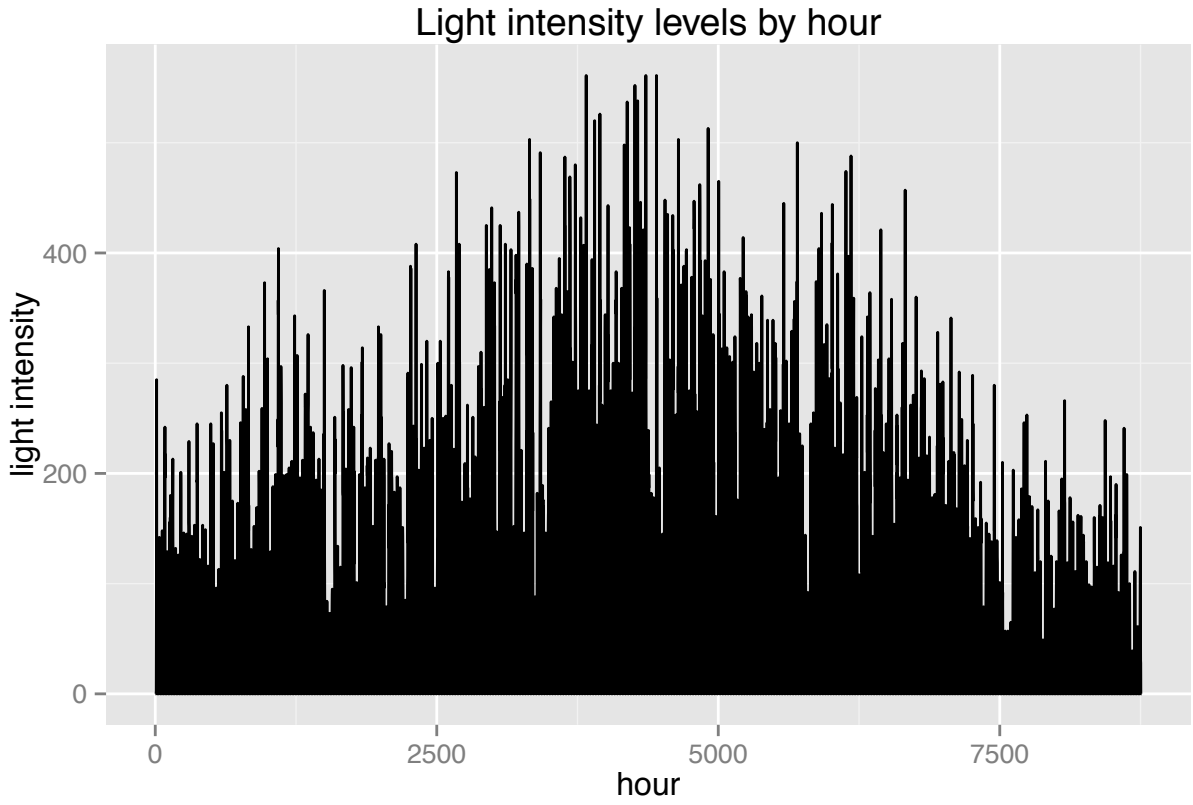
*March 8, 2015*

---

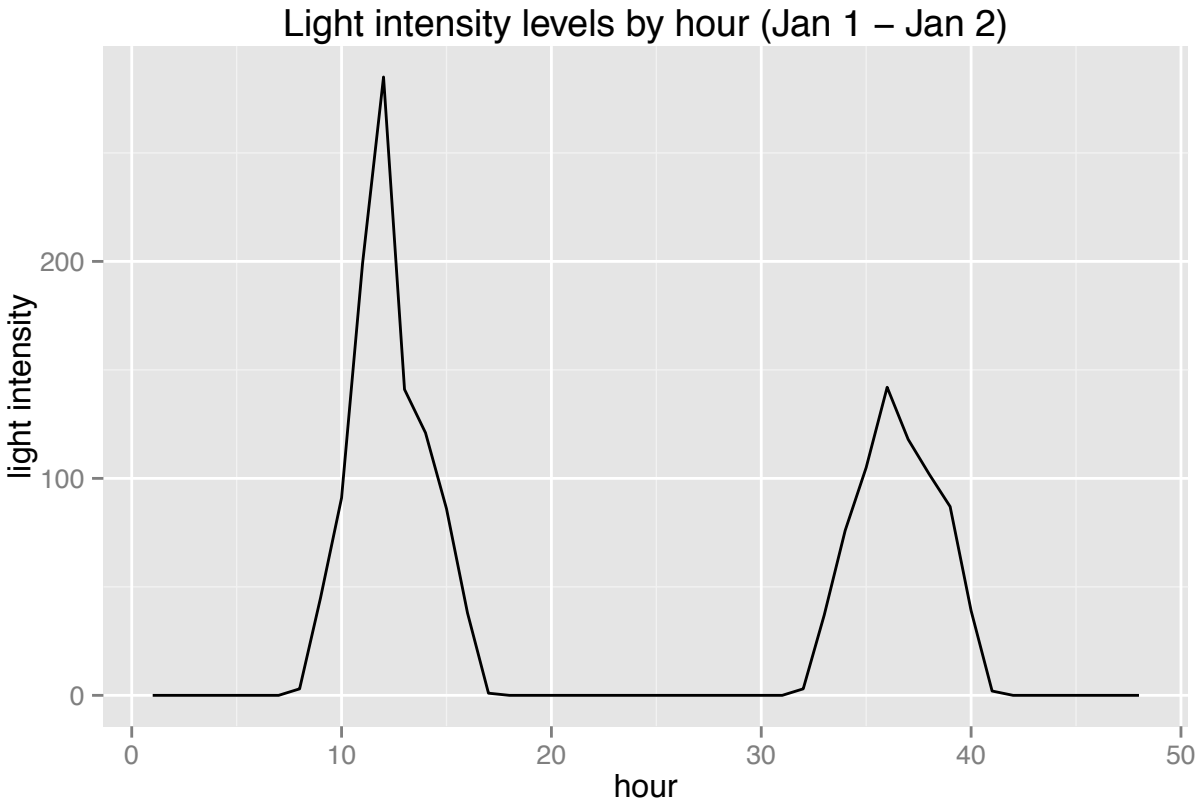
**First we bring in raw data obtained from National Renewable Energy Laboratory**

```
setwd("/Users/Nick/mysisModeling")
solarRaw = read.csv("outsideData/solarData2010.csv")
realSolar = solarRaw$METSTAT.Dif..Wh.m.2.
```

and run a quick check to make sure the data came in properly...




Taking a closer peak at the data we see that the sensor is not actually accurate enough to pick up moonlight...




**Adding the lunar cycle.**

Because of this sensor limitation and the importance of the moon cycle on migration patterns we need to find a way to fill in the missing moon data.

By treating the lunar light intensity as a sinusoidal curve with a period of 27 days and a maximum light intensity of 1 lux (as per this PNAS paper) we can model the moon cycle and then fill in the gaps in the real data with the modeled data.

```
hours = 1:(365*24) # Generate an hours vector
cycle = 27*24 #27 days at 24 hours.

moonLux = NULL #initialize list of moon values

for (hour in hours){ #Loop through the hours of the day to generate data.
  cyclePoint = hour %% cycle
  moonLux = c(moonLux, 0.5 * cos((1/cycle)* 2*pi * cyclePoint ) + .501)
}
```

