Question 1: Optimizing parameters

For this question we have to approximate f(x) by means of parabolic interpolation, that is to say, optimizing the "a" parameters of f tilde(x) = $a_o x+a_1 x+a_2 x^2$, a piecewise parabolic function.

1st point

For the 1st point, we created 3 functions aiming at fitting our linear model through the minimization of the Sum Squared Error.

```
interpolator <- function(a, x){
  X <- as.matrix(c(1, x, x^2), ncol = 1 )
  return(as.vector(a %*% X))
}

SSE <- function(x, a, method1, method2){
  real <- sapply(x, method1)
  pred <- sapply(x, method2, a = a)
  return(sum((real-pred)^2))
}

optimiser <- function(x, method){
  aInit <- c(0, 0, 0)
  res <- optim(aInit, SSE, x = x, method1 = method, method2 = interpolator)
  return(res$par)
}
```

As you can see in the functions, our interpolator It's just the function f tilde(x) = $a_o x+a_1 x+a_2 x^2$, that uses a matrix multiplication between the input data matrix X by our parameters a_i.

The SSE function just sums the squares of the errors between the predictions of our interpolator and the actual values of the target function.

The optimiser functions initialises the a_i values to a vector of 0s, then uses optim function to fit these parameters minimizing the SSE and then returning an array that contains (a_1, a_2, a_3).

Note

Optim function minimization method: implementation of that of **Nelder and Mead algorithm**. that uses only function values and is robust but relatively slow. It works reasonably well for non-differentiable functions.

2nd point

For the second point we are asked to implement a function that approximates a function defined on the interval [0, 1]. It has to approximate the target function in n equal-sized subintervals. So for each subinterval, we will have 3 points to interpolate, (both ends and the medium value).

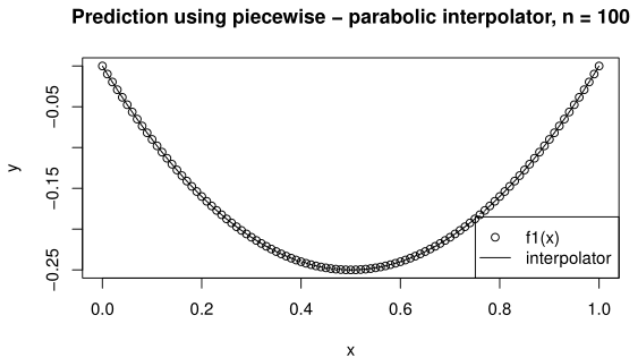
```
aproximate <- function(n, method){
  scale <- 1/n
  midVal <- scale / 2
  result <- list()
  for (i in 1:n) {
    x <- c((scale*i)-scale, (scale * i) - midVal, scale * i)
    result <- append(result, list(optimiser(x, method)))
  }
  return(result)
}
```

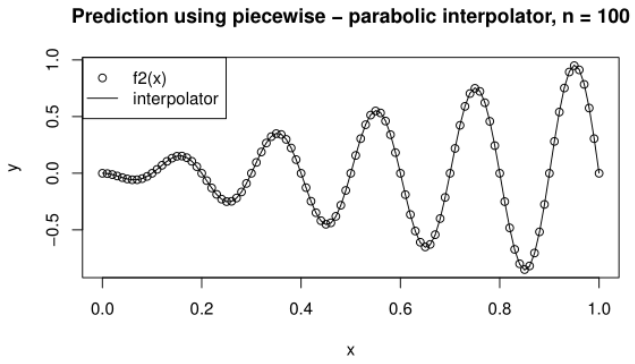
As we can see in the code, with a for loop we append to a list 3 parameters a_i fitted in every iteration given a subinterval [scale times i - scale, scale times i] with the medium value defined as scale times I – midVal (the half of our scale).

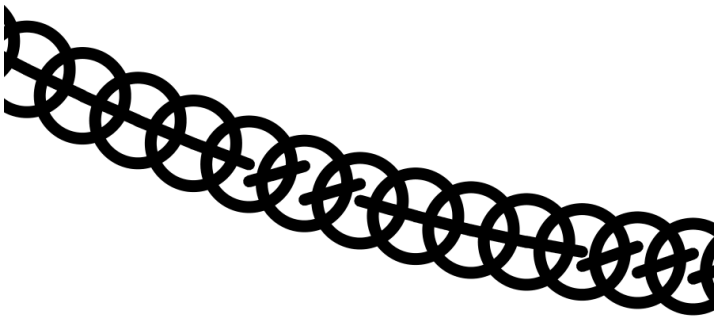
As we can see in the following plots, the plotted circles are the target functions whereas the straight lines correspond to our fitted polynomial approximations for each 3 points given every interval. If we zoom-in we can see our approximation is not completely accurate sometimes.

First plot target function: x + x^2

Second plot target function: -x * sin(10 * pi * x)







Question 2: Maximizing likelihood

First of all, we are given a sample normally distributed with some parameters sigma and mu.

Then, we obtain the Likelihood function, the probability of having these parameters given the data observed. Then we apply logarithms because the resulting function is easier to derive (we avoid the differentiation of exponential terms) and the maxima of Likelihood and log-likelihood are the same.

