Malia Carpio, Ali Castle, Alejandro Serrat

10/29/09

Download newtonFor.R from the CSC 210 website (under Assignment 9)

```
#newton iteration for finding sqrt(2)
#parameters x = starting value
#            final = number of iterations
#usage  newtonFor(3.1, 8)

#to get more digits use options(digits=15)

newtonFor = function(x, final){  #begin function
    print(x)
    for ( i in 1:final){          #begin for index i will run sequence
1 to final
      x = 0.5*(x + 2/x)           #replace old x with new value
      print (x)
    }                             #end for
}                                 #end function
```

`newtonFor = function(x, final){` You can give parameters to the function (highlighted in newtonFor example). The order is important.

The body of the function is between braces {} You can make the function do all kinds of tasks.

In the case of newtonFor.R we are making R do Newton's algorithm for a certain number of iterations.

print(x) means R will print out what you gave it

for() allows you to do loops; whatever is inside the parentheses tells it how to skip around.

The colon is a construct for R. For example, type 1:10 into the R console

```
> 1:10
 [1]  1  2  3  4  5  6  7  8  9 10
```
<-this sequence is the output

You can also save these sequences:

```
> a=1:10
> a
 [1]  1  2  3  4  5  6  7  8  9 10
> b=seq(1,10,by=2)
```

Malia Carpio, Ali Castle, Alejandro Serrat

> b

[1] 1 3 5 7 9

```
for ( i in 1:final){
```

i is not used; it just tells the for() function how many times to run

the second print(x) tells R to print out the new value for x

then it runs back through the for() function

Run it -> File -> Source File -> newtonFor.R

> source("/Users/maliacarpio/Documents/College Documents/Third Semester/CSC 210/R/newtonFor.R")

>

> newtonFor(3.1,8) <- 3.1 is initial condition, 8 iterations

[1] 3.1

[1] 1.872581

[1] 1.470313

[1] 1.415284

[1] 1.414214

[1] 1.414214

[1] 1.414214

[1] 1.414214

[1] 1.414214

now add print(i) to the source file:

```
newtonFor = function(x, final){  #begin function
    print(x)
    for ( i in 1:final){         #begin for index i will run sequence
1 to final
      x = 0.5*(x + 2/x)          #replace old x with new value
      print (x)
      print (i)
    }                            #end for
}                                #end function
```

Malia Carpio, Ali Castle, Alejandro Serrat

Now you have to save the source file and source it again

```
> source("/Users/maliacarpio/Documents/College Documents/Third Semester/CSC 210/R/newtonFor.R")

> newtonFor(3.1,8)
```

[1] 3.1 <- this is initial condition so i=0

[1] 1.872581

[1] 1

[1] 1.470313

[1] 2

[1] 1.415284

[1] 3

[1] 1.414214

[1] 4

[1] 1.414214

[1] 5

[1] 1.414214

[1] 6

[1] 1.414214

[1] 7

[1] 1.414214

[1] 8 <- computed it 8 times

Ok, so we don't need it anymore -> make it into a comment by adding a # in front of it.

```
newtonFor = function(x, final){  #begin function
    print(x)
    for ( i in 1:final){          #begin for index i will run sequence 1 to final
      x = 0.5*(x + 2/x)           #replace old x with new value
      print (x)
#     print (i)
    }                             #end for
```

Malia Carpio, Ali Castle, Alejandro Serrat

```
}                                 #end function
```

Now open newtonWhile.R

```
#newton iteration for finding sqrt(2)
#parameters x = starting value
#           error = error tolerance in the approximation
#usage  newtonWhile(3.1, 0.00001)

#to get more digits use options(digits=15)


newtonWhile = function(x, error){   #function body start
                                    #do the first iteration, compute
error
   x1 = 0.5*(x + 2/x)
   delta = abs( x1 - x )
   cat( "first delta = " , delta , "\n")

   x = x1

   while (delta > error){           #while starts
      oldx = x
      x = 0.5*(x + 2/x)
      delta = abs( x - oldx )
   }                              #while ends

   print(x)
   cat( "last delta = " , delta , "\n")
}                                  #funtion body ends
```

Source the file.