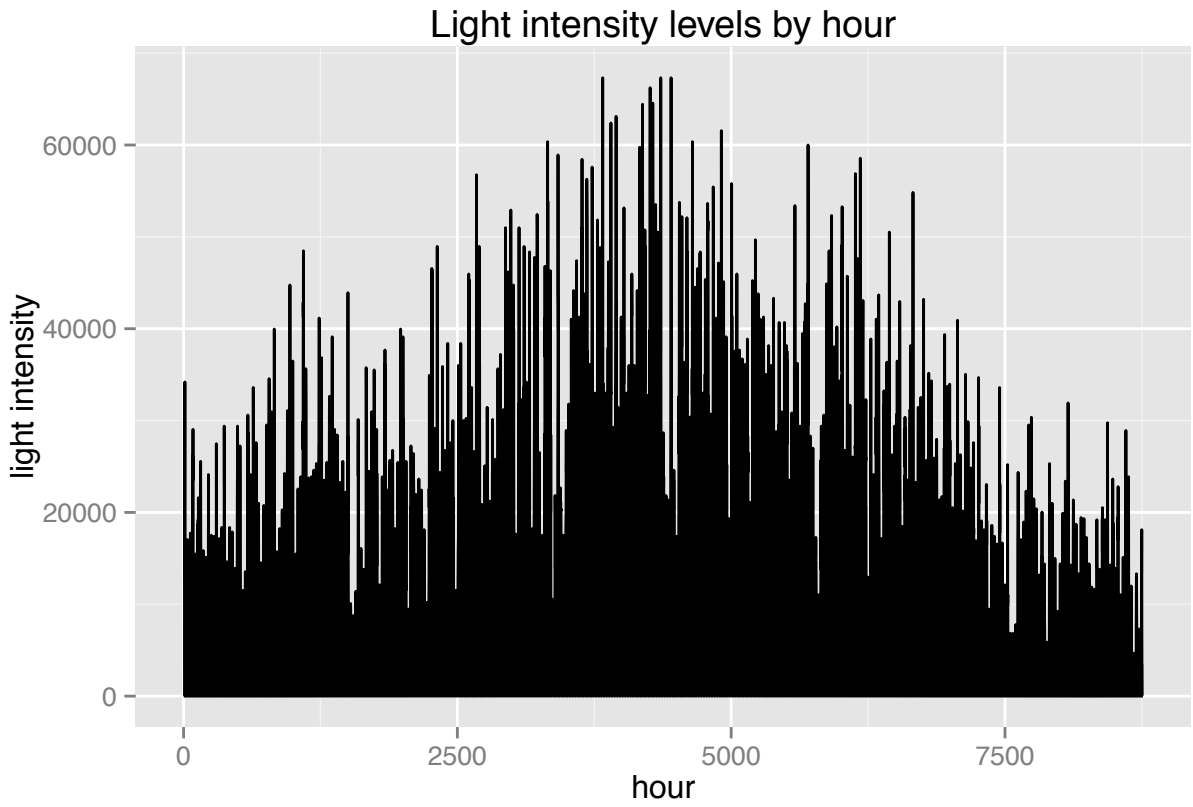
**Combined the data:**

Now that we have the real data in and the modeled lunar cycle we need to combine them to get hour complete solar data.

We are using a conversion ratio of 120 Watt Hours (wh) to a lux as was reported in Jensen et al. 2006 (see papers directory).

```
Wh_to_lux = 120.0; #From Jensen et all.
combinedLight = NULL;
solarLux = NULL;

for (i in hours){
  solarLux = realSolar[i] * Wh_to_lux;
  if (solarLux > moonLux[i]){
    combinedLight = c(combinedLight, solarLux)
  } else {
    combinedLight = c(combinedLight, moonLux[i])
  }
}
```

