# Optimization: Non-Gradient Method

Code

Ivan Yi-Fan Chen

2023 Fall

Hide

```r
library(tidyverse)
```

## 1 Non-gradient Based Methods

- Gradient methods can be restrictive:
  - Differentiability is required.
  - Errors in numerical differentiation.
  - Can be computationally costly, especially for the Hessian matrix.
  - **Local methods instead of global methods.**
- Non-gradient methods:
  - Do not rely on differentiability. (no more numerical differentiation)
  - Searches the **whole** space and are designed to **escape** from local optimalities.
  - Notable algorithms include Genetic Algorithm, Simulated Annealing, Particle Swarm Optimization and Nelder-Mead method.
  - There are restrictions. But we'll see…
- We go into Simulated Annealing and Particle Swarm Optimization as they can be easily implemented from scratch.
- Briefly mention Nelder-Mead and Genetic Algorithm since they are also popular out in the field.

### 1.1 Genetic Algorithm (just to mention)

- Concept was proposed in 1960s, but attentions are limited until the recent decades because of the drastic improvement in computers.
- Mimics the evolution of life: breeding, mutation and selection until the best is obtained.
  - **Initial population:** Several initial guesses to solutions, each of which are evaluated and their fitness to the problem is scored (say order them by the resulting function value).
  - **Selection:** Pairs of "parents" are chosen out from the guesses, and the probability to be chosen depends on the fitness score.
  - **Crossover:** The parents "exchange their genes" to generate several new solutions as their "off-springs".
  - **Mutation:** The "gene" of some of the off-springs are reordered randomly to become new solutions.
  - Use the off-springs, mutated or not, to evaluate the function and inspect their fitness. Us them as a new generation of initial population and keep repeating until convergence.
- How genes are defined:
  - All numbers are translated into binary numbers.

    For example, the binary representation of 3 is "11", 4 is "100", and 5 is "101" and so on.
  - Each binary digit is referred to as a **gene**, and a combination of the genes is referred to as **chromosome**.

    For example, suppose that we have four numbers: 0, 63, 43, and 54. They are represented as "000000", "111111", "101011", and "110110". Then 0 and 1 are genes and the resulting representations are chromosomes.
- Crossover and off-springs:

    - For the parents, convert them into binary and determine a “crossover point” by the binary digit.
    - The digits above the crossover points are exchanged to form off-spring.
    - Suppose that 0 (“000000”) and 63 (“111111”) are selected, then we can set the 3rd digit as the cross over point and do exchange. Then we get 56 (“111000”) and 7 (“000111”) as the off-springs.
    - Number and positions of genes to be exchanged between parents are flexible. What we talked about here is just an example.
- Mutation:
    - For randomly picked off-springs, shuffle its gene to form a new number.
    - Say 56 (“111000”) is selected to be mutated, then we might get 54 (“110110”) by shuffling the binary digits of 56.
- More literally interpretation:
    - Guess solutions and find the current best.
    - Crossover is to search for best results based on the current best results. Our illustrative numerical example resembles a “bracketing”.
    - Mutation allows us to **escape from local solutions**.
    - For whatever we get, evaluate their performance and then let them breed and mutate to gradually improve the performance. *Ore no Shikabane wo Koete Yuke!*
- Drawbacks:
    - Doesn’t look easy to implement by hand. Fortunately there are R packages for it, including `genlag` and `mcga`.
    - Computationally heavy and slow. But could be resolved by running on GPUs or parallel computation.
- Enlightened algorithms such as Simulated Annealing and Particle Swarm Optimization.
    - Simulated Annealing is highly Monte Carlo: randomly pick points near your guess, check fitness, and randomly “mutate” by temporarily taking a worse solution as a new guess.
    - Particle Swarm has a swarm of guesses, each particle evolve by referring to each other and random movements.

## 1.2 Simulated Annealing

- Features:
    - Random draw plus downhill search.
    - Given an initial point, randomly draw in the neighborhood for a new possible initial point.
    - Downhill search only takes better points as a new initial point throughout iterations.
    - Escape from local optimalities by **randomly** accepting a **worse** points as new initial points.
    - The “random acceptance” become **less likely** to occur as the iteration continues. Eventually the algorithm performs only downhill searches.
- What is “annealing”?
    - Annealing is a metallurgy and matel science jargon.
    - Think about forging a katana: https://www.youtube.com/watch?v=u3kkNhIk8Wc (https://www.youtube.com/watch?v=u3kkNhIk8Wc)
    - Heating up the metal so that the atoms in the metal are less stable, allowing the forge master to shape it into a katana before cooling down. But such a process causes the atoms to be clustered and bonded within the metal and reduces ductibility (hardened but easily broken).
    - The **annealing** process refers to heating up the metal again, and then cool it down either slowly or rapidly. Exactly what the “yakiire” process doing.

