Module VI: Evolutionary algorithms

Pawel Chilinski

November 29, 2014

Examples in this document are implemented in R.

Exercise

Using of any program language, write evolutionary strategies for fitting of function to experimental data set. The format of function is a polynomial: $y(x) = ax^2 + bx + c$. Find parameters: a, b, c. Try to implement strategies: (1+1), $(\mu + \lambda)$, (μ, λ). Find and describe differences in efficiency. Try to find, what is a reason of differences in algorithms operating.

The fitness function in my implementation is RMSE over the empirical data. So algorithms try to minimise this fitness. For all evolutionary strategies I am using the the threshold of 10 RMSE. So algorithm stops when RMSE of the best solution is equal or lower to 10. In this case I can compare algorithms using number of generations it took them to reach assumed target.

```
> MAX_RMSE<-10
```

Create table with the empirical data:

```
> #empirical data
> data <- data.frame(
+                 x=c(-10.13,-8.83,-8.02,-7.20,-6.00,-4.99,-4.00,-3.10,-2.00,-1.00,0.00,1.21,2.00,3.00,4.10,
+                                 4.93,6.01,7.10,8.00,9.11,10.00),
+                 y=c(251.23,205.12,143.98,108.32,67.34,44.25,17.93,3.21,-4.89,-8.32,-2.98,7.89,28.54,53.23,
+                                 82.94,122.98,168.94,224.12,278.32,354.23,414.94))
```

Functions and data used by all strategies:

```
> set.seed(123)
> #compute individual fitness
> fitness <- function(individual){
+         #for each data point compute inverse of squared difference
+         a<-individual["a"]
+         b<-individual["b"]
+         c<-individual["c"]
+         sqrt(sum(apply(data, 1, function(row){
+                                                 y<-row["y"]
+                                                 x<-row["x"]
+                                                 empY <- a*x^2+b*x+c
+                                                 (y-empY)^2
+                                         }))/nrow(data))
+ }
> #computes fitness value for the entire population
> populationFitness<-function(population){
+         #computes fitness for the individual composes of a,b,c genes
+
+         #for each individual compute its fitness
+         apply(population, 1, function(individual){
+                                 fitness(individual)
+                         })
+ }
> #returns function that selects best n individuals from the population
> selectBestFun <- function(n){
+                 function(population){
+                         population[order(population$fitness, decreasing = FALSE)[1:n],]
+                 }
+ }
> #checks if the best individual has better fitness that threshold
> bestFitnessBetterThansStoppingCriterionFun <- function(theshold){
```

```
+                 function(population){
+                         population[which.min(population$fitness),"fitness"] <= theshold
+                 }
+ }
> applyGeneticOperations <- function(population, context, geneticOperations){
+         temporaryPopulation <- population
+         for(op in geneticOperations){
+                  ret <- op(temporaryPopulation, population, context)
+                  temporaryPopulation <- ret$temporaryPopulation
+                  population <- ret$population
+         }
+         list(temporaryPopulation=temporaryPopulation,population=population)
+ }
> #Run evolution strategy
> #initialPopulation - initial population
> #createTemporaryPopulationFun - creates temporary population
> #geneticOperations - list of genetic operations to be performed on each population
> #afterGeneticOperationsProcessor - function that receives 2 populations: before genetic operations and
> #after genetic operations and returns population
> #for which fitenss of each individual is computed and
> #then selectBestFun is called
> #computeContextAfterGenerationFun - computes context after
> #selectBestFun - function the select best individuals from population
> #stoppingCriterionFun - function the assesses when we should stop running evolution strategy
> evolutionStrategy <- function(initialPopulation,createTemporaryPopulationFun,geneticOperations,
+                 afterGeneticOperationsProcessor,
+                 computeContextAfterGenerationFun=function(previousPop,newPopulation){NULL},selectBestFun,
+                 stoppingCriterionFun){
+         offspringPopulation <- initialPopulation
+         offspringPopulation$fitness <- populationFitness(offspringPopulation)
+         context<-computeContextAfterGenerationFun(offspringPopulation, offspringPopulation)
+         generation <- 1
+         offspringPopulation$generation <- generation
+         bestIdx<-order(offspringPopulation$fitness,decreasing = F)[1]
+         history<-data.frame(generation=generation,fitness=offspringPopulation$fitness[bestIdx],
+                         a=offspringPopulation$a[bestIdx],
+                         b=offspringPopulation$b[bestIdx],
+                         c=offspringPopulation$c[bestIdx])
+         while(!stoppingCriterionFun(offspringPopulation)){
+                 temporaryPopulation<-createTemporaryPopulationFun(offspringPopulation)
+                 ret<-applyGeneticOperations(temporaryPopulation,context,geneticOperations)
+                 temporaryPopulation <- afterGeneticOperationsProcessor(ret$temporaryPopulation,
+                                 ret$population)
+                 temporaryPopulation$fitness <- populationFitness(temporaryPopulation)
+                 bestPopulation <- selectBestFun(temporaryPopulation)
+                 row.names(bestPopulation) <- NULL
+                 generation <- generation + 1
+                 bestPopulation$generation <- generation
+
+                 context<-computeContextAfterGenerationFun(offspringPopulation, bestPopulation)
+
+                 offspringPopulation<-bestPopulation
+
+                 bestIdx<-order(offspringPopulation$fitness,decreasing = F)[1]
+                 history<-rbind(history,c(generation,offspringPopulation$fitness[bestIdx],
+                                                 a=offspringPopulation$a[bestIdx],
+                                                 b=offspringPopulation$b[bestIdx],
+                                                 c=offspringPopulation$c[bestIdx]))
+         }
+         #return the last generation
+         list(solution=offspringPopulation,history=history)
+ }
> #Generate initial population
> generateInitialPopulation<-function(size,minA,maxA,minB,maxB,minC,maxC) {
+         #population is a data frame (table) which is composed of individuals (as row) and its attributes
+         #(columns) like genotype chromosme (a,b,c),
```

```
+         #deviation chromosome (aDev,bDev,cDev) and fitness value
+         #mutationSuccessRate in the last k generations
+         data.frame(
+                         a=runif(size,minA,maxA),
+                         b=runif(size,minB,maxB),
+                         c=runif(size,minC,maxC),
+                         fitness=NA)
+ }
> #Plots evolutionary strategy fitted curve for different generations
> plotFittedCurveHistory<-function(evolutionStrategyRes,nPlots=min(20,nrow(evolutionStrategyRes$history))){
+         step<-nrow(evolutionStrategyRes$history)/nPlots
+         plotList <- vector('list', nPlots)
+         gen<-1
+         for(i in 1:nPlots){
+
+                 #so we always show the last generation
+                 if(i==nPlots){
+                         gen<-nrow(evolutionStrategyRes$history)
+                 }
+
+                 #create function the will show fitted polynomial
+                 createTheoreticalFunction<-function(){
+                         a<-evolutionStrategyRes$history[ceiling(gen),"a"]
+                         b<-evolutionStrategyRes$history[ceiling(gen),"b"]
+                         c<-evolutionStrategyRes$history[ceiling(gen),"c"]
+                         function(x){
+                                 a*x^2+b*x+c
+                         }
+                 }
+
+                 #constuct plot with empirical points and fitted polynomial
+                 plotList[[i]] <- ggplot(data=data) + geom_point(aes(x,y)) +
+                                 stat_function(fun = createTheoreticalFunction(),colour="blue") +
+                                 theme(text = element_text(size=7),axis.text=element_text(size=5),
+                                                 axis.title=element_text(size=8)) +
+                                 ggtitle(paste("Fitted function in generation:",ceiling(gen)))
+                 gen <- gen + step
+         }
+         #plot all graphs
+         do.call("grid.arrange", c(plotList, nrow = 10, ncol = 2))
+ }
```