You want to stop iterating the algorithm once the digits settle down -> ex. stop running it when I get 6 digits for the square root of 2.

Don't know how many iterates to do; have it stop when it meets a certain error tolerance in the approximation. (in a parameter shown below)

```
newtonWhile = function(x, error){
```

```
> source("/Users/maliacarpio/Documents/College Documents/Third Semester/CSC 210/R/newtonWhile.R")

> newtonWhile(3.1,0.000001)
```

Malia Carpio, Ali Castle, Alejandro Serrat

first delta = 1.227419

[1] 1.414214

last delta = 4.046432e-07

Print both old value and new value so we can compare them (assign a new variable)

while() loops checks the condition inside the parentheses

while(delta>error)

If it is true, while runs through. If it is false, while stops its loop.

Open curveFit.R and source it

```
#fit a line through 2 data points

x = c(1, 2)
y = c(1.5, 2.1)

plot(x, y, main="2 points")
fit = lm(y~x)         #compute the line and dave data in fit
abline(fit)           #plot the line stored in fit
print(fit)
print(summary(fit)) #lots more info

#compute least-square line fit for 3 data points
x = c(1, 2, 3)
y = c(1.5, 1.1, 2.1)

plot(x, y, main="3 points")
fit = lm(y~x)
abline(fit)
print(fit)
print(summary(fit)) #lots more info about the fitness

#compute the parabola through 3 data points
x = c(1, 2, 3)
y = c(1.5, 1.1, 2.1)

plot(x, y, main="parabola through 3 points")
fit = lm(y~x + I(x^2))
print(fit)
curve(3.3 - 2.5* x + 0.7*x^2, 0, 4, add=TRUE)
```

Need to determine a function from data points.

Malia Carpio, Ali Castle, Alejandro Serrat

If you have more than two data points, it can be difficult to fit a curve to the points. Most scientists draw a line of best fit through the middle of the points.

The distance from each point to the line is your "error." If a different line is picked, the error will change. We want to pick a line that will minimize the sum of these distances -> "least-square" fit.

There is a unique parabola that passes through 3 points that are not colinear (a straight line).

Equation of a parabola: $p(t) = a + bt + ct^2$

We need to determine the three unknown parameters from the data points to find the form of the function that will fit through the points. The computer will do this for us.

A cubic functions will fit through 4 points (usually 1 less degree than the number of points).

```
x = c(1, 2)
y = c(1.5, 2.1)

plot(x, y, main="2 points")
fit = lm(y~x)        #compute the line and dave data in fit
abline(fit)          #plot the line stored in fit
print(fit)
print(summary(fit)) #lots more info
```

Create two vectors x and y for the data points

plot() makes a graph of the 2 points

lm(formula) stands for linear model and you give the formula inside the parentheses