The sampled data is normally distributed with μ and σ

$$y \sim N(\mu, \sigma^2)$$

Likelihood of the model:

$$L(p(\mu, \sigma^2|y)) = \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(y-\mu)^2}{2\sigma^2}} = \frac{1}{(\sqrt{2\pi\sigma^2})^n} e^{-\sum_{i=1}^{n} \frac{(y_i-\mu)^2}{2\sigma^2}}$$

log - likelihood of the model:

$$lnL(p(\mu, \sigma|y)) = -\frac{n}{2} ln(2\pi\sigma^2) - \sum_{i=1}^{n} \frac{(y_i-\mu)^2}{2\sigma^2}$$

As we can see in the next equations, we obtain the Maximum Likelihood Estimates of mu and sigma obtaining partial derivates and equating to 0 in each case. Thus we obtain the values of mean and sigma in the table.

MLE estimators are partial derivatives of - log-likelihood

$\hat{\mu}_{MLE}$ estimator:

$$\frac{\partial lnL(p(\mu,\sigma|y))}{\partial \mu} = -\frac{1}{2\sigma^2}\frac{\partial(\sum y_i^2 - 2\mu\sum y_i + n\mu^2)}{\partial \mu} = -\frac{1}{2\sigma^2}(0 - 2\sum y_i + 2n\mu) = \frac{\sum y_i - n\mu}{\sigma^2}$$

$$MLE => \frac{\sum y_i - n\mu}{\sigma^2} = 0$$

$$\hat{\mu}_{MLE} = \frac{\sum y_i}{n}$$

$\hat{\sigma}_{MLE}$ estimator:

$$\frac{\partial lnL(p(\mu,\sigma|y))}{\partial \sigma} = -\frac{n}{\sigma} + \frac{1}{\sigma^3}\sum_{i=1}^{n}(y_i - \mu)^2$$

$$MLE => -\frac{n}{\sigma} + \frac{1}{\sigma^3}\sum_{i=1}^{n}(y_i - \mu)^2 = 0$$

$$\hat{\sigma}^2_{MLE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \mu)^2$$

mean	variance
1.276	2.006

For the third point in this activity, we are to optimize the minus-log-likelihood function with initial parameters mu= 0, sigma= 1. The minus sign assures that we are finding the minima of the function by means of the minimization with optim function.

```
minusLogLikelihood <- function(x){
  n <- length(data)
  data <- data
  mu <- x[1]
  sigma <- x[2]
  part1 <-  ((n/2) * log(2*pi * (sigma^2)))
  part2 <- (sum((data - mu)^2)) / (2 * sigma^2)
  return(part1 + part2)
}
```

The natural logarithm is a monotonically increasing function. If one value of x increases the value of y related to it will increase too. It will eventually ensure max-log-likelihood occurs at the same point where max-likelihood function occurs. It is also easier and simpler to work with logarithms instead of original likelihood. Taking derivative from a logarithm is simpler than with the original likelihood which are often exponential functions.

In the next piece of code, we define our gradients of the minus-log-likelihood function to be introduced in the optim functions.

```r
gradient <- function(x){
  mu <- x[1]
  sigma <- x[2]
  n <- length(data)
  gMu <- sum(mu - data)/(sigma^2)
  gSigma <- (n/sigma) - ((1/(sigma^3)) * sum((data-mu)^2))
  return(c(gMu,gSigma))
}
```

In the next piece of code, we see how we applied optim function to minimize the minus-log-likelihood function. We conduct these minimization experiments varying the **gr** parameter. With the gr parameter default, this function calculates the gradients by finite-difference approximation, thus calculating the direction of steepest decreasing.

We will also vary the algorithm of gradient minima finding: BFGS (Broyden–Fletcher–Goldfarb–Shannoand, a Quasi-Newton) and CG (conjugate gradient method).

```r
init <- c(0, 1)
test1 <- optim(c(0, 1), minusLogLikelihood,  method = "CG")
test2 <- optim(c(0, 1), minusLogLikelihood, gr = gradient,  method = "CG")
test3 <- optim(c(0, 1), minusLogLikelihood, method = "BFGS")
test4 <- optim(c(0, 1), minusLogLikelihood, gr = gradient, method = "BFGS")
```

At the end we will obtain these results.

	converge	mean	variance	n. of functions	n. of gradients
minus loglikelihood CG	Yes	1.27552771909709	2.00597650338868	208	35
minus loglikelihood CG gradient	Yes	1.27552759112531	2.00597647249389	53	17
minus loglikelihood BFGS	Yes	1.27552755151932	2.00597696486639	41	15
minus loglikelihood BFGS gradient	Yes	1.27552755040258	2.00597654945241	39	15

We would recommend to use BFGS with specified gradient, because it required to evaluate less functions and gradients as a result of being faster.

Discussion of methods CG and BFGS

CG is more memory efficient as it doesn't store a matrix. Therefore they may be successful in much larger optimization problems.

However, BGFS uses less function calls and it converges faster. BFGS will converge in fewer steps than CG, and has a little less of a tendency to get "stuck" and require slight algorithmic tweaks in order to achieve significant descent for each iteration.

CG requires matrix-vector products, which may be useful to you if you can calculate directional derivatives (again, analytically, or using finite differences).

A finite difference calculation of a directional derivative will be much cheaper than a finite difference calculation of a Hessian (like in BGFS), so if you choose to construct your algorithm using finite differences, just calculate the directional derivative directly. This observation,

however, doesn't really apply to BFGS, which will calculate approximate Hessians using inner products of gradient information.

Our gradient calculations helped in convergence, because they allowed to reduce function calls