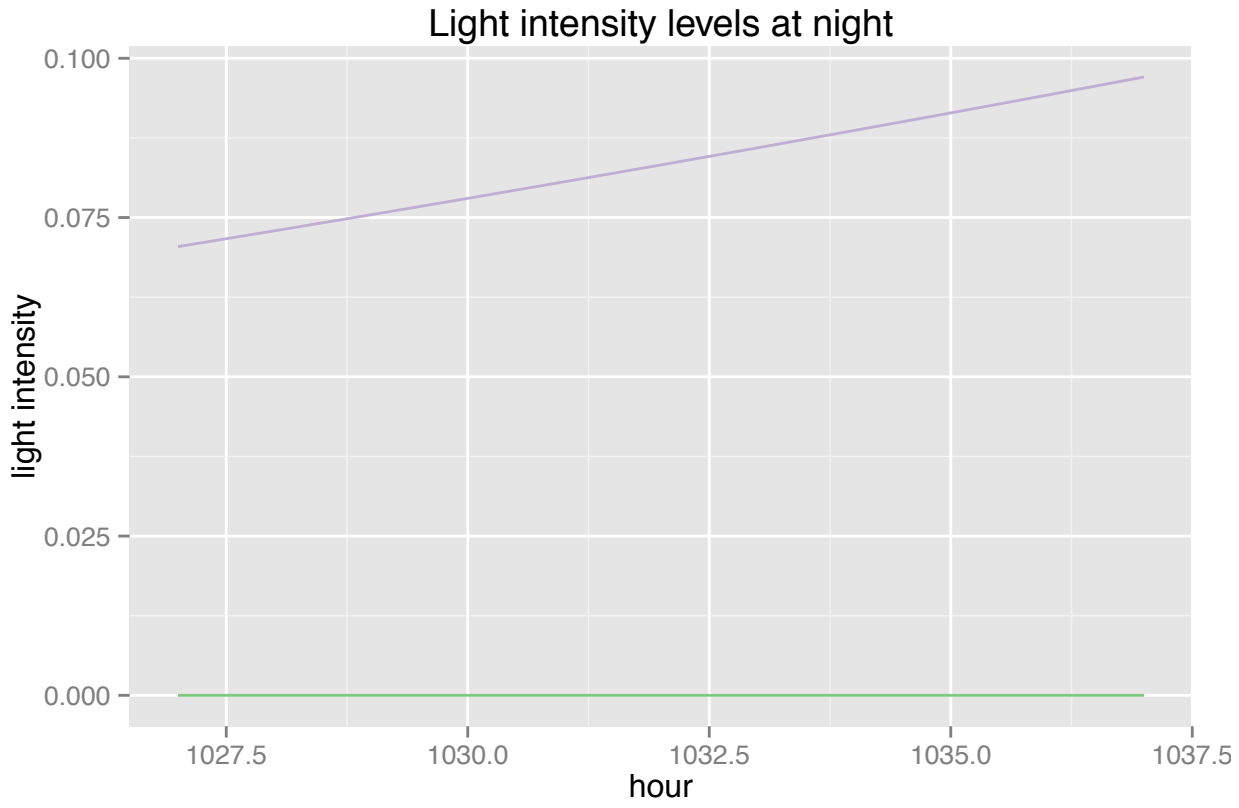
Now we can plot to see if the lunar data successfully made it in:




Well it's pretty hard to tell if anything happened. However, take note of the y-axis. Pretty hard to note a change of $< 1$ lux when it goes up to 60,000.

To check one more time lets zoom in on a given night. In this case between the hours of 1027 and 1037 (a february night).

We can see that the new (purple) data doesn't touch zero unlike the raw (green) data.




Now we can export this data to the data directory.

```
setwd("/Users/Nick/mysisModeling")
write.csv(combinedLight, "data/light_day_moon_hour.csv", row.names=FALSE)
```