(1+1) evolution strategy

Functions used by (1+1) evolution strategy:

```
> #return function performing mutation for the (1+1) evolution strategy
> #initDev - initial deviation
> #successK - how many previous generations should be considered to compute succcess rate
> mutationFunOnePlusOne <- function(initDev,successK){
+         function(temporaryPopulation,population,context){
+                 mutatedPopulation <- temporaryPopulation
+                 if(! "dev" %in% colnames(population)){
+                         population$dev <- initDev
+                         mutatedPopulation$dev <- initDev
+                 }
+
+                 if(! "successRate" %in% colnames(population)){
+                         population$successRate <- 0
+                         mutatedPopulation$successRate <- 0
+                 }
+
+                 for(i in 1:nrow(population)){
+                         mutatedPopulation[i,c("a","b","c")]<-population[i,c("a","b","c")] +
+                                         population[i,rep("dev",3)] * rnorm(3)
+                         mutatedPopulation$fitness <- fitness(mutatedPopulation[i,c("a","b","c")])
+
```

```
+                         #adjusting success rate
+                         diff <- population[i,"fitness"] - mutatedPopulation[i,"fitness"]
+                         if(diff > 0) {
+                                 successRate <- min(1, population[i,"successRate"] + 1/successK)
+                         } else {
+                                 successRate <- max(0,population[i,"successRate"] - 1/successK)
+                         }
+                         population[i,"successRate"]        <- successRate
+                         mutatedPopulation[i,"successRate"]        <- successRate
+
+                         #adjust deviation based on success rate
+                         sucessRate <- population[i,"successRate"]
+
+                         if(sucessRate < 0.2){
+                                 newDev <- 0.82 * population[i,"dev"]
+                         }else if (sucessRate > 0.2){
+                                 newDev <- 1.2 * population[i,"dev"]
+                         }else{
+                                 newDev <- population[i,"dev"]
+                         }
+                         population[i,"dev"] <- newDev
+                         mutatedPopulation[i,"dev"] <- newDev
+                 }
+                 list(population=population,temporaryPopulation=mutatedPopulation)
+         }
+ }
```

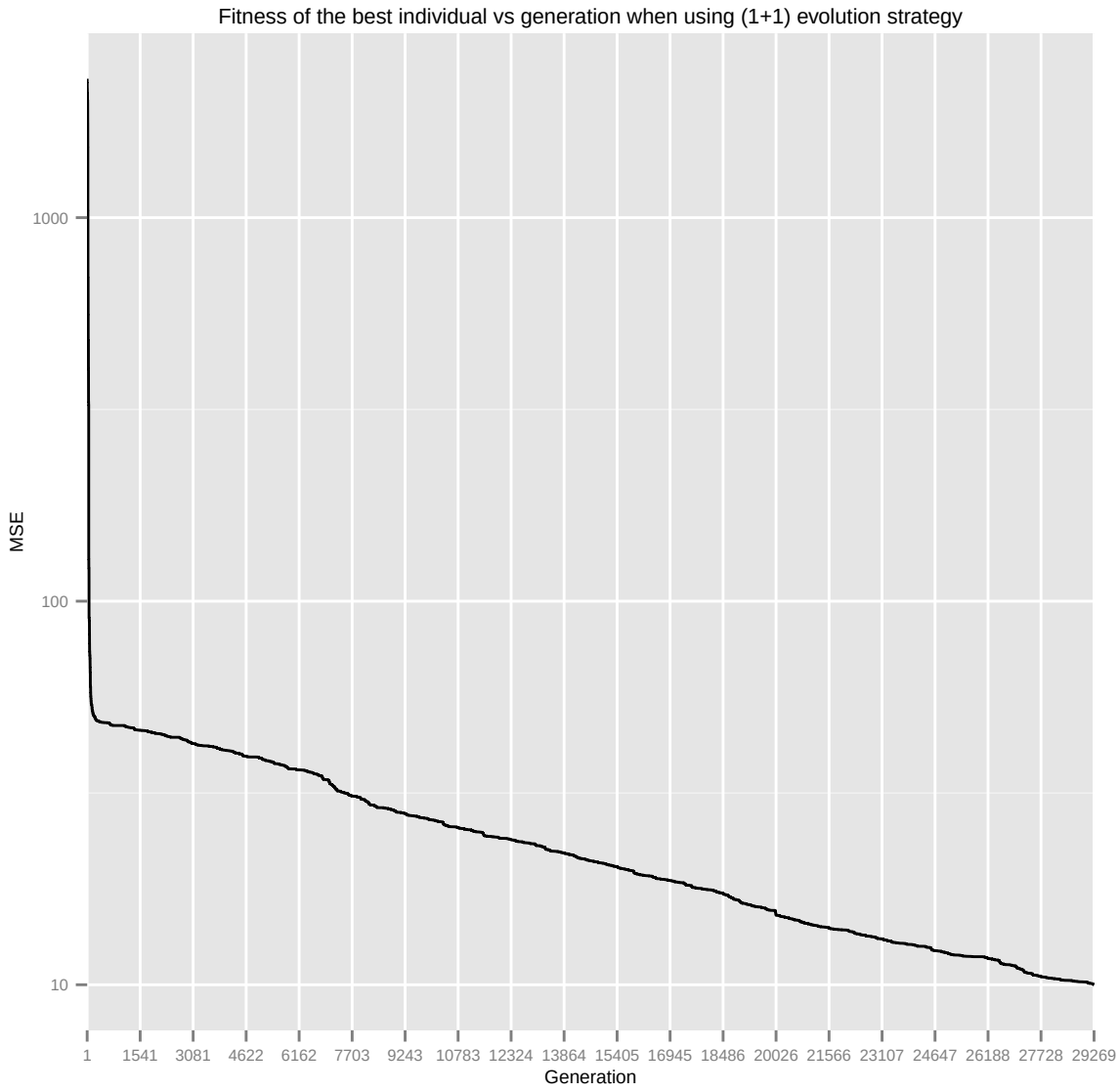
Running (1+1) evolution strategy:

```
> onePlusOneRes <- evolutionStrategy(
+                 #(1+1) evolution strategy, simply start with population with one individual
+                 initialPopulation = generateInitialPopulation(size=1,minA = -100, maxA = 100, minB = -100,
+                         maxB = 100, minC = -100, maxC = 100),
+                 #in case of (1+1) evolution strategy we simply return current population
+                 createTemporaryPopulationFun = function(population){population},
+                 #for (1+1) using only mutation
+                 geneticOperations = c(mutationFunOnePlusOne(initDev = 1, successK = 5)),
+                 #we combine individual after mutation with original individual so we can select the best in
+                 #the next step
+                 afterGeneticOperationsProcessor = function(temporaryPopulation,offspringPopulation) {
+                         rbind(temporaryPopulation, offspringPopulation)
+                 },
+                 #always select best from the pool of the mutated and old individual
+                 selectBestFun = selectBestFun(1),
+                 #stop when the individual has fintess better than
+                 stoppingCriterionFun = bestFitnessBetterThansStoppingCriterionFun(MAX_RMSE))
```