- During annealing process, heating up the metal activates the atoms again so that the atoms can diffuse and return to their equilibrium state and crystalize more easily. During the cooling down process, more and more atoms are crystalized and eventually stability is achieved so that the atoms no longer move around.

- Simulated annealing mimics the metallurgical annealing process:
    - Determine the "temperature parameter", which is reducing as the iteration goes.
    - The algorithm randomly search around the neighborhood of the initial point.
    - When the temperature is high, the algorithm is "unstable" in the sense that it is **very likely** to use a neighbor point as a new initial point for next iteration, even if it is a bad point to pick from the view of downhill algorithms.
    - The temperature reduces as we continue to iterate. But the lower the temperature, the **less likely** that the algorithm takes a bad point as a new initial point, just like how atoms become more stable in the metal.
    - For low enough temperature, the algorithm takes only better points as new initial points. Hence it is genuinely a downhill search then (aka **greedy algorithm**).
- Key components of simulated annealing:
    - A cooling schedule $t(i)$ that determines how the temperature goes down as the iteration continues. We require $t(i)$ to be decreasing in $i$ (otherwise it is not cooling down…)
    - A temperature scale $T_s$.
    - An acceptance probability function $P(e, e_{new}, T)$, where $T$ is the temperature at the iteration, $e$ and $e_{new}$ are the **energy levels** at the current initial point and the neighbor point. The **energy levels** simply refers to the function value evaluated at the point.
    - For **minimization** problem we require $P = 1$ if $e_{new} < e$, i.e., the neighborhood yields a lower value than the current initial point.

  But when $e_{new} > e$ we "accept" the neighborhood with a probability $P \in [0, 1]$, and this probability is **decreasing** in temperature $T$.

- The settings to the cooling schedule and acceptance probability are flexible, but typically:
    - The acceptance probability when $e_{new} > e$ is usually specified as

$$P = e^{-\frac{e_{new}-e}{T}} = \frac{1}{e^{\frac{e_{new}-e}{T}}}$$

        - The acceptance probability is larger for higher $T$ or lower $e_{new} - e$.
        - Higher $P$ for higher $T$ is the spirit of annealing.
        - Higher $P$ for lower $e_{new} - e$ is reasonable: **energy flows from high to low**. Even if we need to climb uphills for a downhill problem, **we should not be climbing up too much**.
    - Two frequently used settings to the cooling schedule:
        1. $t(i) = t^{i-1}$ where $t \in (0, 1)$ so that $T = t^{i-1} T_s$. Larger $t$ prolongs the cooling time.
        2. $t(i) = (1 - s)^{i-1}$ where $s \in (0, 1)$ so that $T = (1 - s)^{1-i}$. Smaller $s$ prolongs the cooling time.
        - Temperature reduces in $i$ for both settings.
        - Setting 1 has a thinner right tail, hence the rate of cooling is much faster in the early stage than in Setting 2.
        - Cooling schedule and temperature controls how long that the algorithm stays aggressive. A high $T_s$ or a cooling schedule that cools down slowly prolongs the aggressive stage.
- The probability distribution used to draw random neighborhood is also flexible.
    - Normal distribution with zero mean is an option. One can choose the standard deviation to determine how likely that an extreme value is used as a neighbor point.