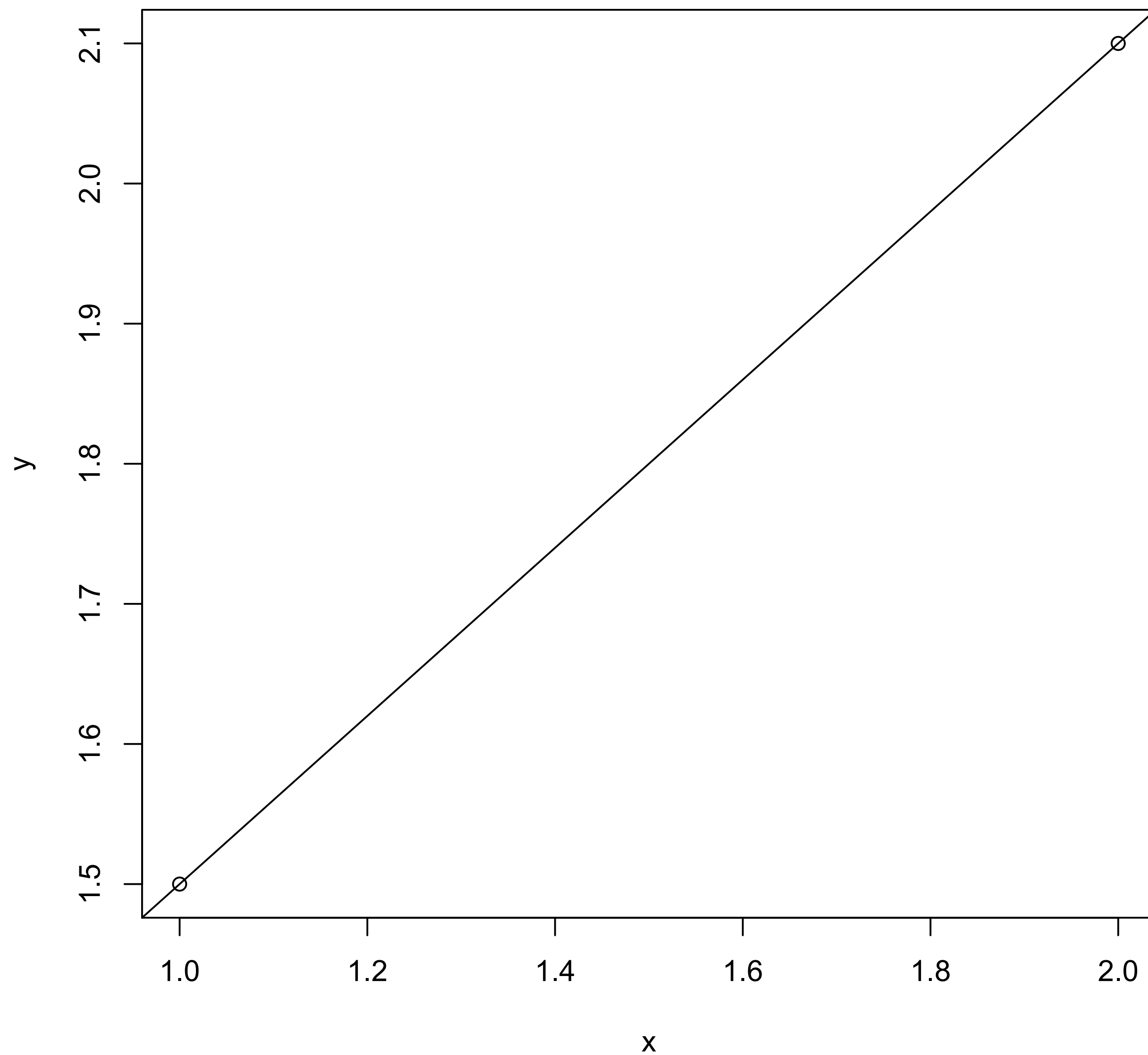
If you want to determine the linear function as a function of x, you use y~x

It does all the error distances and gives you the slope (b) and the intercept (a) of the least-square fit line. (If the formula for the line = a + bt)

Source the file

Use the back/forward commands to see the first graph (check notes from 10/27/09 to see how to use these commands).

Malia Carpio, Ali Castle, Alejandro Serrat




```
> source("/Users/maliacarpio/Documents/College Documents/Third Semester/CSC 210/R/curveFit.R")
```

Call:

lm(formula = y ~ x)

Coefficients:

(Intercept)   x   <- gives slope and intercept

0.9   0.6

Malia Carpio, Ali Castle, Alejandro Serrat

Call:

lm(formula = y ~ x)

Residuals:

ALL 2 residuals are 0: no residual degrees of freedom!