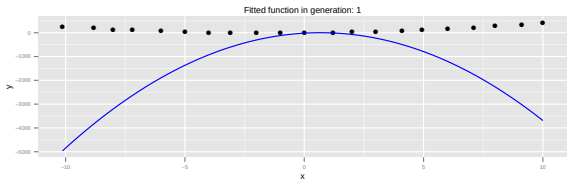
Parameters of the best solution are:

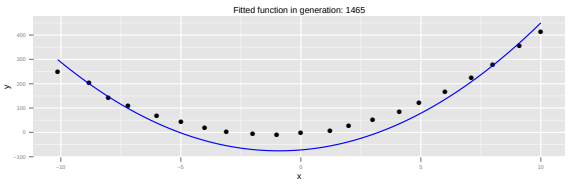
```
> onePlusOneRes$solution

         a        b         c  fitness generation        dev successRate
1 3.561066 8.251254 -16.05652 9.986798      29269 0.02682145         0.2
```

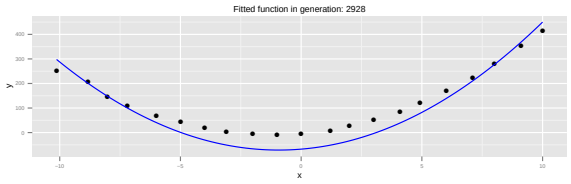
Show history of fitness (MSE) of individuals through generations when looking for the best solution using (1+1) evolution strategy:

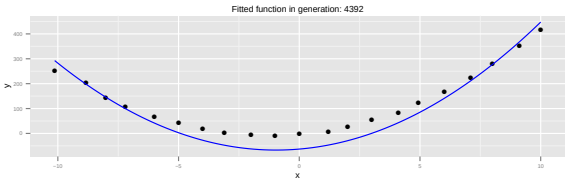


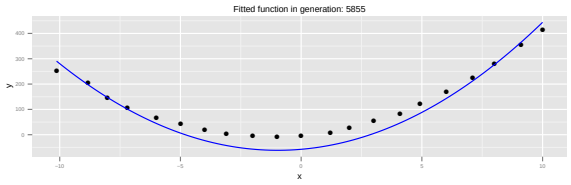
Show different functions that were found during finding optimal solution:

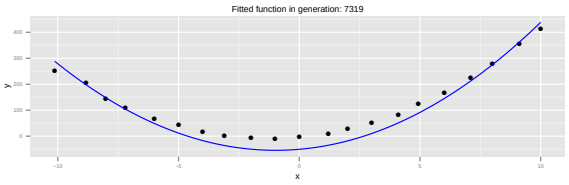

Fitted function in generation: 1
x
y

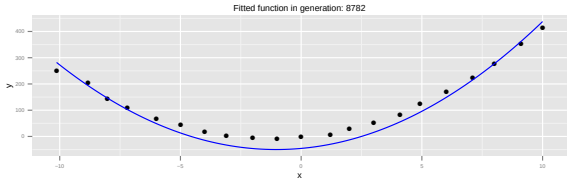

Fitted function in generation: 1465
x
y

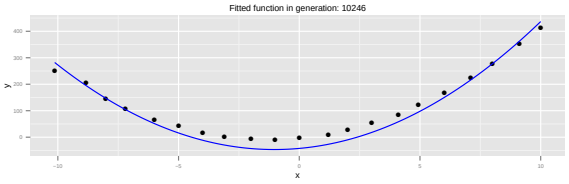

Fitted function in generation: 2928
x
y

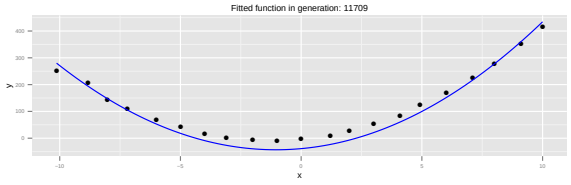

Fitted function in generation: 4392
x
y

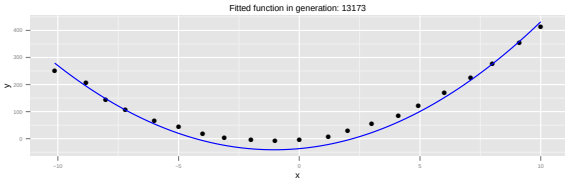

Fitted function in generation: 5855
x
y

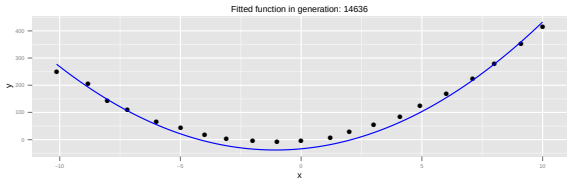

Fitted function in generation: 7319
x
y

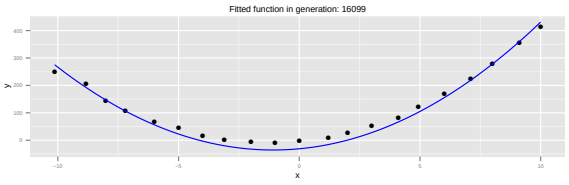

Fitted function in generation: 8782
x
y

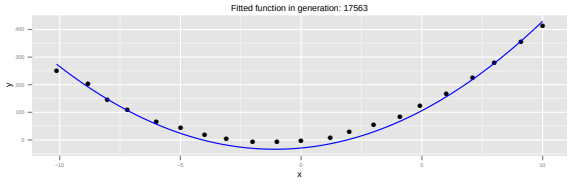

Fitted function in generation: 10246
x
y

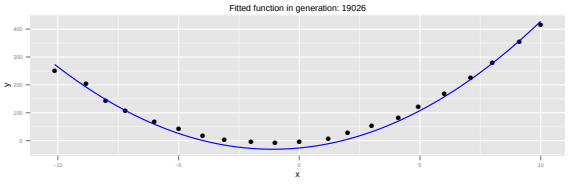

Fitted function in generation: 11709
x
y

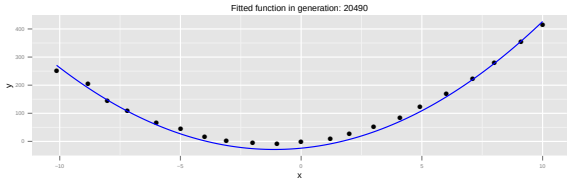

Fitted function in generation: 13173
x
y

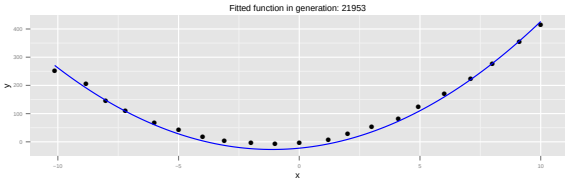

Fitted function in generation: 14636
x
y

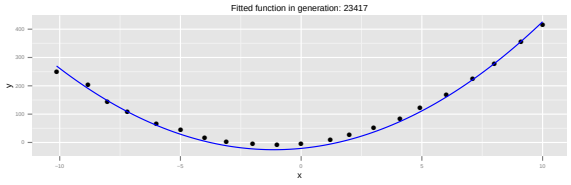

Fitted function in generation: 16099
x
y

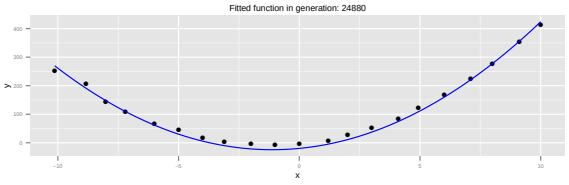

Fitted function in generation: 17563
x
y

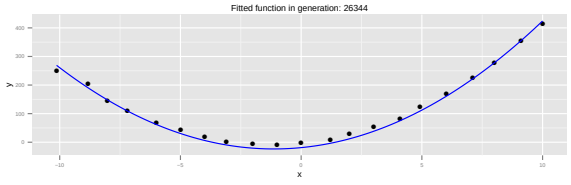

Fitted function in generation: 19026
x
y

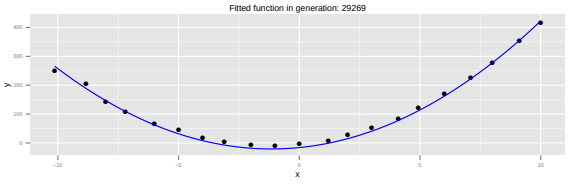

Fitted function in generation: 20490
x
y


Fitted function in generation: 21953
x
y


Fitted function in generation: 23417
x
y


Fitted function in generation: 24880
x
y


Fitted function in generation: 26344
x
y


Fitted function in generation: 29269
x
y

$(\mu + \lambda)$ evolution strategy

Functions used by $(\mu + \lambda)$ evolution strategy:

```
> #return function performing crossover
> #initDev - initial deviation
> muPlusLambdaCrossoverFun <- function(initDev){
+         function(temporaryPopulation,population,context){
+                 #fill in if deviations missing
+                 if(! "aDev" %in% colnames(temporaryPopulation)){
+                         population$aDev <- initDev
+                         temporaryPopulation$aDev <- initDev
+                 }
+                 if(! "bDev" %in% colnames(temporaryPopulation)){
+                         population$bDev <- initDev
+                         temporaryPopulation$bDev <- initDev
+                 }
+                 if(! "cDev" %in% colnames(temporaryPopulation)){
+                         population$cDev <- initDev
+                         temporaryPopulation$cDev <- initDev
+                 }
+
+                 popSize <- nrow(temporaryPopulation)
+                 shuffleIdx <- sample(popSize,popSize,replace = F)
+                 ##create crossed over population the same size as temporaryPopulation
+                 for(i in seq(1,popSize,2)){
+                         parentsIdx<-shuffleIdx[i:(i+1)]
+                         parents <- temporaryPopulation[parentsIdx,]
+                         alfa<-runif(1)
+                         #crossover the parents
+                         parents[1,c("a","b","c","aDev","bDev","cDev")]<-
+                                         alfa*temporaryPopulation[parentsIdx[1],
+                                                         c("a","b","c","aDev","bDev","cDev")]+
+                                         (1-alfa)*temporaryPopulation[parentsIdx[2],
+                                                         c("a","b","c","aDev","bDev","cDev")]
+                         parents[2,c("a","b","c","aDev","bDev","cDev")]<-
+                                         alfa*temporaryPopulation[parentsIdx[2],
+                                                         c("a","b","c","aDev","bDev","cDev")]+
+                                         (1-alfa)*temporaryPopulation[parentsIdx[1],
+                                                         c("a","b","c","aDev","bDev","cDev")]
+                         temporaryPopulation[parentsIdx,]<-parents
+                 }
+                 list(population=population,temporaryPopulation=temporaryPopulation)
+         }
+ }
> #return function performing mutation
> #crossOverProb - with what probabilty we perform crossover on a given pair
> muPlusLambdaMutationFun <- function(){
+         function(temporaryPopulation,population,context){
+                 K<-context[["K"]]
+                 mu<-nrow(population)
+                 gammaPrime<-K/sqrt(2*sqrt(mu))
+                 gamma<-K/sqrt(2*mu)
+                 for(i in 1:nrow(temporaryPopulation)){
+                         temporaryPopulation[i,c("a","b","c")] <-
+                                         temporaryPopulation[i,c("a","b","c")] +
+                                         temporaryPopulation[i,c("aDev","bDev","cDev")]*rnorm(3)
+                         temporaryPopulation[i,c("aDev","bDev","cDev")] <-
+                                         temporaryPopulation[i,c("aDev","bDev","cDev")] *
+                                         exp(gammaPrime*rnorm(1)+gamma*rnorm(3))
+                 }
+                 list(population=population,temporaryPopulation=temporaryPopulation)
+         }
+ }
```