    - Uniform distribution is also used. One might want to use $U[0,1]$ and transform the problem into this interval.
- General procedure:
    - Choose an arbitrary initial point, evaluate the function, and set it as the best outcome.
    - Draw a neighbor point, evaluate and compare with the current initial point and the best point.
    - If the neighbor is better than the best one, update the best with the neighbor point.
    - Relocate initial point:
        - If the neighbor is better than the current initial point, move to the neighbor.
        - If the neighbor is worse, then move to the neighbor by probability $P(e, e_{new}, T)$.
    - Repeat the procedure above until the iteration stops, say when running out of iteration number, when the temperature falls below a specified level, or when the convergence conditions are meet (maybe that the best point is not updated for a long enough time).
- But this algorithm is so dicey! Can we be so lucky to reach the global minimum?
- The answer is YES. The original paper has shown that SA **converges to the global optimal with probability one**. This is just an application of Law of Large Numbers.
- But it takes a lot of iterations to do so. Hence SA is not an efficient algorithm.
- Demonstrate with our old friend $y = x^4 - 15x^2 + 8x + 30$. Recall that it has two minima at $x \approx \{-2.88, 2.56\}$.

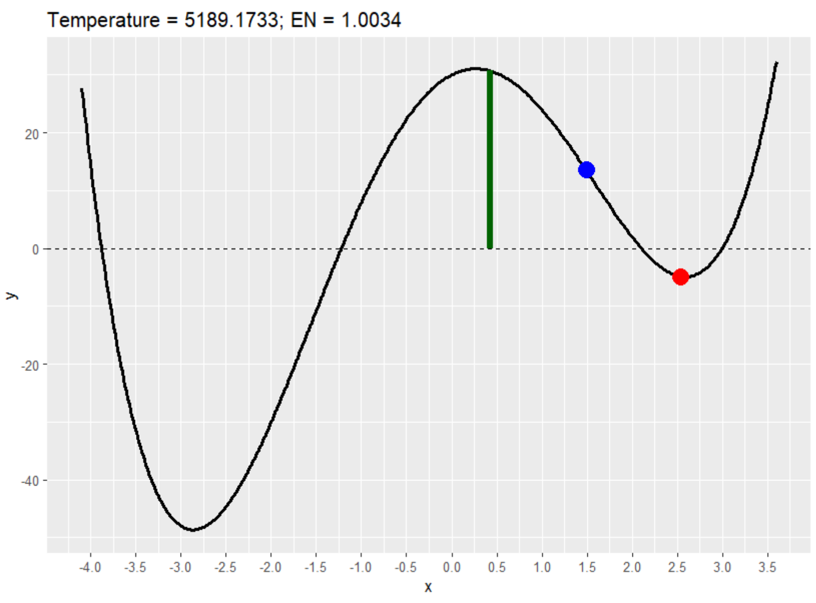
Hide

```
pf<-function(x) {x^4-15*x^2+8*x+30}
```

- Green bar is the initial point in the iteration, blue point is the neighbor point drawn near the initial point, and red point is the best point up to the current iteration.

**Iteration**

5



## 1.2.1 Implementation on R

- Algorithm:

    **Step 0:** Choose initial temperature $T_s$, cooling schedule $t(i)$, acceptance probability $P(e, e_{new}, T)$, initial point $x_0$, probability distribution to draw neighbor points, and maximum number of iterations. If applicable, also other conditions to stop the iteration.

    **Step 1:** Initialize by evaluate $f(x)$ at $x_0$, document it as the best one $x_{best}$.

    **Step 2:** Draw a random number from the distribution as $dx$. Then compute the neighbor point as $x_{neighbor} \equiv x_0 + dx$.

**Step 3:** Compare $f(x_{neighbor})$ with $f(x_{best})$. If $f(x_{neighbor}) < f(x_{best})$, update $x_{best}$ with $x_{neighbor}$.

**Step 4:** Compare $f(x_{neighbor})$ as $f(x_0)$. Then:

If $f(x_{neighbor}) < f(x_0)$, use $x_{neighbor}$ as the new $x_0$.

If $f(x_{neighbor}) > f(x_0)$, then compute $P(e, e_{new}, T)$. Draw a random number $r$ from $U[0, 1]$, and use $x_{neighbor}$ as the new $x_0$ if $r < P(e, e_{new}, T)$. Otherwise keep using the current $x_0$ as the initial point.

**Step 5:** Repeat Steps 2 to 4 until we run out of iteration or until the terminating conditions are meet. Report the latest best result as the outcome.