Coefficients:

	Estimate	Std. Error	t value	Pr(>|t|)
(Intercept)	0.9	NA	NA	NA
x	0.6	NA	NA	NA

Residual standard error: NaN on 0 degrees of freedom

Multiple R-squared: 1, Adjusted R-squared: NaN

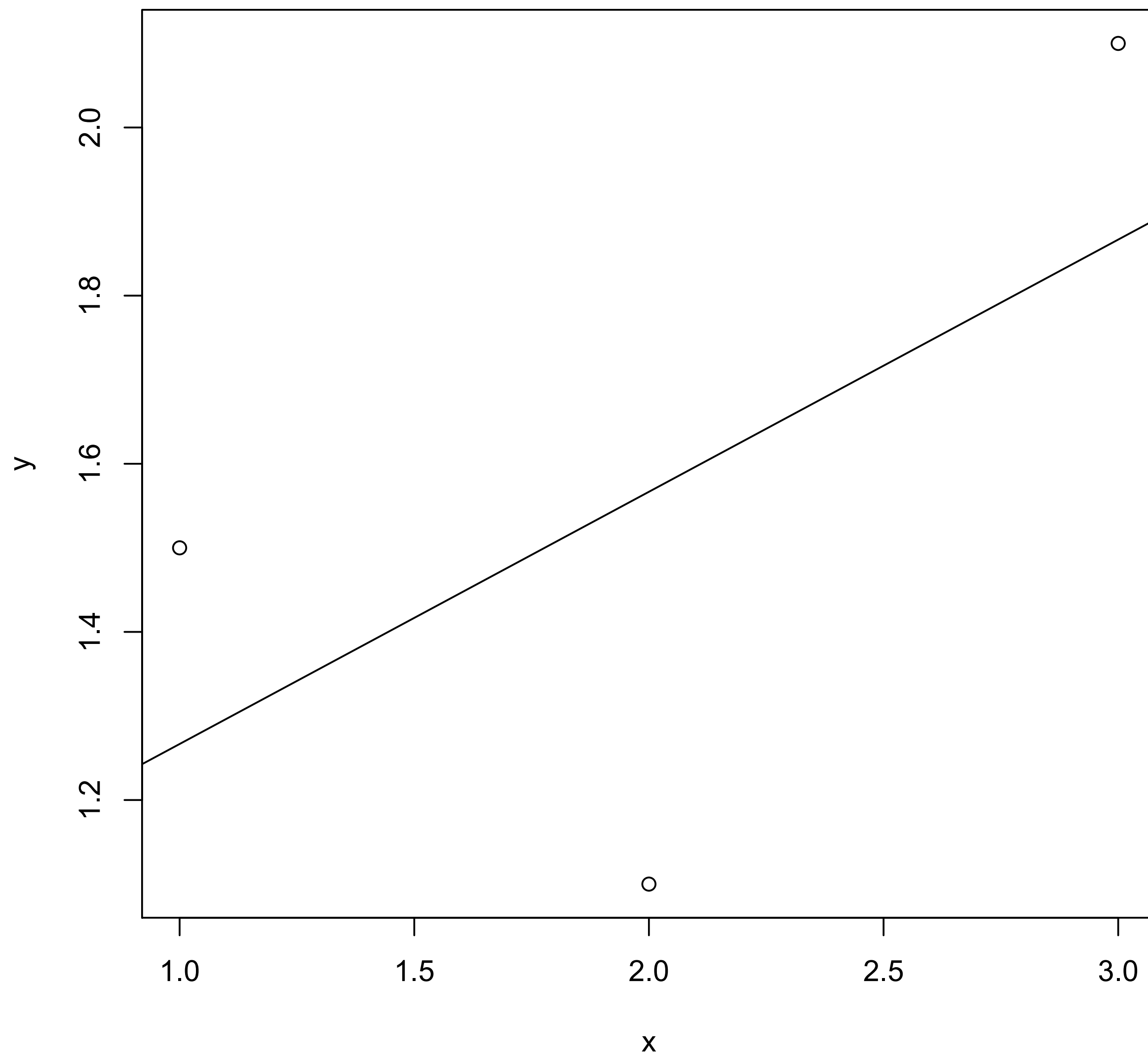
F-statistic: NaN on 1 and 0 DF, p-value: NA

Now we want to fit a line through 3 points.

```
plot(x, y, main="3 points")
fit = lm(y~x)
abline(fit)
print(fit)
print(summary(fit)) #lots more info about the fitness
```

Go forward in the graph window to see the graph for the 3 point line:

Malia Carpio, Ali Castle, Alejandro Serrat



Check the R console->

Call:

lm(formula = y ~ x)

Coefficients:

(Intercept) x

0.9667 0.3000

Call:

Malia Carpio, Ali Castle, Alejandro Serrat

lm(formula = y ~ x)

Residuals: <-gives distances from the line

1 2 3

0.2333 -0.4667 0.2333

Coefficients:

	Estimate	Std. Error	t value	Pr(>|t|)
(Intercept)	0.9667	0.8731	1.107	0.468
x	0.3000	0.4041	0.742	0.593

Residual standard error: 0.5715 on 1 degrees of freedom

Multiple R-squared: 0.3553,   Adjusted R-squared: -0.2895

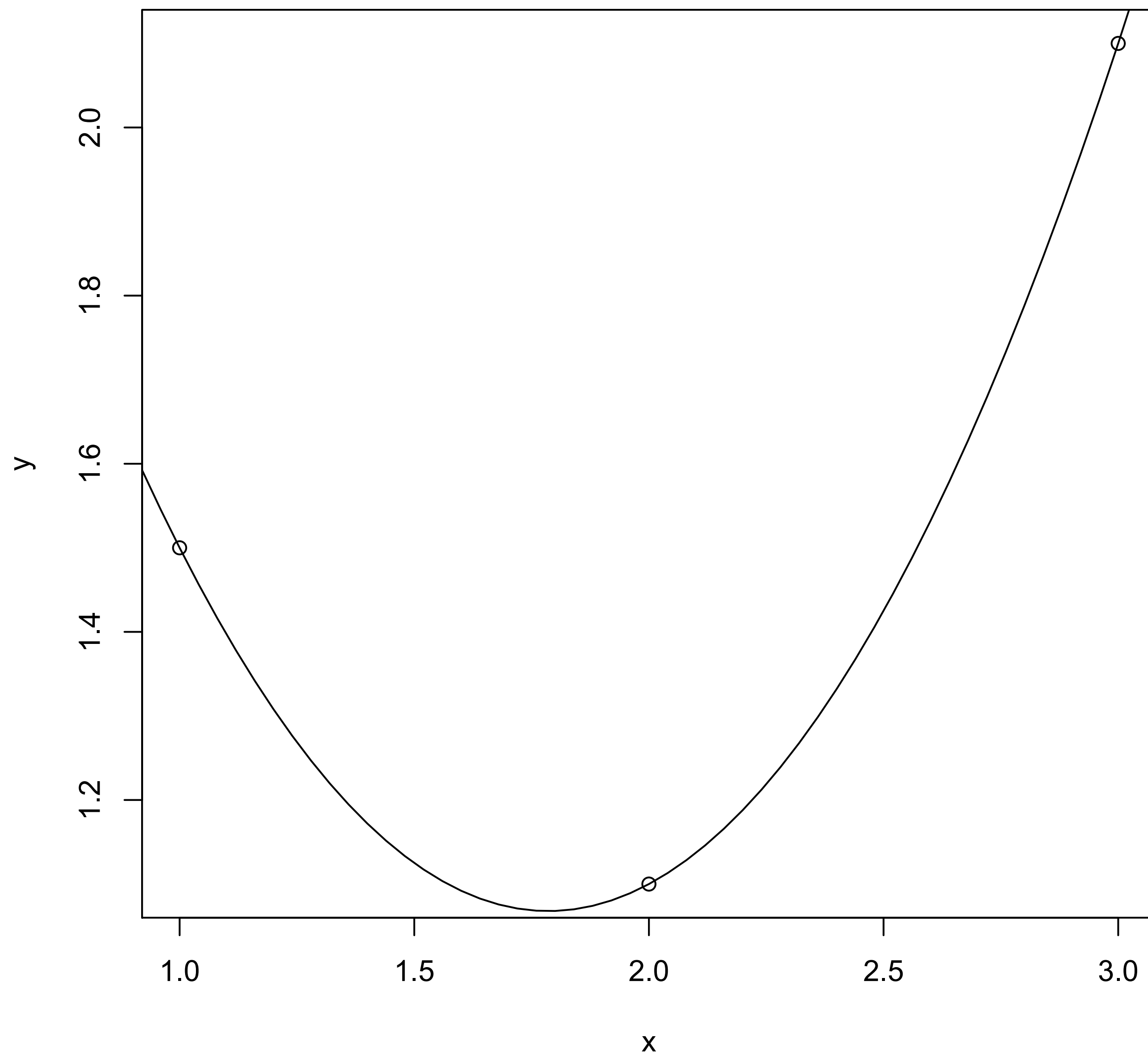
F-statistic: 0.551 on 1 and 1 DF, p-value: 0.5935