Run $(\mu + \lambda)$ evolution strategy:

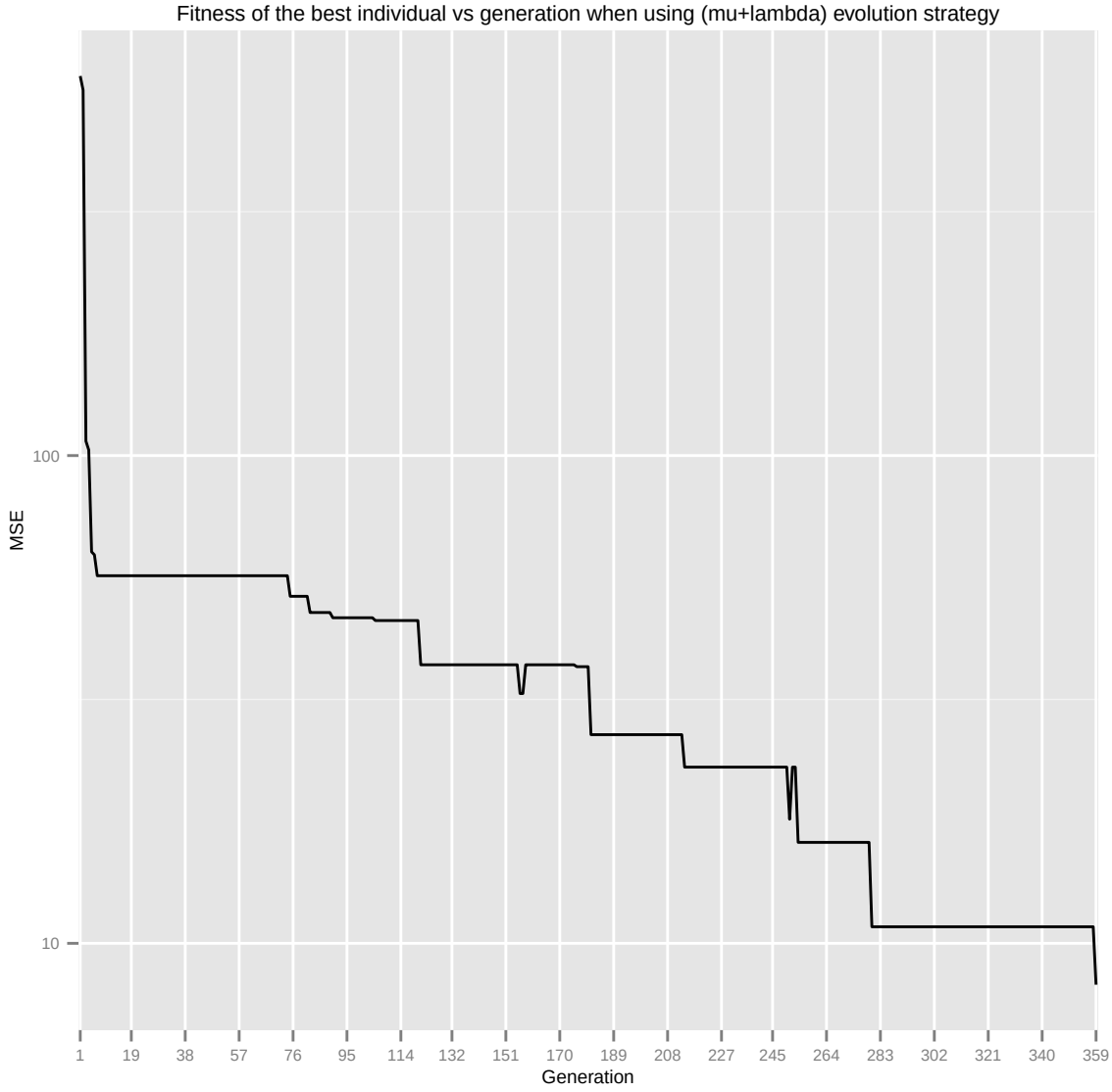
```
> #number of individulas in the initial population
> mu<-10
```

```
> #number of individuals selected to temporary population
> lambda<-20
> muPlusLambdaRes <- evolutionStrategy(
+                 #(mu+lambda) evolution strategy, start with population of mu individuals
+                 initialPopulation = generateInitialPopulation(size=mu,minA = -100, maxA = 100, minB = -100,
+                                 maxB = 100, minC = -100, maxC = 100),
+                 #(mu+lambda) evolution strategy, reproduce temporary population to size lambda
+                 createTemporaryPopulationFun = function(population){
+                         population[sample(1:nrow(population),size=lambda, replace=T),]
+                 },
+                 #for (mu+lambda) using mutation and crossover
+                 geneticOperations = c(muPlusLambdaCrossoverFun(initDev = 1),muPlusLambdaMutationFun()),
+                 #we combine temporary and original populations
+                 afterGeneticOperationsProcessor = function(temporaryPopulation,offspringPopulation) {
+                                 rbind(temporaryPopulation, offspringPopulation)
+                         },
+                 #select mu best individuals from the pool of the modified and original populations
+                 selectBestFun = selectBestFun(mu),
+                 #compute number of chromosomes better than the best in the previous generation
+                 computeContextAfterGenerationFun=function(previousPop,newPopulation){
+                         list("K"=sum(sort(newPopulation$fitness)-sort(previousPop$fitness) < 0))
+                 },
+                 #stop when the best individual has fintess better than
+                 stoppingCriterionFun = bestFitnessBetterThansStoppingCriterionFun(MAX_RMSE))
```