- You can't control the random draw $dx$, hence you may run into a situation that $x + dx$ falls out of the boundary. You may use an `if-else` to detect this, and force $x + dx$ to be the boundary point if it goes beyond.
- Home-made simulated annealing algorithm `simann`
  - `simann(fun,x0,bnd,imax,s,...)`
  - `fun` is a function, and potentially has multiple arguments.
  - `x0` is a named numeric vector. The names contains the names of variables to be optimized, and the elements are the initial value for these variables. For example, `x0=c(x=1,y=2,z=3)`
  - `bnd` is a list containing two numeric vectors of equal length. The first vector `bnd[[1]]` is the lower bound to search for each of the variables, and the second vector `bnd[[2]]` is for upper bounds. For example, `bnd=list(c(0,1,2), c(3,4,5))`
  - `sd` is the standard deviation of the Normal random draw.
  - `imax` is the maximum amount of iteration defaulted to 1000.
  - `s` is the step size to determine the temperature gradient.
  - `Ts` is a scaling factor to temperature defaulted to 1.
  - Returns a list containing the best `x` and objective value by the end of iteration, as well as histories for current, neighbor and best points and values during the iteration.

Hide

```r
#Simulated annealing with Normal Distribution

simann<-function(fun,x0,bnd,sd=1,imax=1000,s=1e-3,Ts=1,...) {

  #Check eligibility
  if(!is.function(fun)) {stop("fun must be a function.")}
  if(is.null(names(x0))) {stop("x0 must be named.")}
  if(!is.numeric(x0)||!is.vector(x0)) {stop("x0 must be a numeric vector.")}
  if(!is.list(bnd)||length(bnd)!=2) {stop("bnd must be a list of length 2.")}
  if(!is.vector(bnd[[1]])||!is.vector(bnd[[2]])) {stop("Elements of bnd must be vectors")}
  nx<-length(x0)
  if(length(bnd[[1]])!=nx||length(bnd[[2]])!=nx) {stop("Invalid boundaries.")}
  if(sum(bnd[[1]]>bnd[[2]])!=0) {stop("Lower bounds exceed upper bounds.")}

  #Construct lower and upper bounds
  lb<-bnd[[1]]
  ub<-bnd[[2]]

  #Replace out-of-bound arguments with boundary values
  x0[which(x0<lb)]<-lb[which(x0<lb)]
  x0[which(x0>ub)]<-ub[which(x0>ub)]


  #Predraw parameter from N(0,sd) and store into a matrix with each column for one variable.
  drwx<-rnorm(n=(nx*imax),mean=0,sd=2)%>%matrix(nrow=imax,ncol=nx)
  #Predraw for random relocation
  drwl<-runif(n=imax,min=0,max=1)

  #Initialize current, neighbor, and best x and values.
  x_current<-x0
  x_neighbor<-x0
  x_best<-x0

  val0<-do.call(fun,as.list(x0,...))
  val_current<-val0
  val_neighbor<-val0
  val_best<-val0

  #Initialize history
  history.current<-data.frame(matrix(NA,nrow=(1+imax),ncol=(2+nx)))
  history.current[1,]<-c(0,x_current,val_current)
  colnames(history.current)<-c("i",names(x_current),"val_current")
  history.neighbor<-history.current
  colnames(history.neighbor)[3]<-"val_neighbor"
  history.best<-history.current
  colnames(history.best)[3]<-"val_best"

  #Start looping
  for (i in 1:imax) {

    temp<-(1-s)^i #Temperature schedule.

    x_neighbor<-x_current+drwx[i,] #Draw neighbor point, but need to deal with out-of-bound issue.
    x_neighbor[which(x_neighbor<lb)]<-lb[which(x_neighbor<lb)]
    x_neighbor[which(x_neighbor>ub)]<-ub[which(x_neighbor>ub)]
    val_neighbor<-do.call(fun,as.list(x_neighbor,...)) #Evaluate value at neighbor point.
    #If neighbor point yields a lower value than the current best, update the best record.
    if(val_neighbor<val_best) {
      x_best<-x_neighbor
      val_best<-val_neighbor
    }

    history.current[i+1,]<-c(i,x_current,val_current)
    history.neighbor[i+1,]<-c(i,x_neighbor,val_neighbor)
    history.best[i+1,]<-c(i,x_best,val_best)

    #Move from current point to neighbor point if either the value of objective function is lower or simply randomly.
    if(val_neighbor<val_current||drwl[i]<exp(-(val_neighbor-val_current)/(Ts*temp))) {
      x_current<-x_neighbor
      val_current<-val_neighbor
    }

    if(i==imax) {
      cat("Iteration ended.\n")
      cat("Best parameters:",paste(names(x_best),x_best,sep=" = "),"\n")
      cat("Best value:",val_best,"\n")
      rst<-list(x_best=x_best,val_best=val_best,history_current=history.current,
                history_neighbor=history.neighbor,history_best=history.best)
      return(rst)
    }

  }

}
```

- The algorithm gets the global minimum in 100 iterations.