Now we want to find a parabola through the three points.

One way is to guess, another way is to use an algorithm to find the coefficients, or we can cheat and use lm().

Go forward in the graph window to see the parabola:

Malia Carpio, Ali Castle, Alejandro Serrat




```
#compute the parabola through 3 data points
x = c(1, 2, 3)
y = c(1.5, 1.1, 2.1)

plot(x, y, main="parabola through 3 points")
fit = lm(y~x + I(x^2))
print(fit)
curve(3.3 - 2.5* x + 0.7*x^2, 0, 4, add=TRUE)
```

Run the linear model, but not just as a function of x, but as a function of x2. I means Identity. `fit = lm(y~x + I(x^2))` <-this means you are looking for y as a function of x

Malia Carpio, Ali Castle, Alejandro Serrat

+ x2.

`fit = lm(y~x + I(x^2))` <- Asking lm() to fit() this quadratic function through the three points

print(fit) spits out three coefficients (see R console):

Call:

lm(formula = y ~ x + I(x^2))

Coefficients:

(Intercept)	x	I(x^2)
3.3	-2.5	0.7

The computer is saying that the parabola is equal to p(t) = 3.3 - 2.5x + $0.7x^2$

Checked parameters for parabola and plotting using curve() function

```
curve(3.3 - 2.5* x + 0.7*x^2, 0, 4, add=TRUE)
```

`add=TRUE` means you're adding this new curve to the graph with the points

Example of parameter estimation of data:

Look at the Logistic differential equation <- modeled population growth

$$\frac{dx}{dt} = rx(1 - \frac{x}{k})$$

where x(t) is population size at time t, r is growth rate, and K is carrying capacity

The r and K from wild populations can be very complicated, so test the model in ideal conditions from populations in the laboratory (ex. grow beetles or bacteria in idealized conditions and count them every day).

Lab data for flour beetle:

Open beetle.R

```
#flour beetle data

#daily count of beetles for 16 days
y = c(112, 152, 212, 258, 306, 309, 315, 310,
298, 290, 303, 295, 311, 308, 299, 309)
print(y)

#daily count of beetles for 15 days
```

Malia Carpio, Ali Castle, Alejandro Serrat

```
y1 = c(112, 152, 212, 258, 306, 309, 315, 310,
298, 290, 303, 295, 311, 308, 299)

#time in days
x = seq(0, length(y)-1)
print(x)

plot(x, y, main="Daily beetle counts")

#create a rate vector and load it with computed rates
rate = 0
#print (rate)

for (i in 1:length(y)-1){
rate[i] = (y[i+1] - y[i])/y[i]
}
print(rate)

#linear least squares fit
#drop last y entry

plot(y1, rate, col="red", main="Least square fit to rates")
fit = lm(rate~y1)
abline(fit)
print(fit)
print(summary(fit)) #lots more info
```

This is all discrete data, so we have to discretize this problem.

Recall derivative:

$$\frac{dx}{dt} = \lim_{h->0} \frac{x(t+h) - x(t)}{h}$$

take h=1 day

$$\frac{dx}{dt} \approx x(t+1) - x(t)$$

$$x(t+1) - x(t) = rx(t)(1 - \frac{x(t)}{k})$$

$$rewrite:$$

$$\frac{x(t+1) - x(t)}{x(t)} = r(1 - \frac{x(t)}{k})$$

The left side is computable. The right side is a linear function. More next lecture....