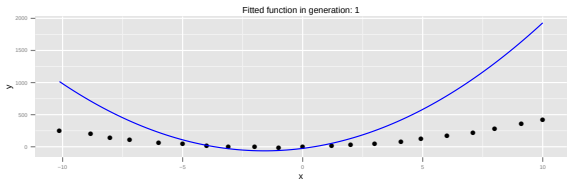
Parameters of the best solution are:

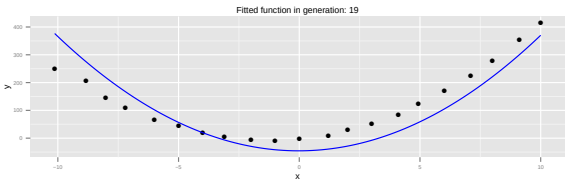
```
> muPlusLambdaRes$solution[order(muPlusLambdaRes$solution$fitness,decreasing = F)[1],]

         a        b         c  fitness generation     aDev      bDev     cDev
1 3.506481 8.909024 -5.483478 8.226289        359 52.99682 0.7378195 10.91198
```

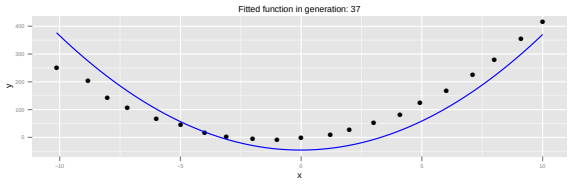
Show history of fitness (MSE) of individuals through generations when looking for the best solution using $(\mu + \lambda)$ evolution strategy:

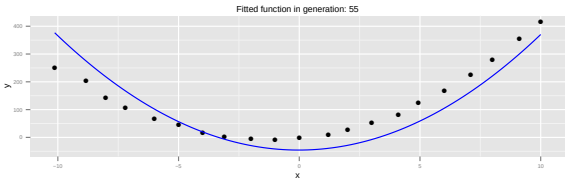


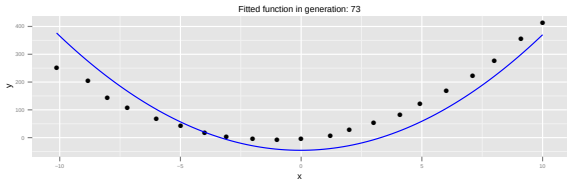
Show different functions that were found during finding optimal solution:

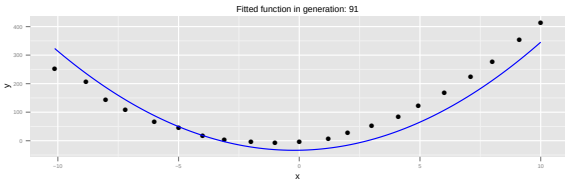

Fitted function in generation: 1
y
x

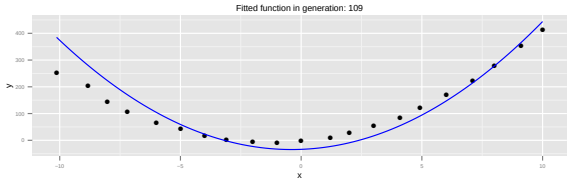

Fitted function in generation: 19
y
x

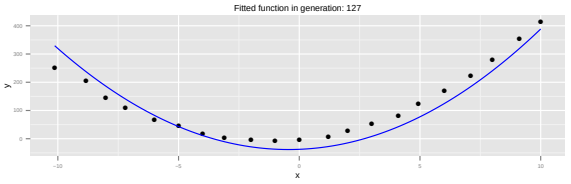

Fitted function in generation: 37
y
x

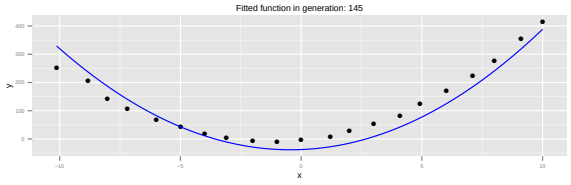

Fitted function in generation: 55
y
x

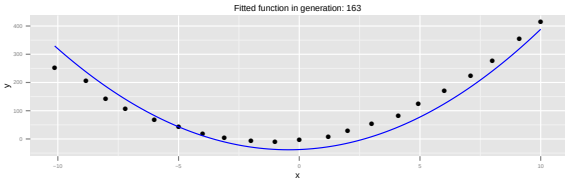

Fitted function in generation: 73
y
x

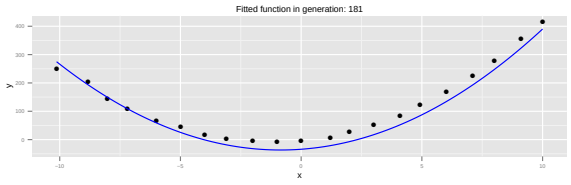

Fitted function in generation: 91
y
x

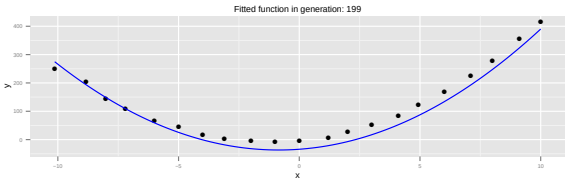

Fitted function in generation: 109
y
x

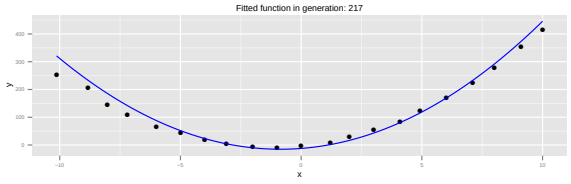

Fitted function in generation: 127
y
x

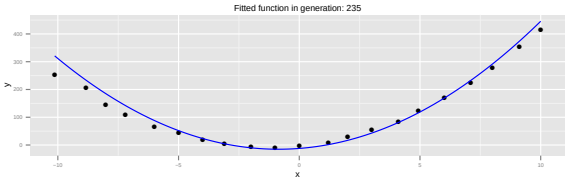

Fitted function in generation: 145
y
x

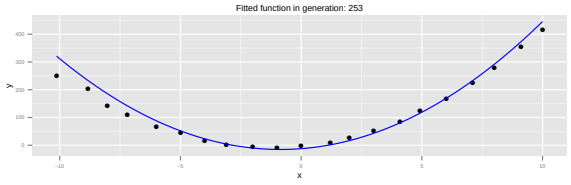

Fitted function in generation: 163
y
x

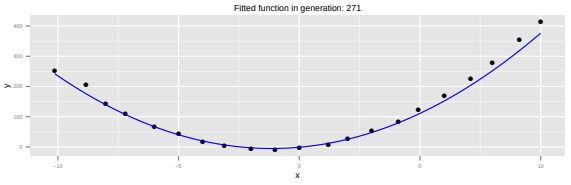

Fitted function in generation: 181
y
x

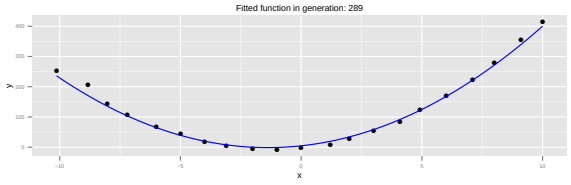

Fitted function in generation: 199
y
x

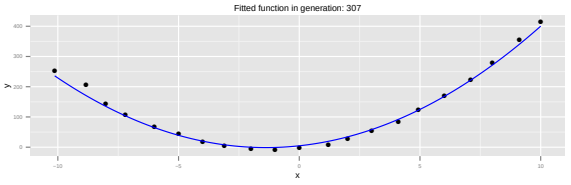

Fitted function in generation: 217
y
x

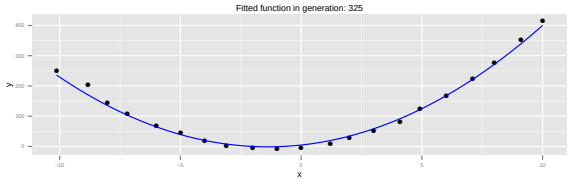

Fitted function in generation: 235
y
x

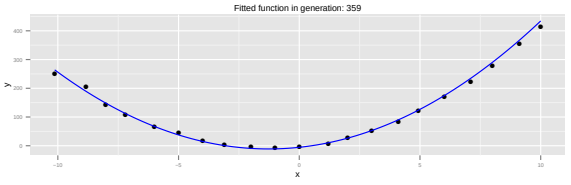

Fitted function in generation: 253
y
x


Fitted function in generation: 271
y
x


Fitted function in generation: 289
y
x


Fitted function in generation: 307
y
x


Fitted function in generation: 325
y
x


Fitted function in generation: 359
y
x

(μ, λ) evolution strategy

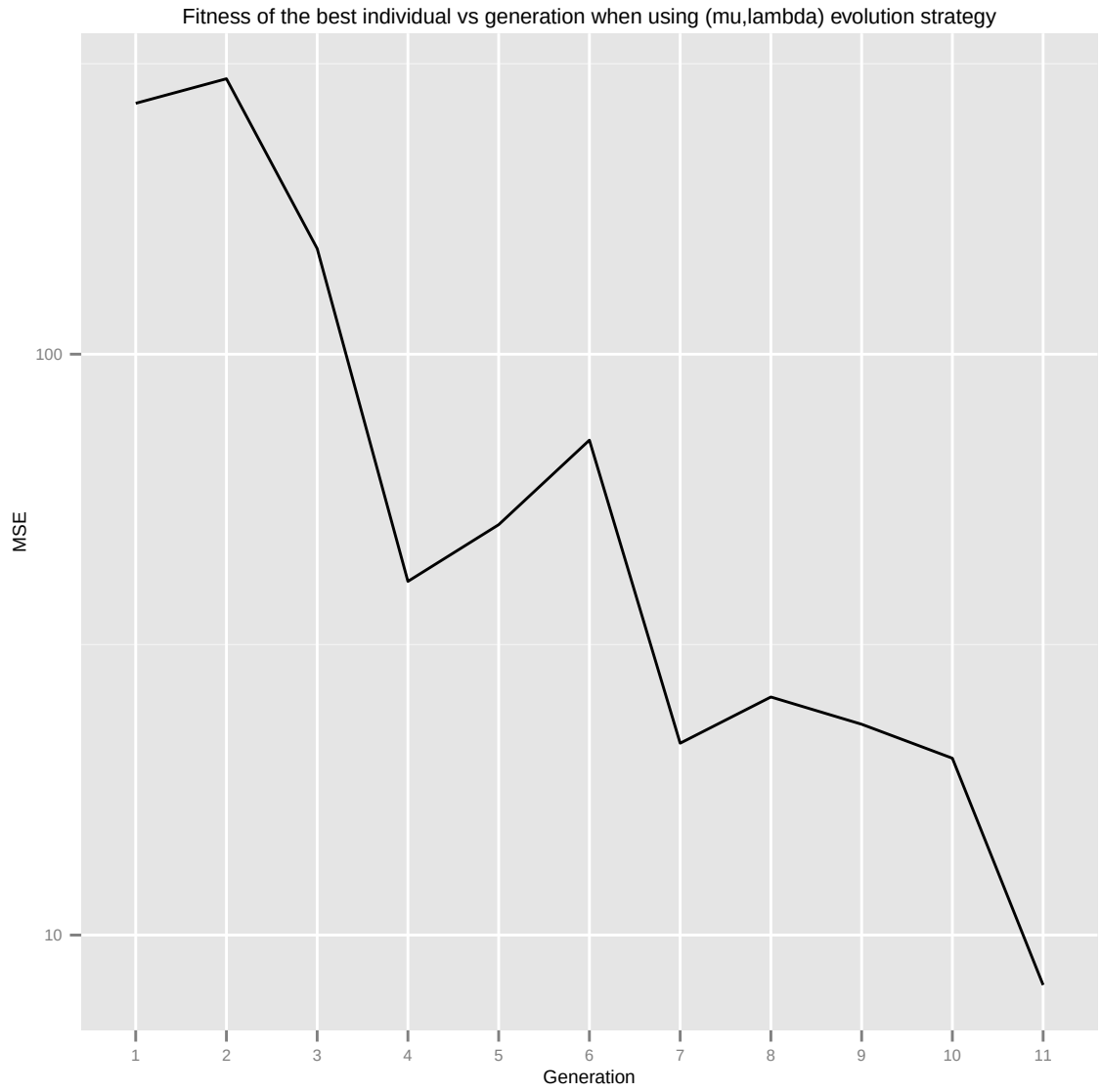
Run (μ, λ) evolution strategy:

```
> #number of individulas in the initial population
> mu<-10
> #number of individuals selected to temporary population
> lambda<-20
> muAndLambdaRes <- evolutionStrategy(
+                 #(mu,lambda) evolution strategy, start with population of mu individuals
+                 initialPopulation = generateInitialPopulation(size=mu,minA = -100, maxA = 100, minB = -100,
+                                 maxB = 100, minC = -100, maxC = 100),
+                 #(mu,lambda) evolution strategy, reproduce temporary population to size lambda
+                 createTemporaryPopulationFun = function(population){
+                         population[sample(1:nrow(population),size=lambda, replace=T),]
+                 },
+                 #for (mu,lambda) using mutation and crossover
+                 geneticOperations = c(muPlusLambdaCrossoverFun(initDev = 1),muPlusLambdaMutationFun()),
+                 #for (mu,lambda) we use only temporaryPopulation
+                 afterGeneticOperationsProcessor = function(temporaryPopulation,offspringPopulation) {
+                         temporaryPopulation
+                         },
+                 #select mu best individuals from the pool of the modified population
+                 selectBestFun = selectBestFun(mu),
+                 #compute number of chromosomes better than the best in the previous generation
+                 computeContextAfterGenerationFun=function(previousPop,newPopulation){
+                         list("K"=sum(sort(newPopulation$fitness)-sort(previousPop$fitness) < 0))
+                 },
+                 #stop when the best individual has fintess better than
+                 stoppingCriterionFun = bestFitnessBetterThansStoppingCriterionFun(MAX_RMSE))
```