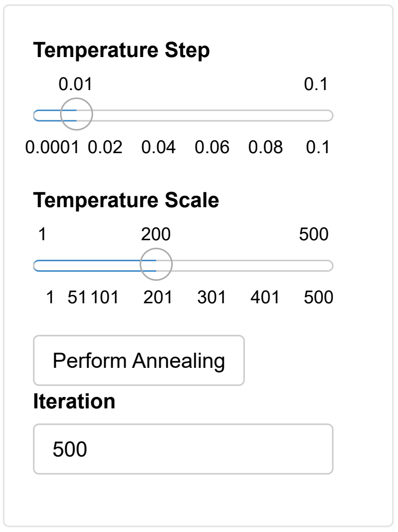
Hide

```
#Univariate case

x0<-c(x=2)
bnd<-list(-4,3.5)
rst<-simann(pf,x0,bnd,sd=3,imax=100)
```

```
## Iteration ended.
## Best parameters: x = -2.8685432669536
## Best value: -48.66757
```

- Role of temperature schedule and scaling factor. Observe how the pattern vary with different settings to `s` and `Ts`.



**Temperature Step**

0.01 — 0.1

0.0001 0.02 0.04 0.06 0.08 0.1

**Temperature Scale**

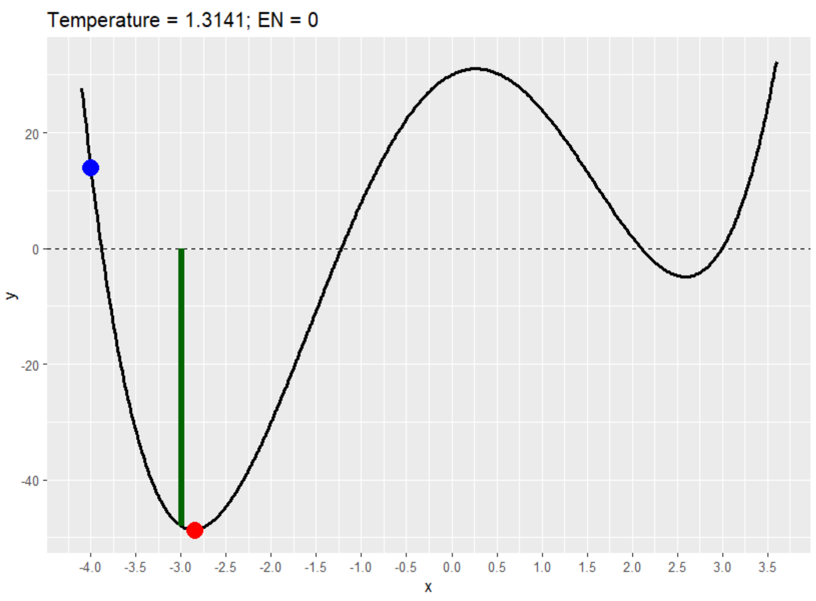
1 — 200 — 500

1 51 101 201 301 401 500

Perform Annealing

**Iteration**

500



- Easily adapted to a multivariate environment.

Hide

```r
#Multivariate case
ufn<-function(x,y){x^2+y^2}

x0<-c(x=5,y=7)
bnd<-list(c(0,0),c(10,10))
rst<-simann(ufn,x0,bnd,sd=3,imax=100)
```

```
## Iteration ended.
## Best parameters: x = 0 y = 0
## Best value: 0
```