# Depth Model

*Nick Strayer*

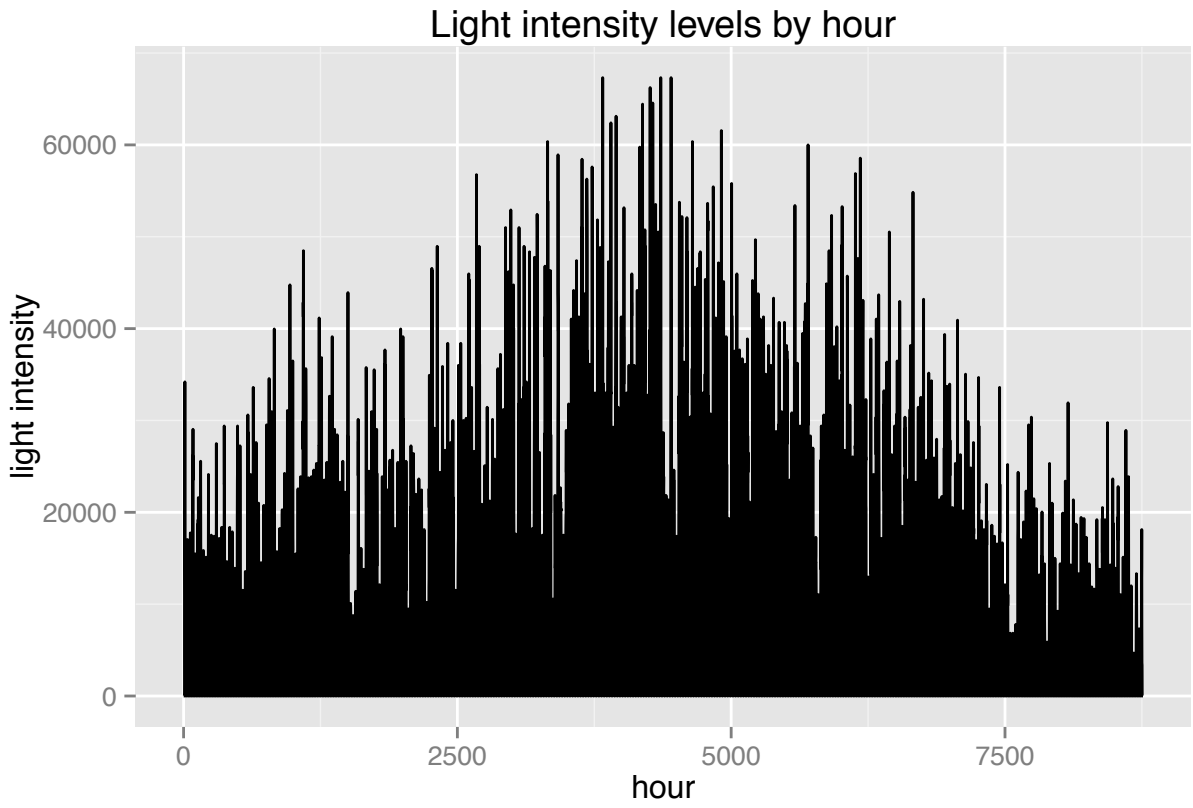
*March 8, 2015*

Just getting some initializing stuff out of the way and bring in some outside data that was generated in `thermoclineLevels.r` and `solarData.r`:

```
setwd("/Users/Nick/mysisModeling")
library(ggplot2)
depthData = read.csv("data/Depth_Thermocline_Hour.csv")$dist
lightData = read.csv("data/light_day_moon_hour.csv")$x
hours = 1:(365*24)
```

Do a quick plot to make sure the data came in properly:

```
lightDataDf = as.data.frame(cbind(hours, lightData))
m = ggplot(lightDataDf, aes(x = hours, y = lightData)) + geom_line()
m + labs(title = "Light intensity levels by hour", x = "hour", y = "light intensity")
```




## Mylux limit model:

Now we code put in the model for depth at mylux intensity limit.