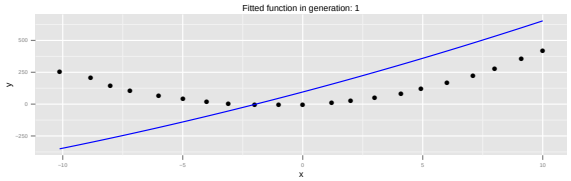
Parameters of the best solution are:

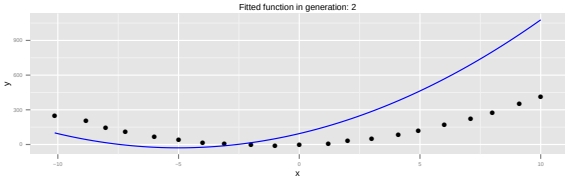
```
> muAndLambdaRes$solution[order(muAndLambdaRes$solution$fitness,decreasing = F)[1],]

         a        b        c  fitness generation       aDev         bDev
1 3.405486 8.276568 2.298179 8.205355         11 0.06710747 0.0001009383
      cDev
1 22.78981
```

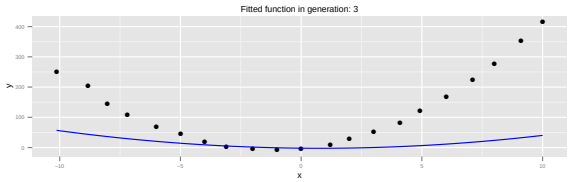
Show history of fitness (MSE) of individuals through generations when looking for the best solution using (μ, λ) evolution strategy:

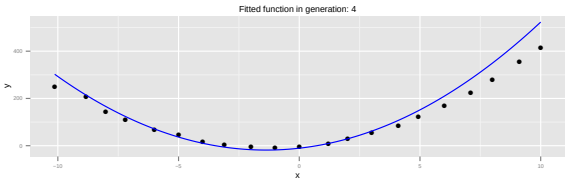


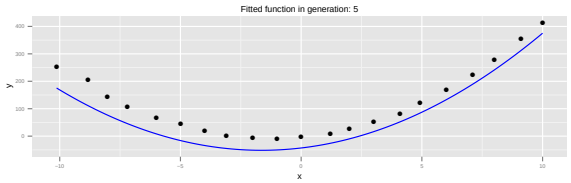
Show different functions that were found during finding optimal solution:

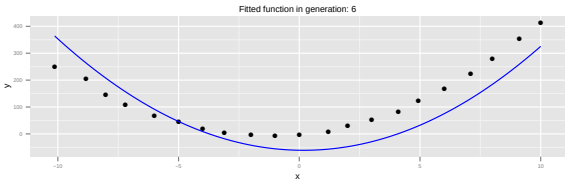

Fitted function in generation: 1
y
x

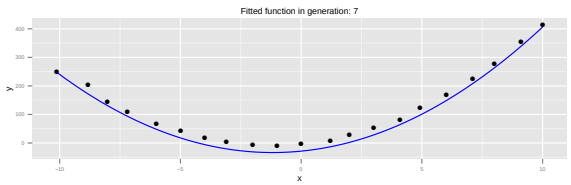

Fitted function in generation: 2
y
x

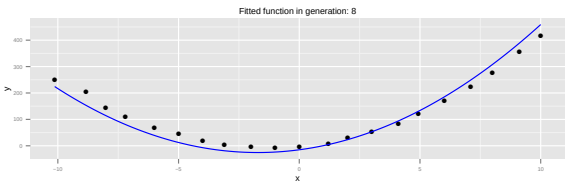

Fitted function in generation: 3
y
x

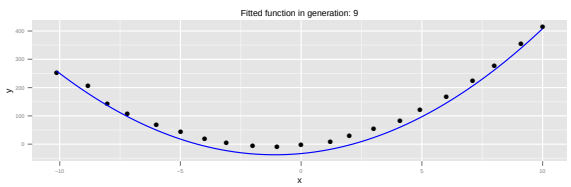

Fitted function in generation: 4
y
x

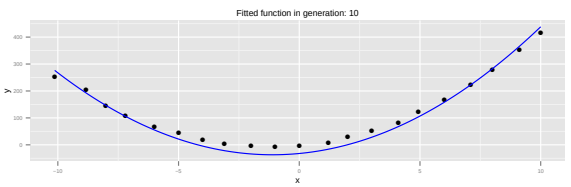

Fitted function in generation: 5
y
x

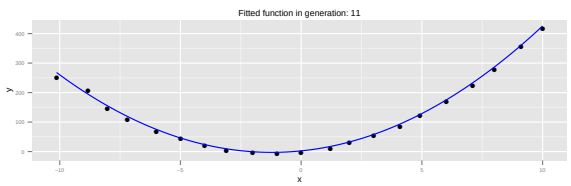

Fitted function in generation: 6
y
x


Fitted function in generation: 7
y
x


Fitted function in generation: 8
y
x


Fitted function in generation: 9
y
x


Fitted function in generation: 10
y
x


Fitted function in generation: 11
y
x

Summary

We can see that starting with bigger number of individuals in population we are able to converge quicker to satisfactory solution (as was stated in the lecture because we are able to start from different places in the search space we would explore it more thorougly). Also it looks that choosing only from the pool of modified individuals is more beneficial than combining previous population with the currently modified one.