### 1.2.2 Discussions

- Performance of SA can be bad for the following reasons:
  1. The random draw of $dx$ is not "open minded" enough. Say a normal distribution is unlikely to give you a far enough point. You can get trapped in a local valley just because your draws don't take you outside of it (and not guaranteed because of $P$).
  2. If we use $P = e^{-\frac{e_{new}-e}{T}}$, it is possible that the algorithm is too passive / aggressive because $e_{new} - e$ and $T$ are not to the same scale. For example, if we choose $T_s = 1$ then the denominator is no larger than $1$. In contrast $e_{new} - e$ is usually larger than 1, hence we have a very very small $P$ even in the very beginning!
- Refinements against the issues above:
  1. When using $T_s = 1$, normalize the objective function. It can vary from case to case. For example, when running a regression $y_i = \beta_0 + \beta_1 x_i + \epsilon_i$ where $i$ denotes individuals / observations, one can divide this regression by $y_{max} - y_{min}$ where $y_{max}$ and $y_{min}$ respectively denote the highest and lowest $y$ in data. In this way the "moment condition" $y_i - \beta_0 - \beta_1 x_i$ is within $[0, 1]$ hence is to the same scale with the normalized temperature. This applies to each of the moment conditions used in both GMM and SMM.

  2. Choose $T_s$ using the information of the function. This is flexible, but here we provide a possible way: Draw two random points from the variable space, evaluate them with the function and take the difference, i.e., $\Delta \equiv |f(x_1) - f(x_2)|$ where $x_1$ and $x_2$ are the two random draws. Then use $T_s \equiv \Delta / \ln p_0$ where $p_0 \in (0, 1)$ is

chosen by the user.

3. Search in narrow intervals. This is for practical purpose: looking at the whole real line is going to take forever! Moreover, we already have some priors about the parameters we will be checking. This is particularly true if we use simulated annealing for calibrating / estimating theoretical parameters, where we can search for these parameters following the literature.

4. Normalize the parameters, and draw from $U[0,1]$. For example, if we know that a parameter falls somewhere within the interval $(-5, 995)$ of range $1000$, then we apply a transformation $y = (x+5)/1000$ so that the parameter space becomes $[0,1]$. Drawing from this interval is much more efficient as all outcome, including extreme values, come with the same probability. An alternative way is to draw from $U[0,1]$ and then recover the parameter using the above transformation before plugging into the function.

- Drawbacks of SA:
  - Can still have difficulties getting out of a local valley. For example, the global valley is really deep, but it is surrounded by terrace like flats so that uphill movements can happen quite often (since $e_{new} - e$ is small). While the local valley is narrow, it is right in the middle of cliffs so that $e_{new} - e$ is much higher, causing it difficult to escape from.
  - Can take **forever** to really get to the global optimal.

### 1.2.3 In-Class Exercise

- Revise `simann` such that it normalizes variables of interest into $[0,1]$. You should achieve the following points:
  1. The objective function is general. Hence the normalization above should be applicable to all general variables. That is, if the function takes $x, y, z$ as inputs and we want to search for all of them, then `simann` need to normalize these variables. Note that you **cannot** control how the variables are named by the user.
  2. The normalization must be applicable to any arbitrary upper and lower bounds supplied by the user. For example, if $x \in (-\infty, \infty)$, $y \in (-5, 3)$ and $z \in (0, \infty)$, your algorithm need to identify the boundaries for each of the variables and normalize them correspondingly such that all variables are in $(0,1)$.

  I suggest that we first figure out the appropriate transformation $v \equiv t(x) \in (0,1)$ for each cases, and supply the inverse transformation $x = t^{-1}(x)$ in `simann` in a hard-coded manner. After drawing $v$ from $(0,1)$ we can then recover $x$ from $t^{-1}(x)$ and supply to the user-supplied function. The first point thus requires some works via `if-else`. For the second point, we have introduced ready-to-use $t(x)$ long ago. Check your past lecture notes if you don't want to solve by hands.

## 1.3 Particle Swarm Optimization

- Searching for the optimal on ones own is not easy, so how about searching with a group and referring to the findings of group members? Think of how ants and bees explore for food.
  - Lots of insects wandering in the field to look for food by tracing its smell.
  - Going around seemingly randomly, but then start to cluster towards the food if any of the member managed to locate it.
  - Each of the insect refers to its own findings as well as findings of the **swarm** to determine where the food is.
- Or think of how Straw Hats Pirates looking for the One Piece:
  - All members scattered in the world, each of them explore around their location to look for clues that lead to the One Piece.
  - Each member document the clues they found today, and submit these clues back to Thousand Sunny, their HQ. Then Thousand Sunny send the findings thus far by all members to each members.
  - Given the logs from Thousand Sunny, each member knows his **own** best record, and the **group's** best record.