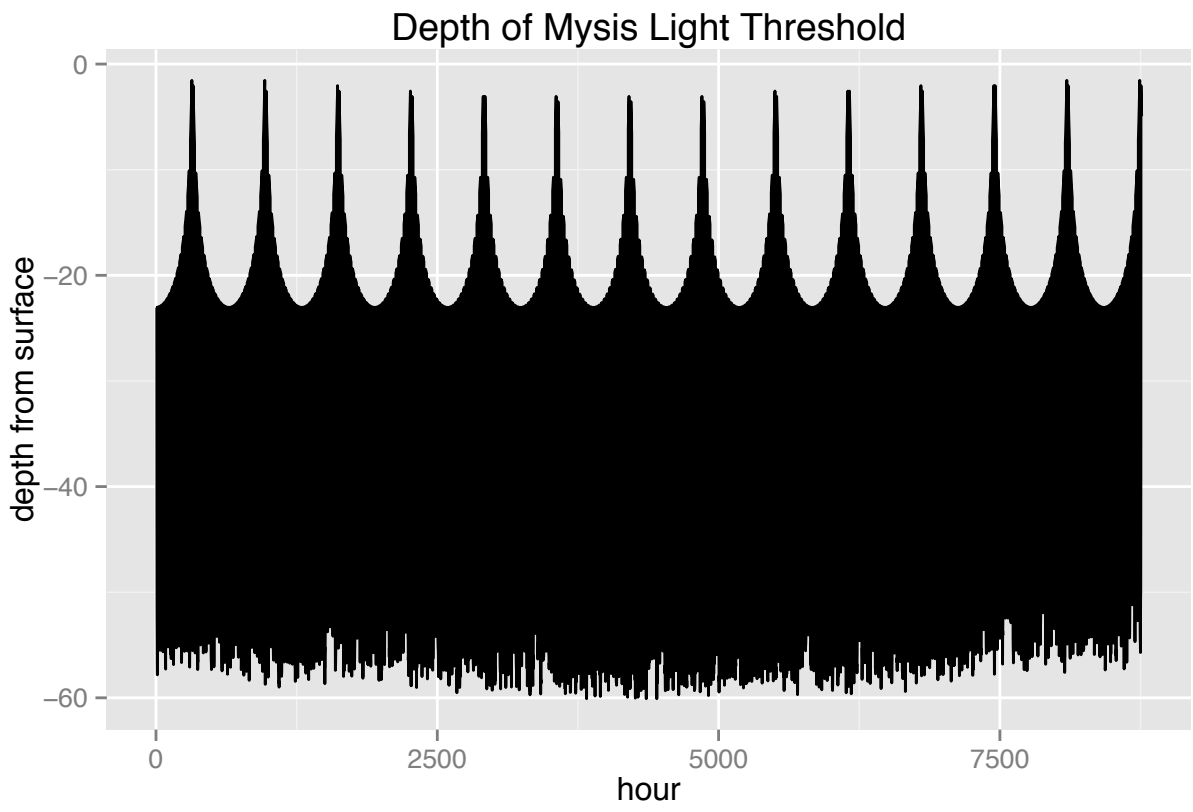
Distance of light threshold: $f(I_o) = \frac{1}{k}(ln(I_o) - ln(I_x))$

```
lightDepth = function(surfaceLight){

  k   = 0.3  #extinction coefficient
  I_x = 0.001 #Mysis light threshold (paper quotes between 10^-2 and 10^-4)

  distance = (1/k) * (log(surfaceLight) - log(I_x))
  if (distance < 0){ distance = 0 }
  return(distance)
}
```

### Mylux limit depth

Now we will run the raw data through our light depth function and plot:

```
isocline = NULL
for (hour in lightData){
  if (hour == 0){
    isocline = c(isocline, 0) #logs dont play nice with 0s
  } else {
    isocline = c(isocline, lightDepth(hour))
  }
}

isoclineDf = as.data.frame(cbind(hours, isocline))
m = ggplot(isoclineDf, aes(x = hours, y = -isocline)) + geom_line()
m + labs(title = "Depth of Mysis Light Threshold", x = "hour", y = "depth from surface")
```