- Referring to these records, the member determines their new location to search for clues. His decision depends on his **current direction of searching**, **his own best location**, and the **group's best location**.
- The member insists on his own direction to go, but at the same time he should also try to move towards his past best location and the group's past best location to really be consistent with clues found thus far. Eventually a new direction is determined based on these information.
- Keep doing so, then Straw Hat Pirates reach the One Piece.

- The PSO algorithm is an implementation of the idea of **simulating swarm behaviors** to search for the optimal solution.
    - A swarm intelligence algorithm.
- Key settings:
    - Number of **particles** $x_i$ to be spread in the space.
    - Random draw to the moving direction for each of the particles.
    - Each particle also checks how far away it is from both its best location thus far $x_{best,i}$, and the swarm's best location $x_{best,global}$ thus far.
    - **Inertia weight** $w$: scales with the **previous** moving direction to determine current moving direction. Higher $w$ means that the particle tend to follow the previous move.
    - **Cognitive weight** $c_1$: controls to what extent that the particle sticks with its best location (best experience). Think of Usopp tries to argue with his friends to stay on an island.
    - **Social weight** $c_2$: controls how important that the group's best matters for the particle's next move. Think of it as a peer pressure.
    - But particles do not take the best records completely! The random draw $r \in (0,1)$ for each iteration determines whether the particle actually refers to the best records. Note that $r$ is applied to **all** particles instead of particle-specific.
- Together we have a particle $i$'s direction of moving at iteration $t$, $dx_{i,t}$ as

$$dx_{i,t} = w \cdot dx_{i,t-1} + r \cdot c_1 (x_{i,t-1} - x_{best,i}) + r \cdot c_2 (x_{i,t-1} - x_{best,global})$$

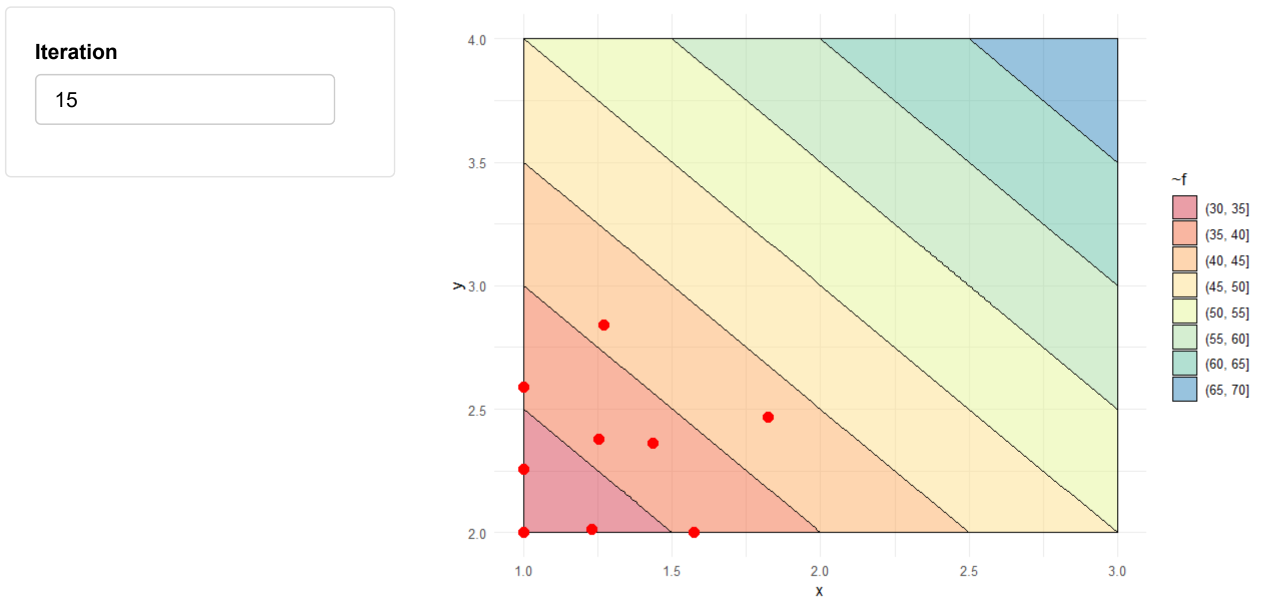
  The particle's location after this movement is $x_i, t = x_{i,t-1} + dx_{i,t}$.