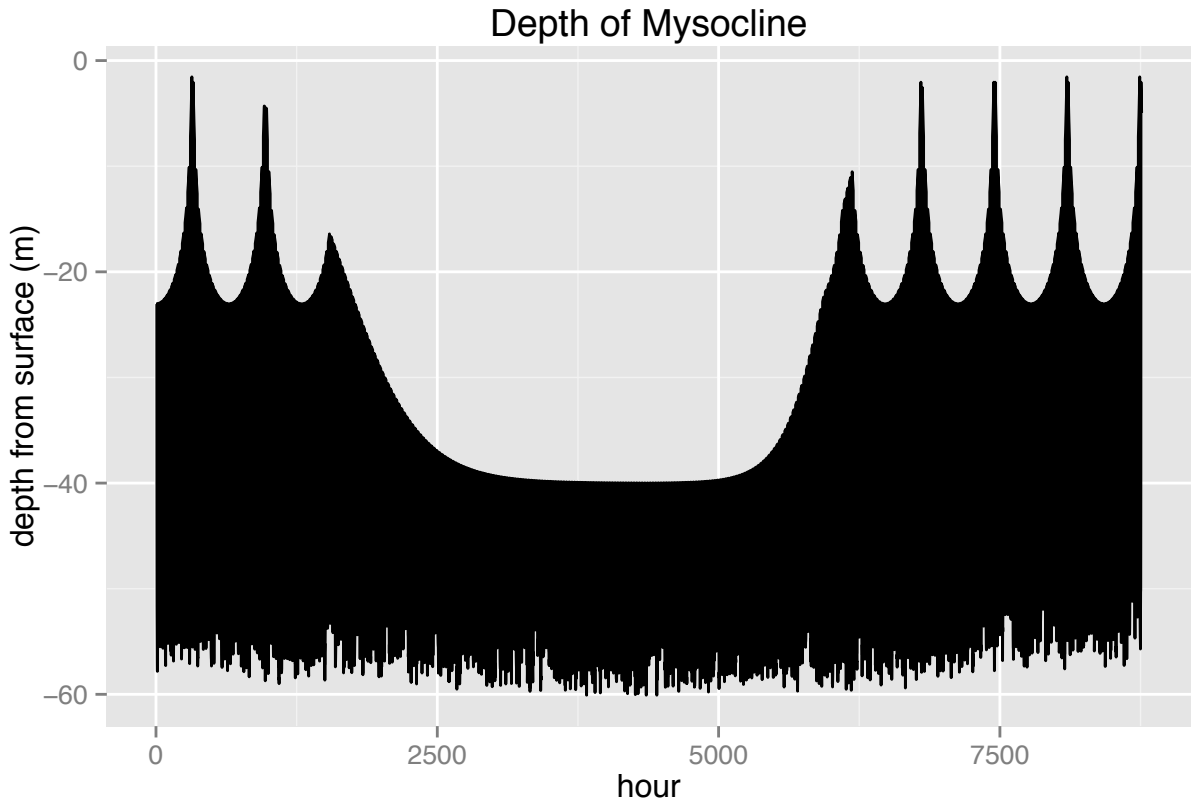


```
setwd("/Users/Nick/mysisModeling/paperMaterials/figures")
ggsave(filename = "pres_lightThreshold.pdf", width = 6, height = 2.5)
```

### Mysocline

Combining the isocline with the thermocline to get the mysocline and output to file:

```
mysocline = NULL
for (i in 1:(24*365)){
  if (depthData[i] > isocline[i]){
    mysocline = c(mysocline, depthData[i])
  } else {
    mysocline = c(mysocline, isocline[i])
  }
}

mysoclineDf = as.data.frame(cbind(hours, mysocline))
m = ggplot(mysoclineDf, aes(x = hours, y = -mysocline)) + geom_line()
m = m + labs(title = "Depth of Mysocline", x = "hour", y = "depth from surface (m)")
m
```




```
setwd("/Users/Nick/mysisModeling/paperMaterials/figures")
ggsave(filename = "pres_mysocline.pdf", width = 6, height = 2.5)

#write.csv(mysocline, "data/mysocline_hour.csv", row.names=FALSE)
```