- Note that in a higher dimension world, say (x,y), we obtain the movement along each axis separately as

$$\begin{bmatrix} dx_{i,t} \\ dy_{i,t} \end{bmatrix} = w \begin{bmatrix} dx_{i,t-1} \\ dy_{i,t-1} \end{bmatrix} + r \cdot c_1 \begin{bmatrix} x_{i,t-1} - x_{best,i} \\ y_{i,t-1} - y_{best,i} \end{bmatrix} + r \cdot c_2 \begin{bmatrix} x_{i,t-1} - x_{best,global} \\ y_{i,t-1} - y_{best,global} \end{bmatrix}$$

- General procedure:
    1. Spread the particles on the space. Evaluate the function values, document them as the best results thus far for each particle, and document the best one as the global best.
    2. Draw steps $dx$ for each particle on each dimension, draw $r$. Determine the direction of movement hence new location for each particle by referring to $dx$, best of the particle and global best.
    3. Evaluate at the new location, update particle's best and global best.
    4. Repeat 2 and 3 until the end of iteration.

- An illustration with $f(x,y,z) = (x+y)z$ with $z = 10$ on the 2-d space $x \times y = [1,3] \times [2,4]$



Iteration

15

### 1.3.1 Implementation on R

- Algorithm:

**Step 0:** Determine maximum number of iterations $imax$, weights $w$, $c_1$ and $c_2$, number of particles, how particles are spread in the space. Can spread with a equidistant grid, randomly, or with LDS.

**Step 1:** Evaluate $f(x)$ for each particle, document as $x_{best}$ (since this is the very first time), and document the best among $x_{best}$ as the global best.

**Step 2:** Draw $dx$ for each of the particle from a distribution (usually $U[0,1]$), draw $r$ from $U[0,1]$, then compute $dx_{i,t}$.

**Step 3:** Update locations for particles as $x_{i,t} = x_{i,t-1} + dx_{i,t}$, evaluate $f(x)$ for each particle. Then:

a. For each particle, update $x_{best,i}$ if $f(x_{i,t}) < f(x_{best,i})$.

b. For the swarm, update $x_{best,global}$ if any of the $x_{best,i}$ yields a lower value than the global best.

**Step 4:** Repeat Steps 2 and 3 so that the particles move around and search for the optimal until the end of iteration. Then report the global best value and location as the finding.

- Note that we need to include an `if-else` syntax to deal with boundary problems.

Hide

```
#PSO using an equidistance grid and uniform distribution (Grid is suggestive. Also consider Halton an
d random)

PSO<-function(fun,bnd,n=3,w=0.8,c1=0.05,c2=0.05,imax=1000,...) {

  #Check eligibility
  if(!is.function(fun)) {stop("fun must be a function.")}
  if(!is.list(bnd)||names(bnd)!=c("var","lb","ub")) {
    stop("bnd must be a named list with names being c(\"var\",\"lb\",\"ub\").")}
  if(length(bnd[[1]])!=length(bnd[[2]])||length(bnd[[1]])!=length(bnd[[3]])) {
    stop("Length of bnd$var, bnd$lb and bnd$ub do not align.")
  }
  if(!is.character(bnd[[1]])||!is.numeric(bnd[[2]])||!is.numeric(bnd[[3]])) {
    stop("bnd$var must contain variable names, and bnd$lb and bnd$ub must contain lower and upper bou
nds of the variables.")}
  if(sum(bnd[[2]]>bnd[[3]])!=0) {stop("Lower bounds exceed upper bounds.")}

  #Generate grid, total points = n^d
  lb<-bnd$lb
  ub<-bnd$ub

  d<-length(bnd$var)
  gd<-lapply(c(1:d),function(x){seq(lb[x],ub[x],length.out=n)})%>%
    do.call(cbind,.)%>%'colnames<-'(bnd$var)
  c<-ncol(gd)
  unit<-rep(1,n)
  location<-gd[,1]
  i<-1
  while(i<c) {
    location<-cbind(location%x%unit,rep(gd[,i+1],n))
    i<-i+1
  }

  #Bind to input arguments
  colnames(location)<-bnd$var
  arg<-cbind(location,...)
  #Initial perturbation
  pbest.val<-apply(arg,MARGIN=1,
                   FUN=function(x){
                     x<-as.list(x)
                     do.call(fun,x)})
  pbest.loc<-location
  gbest.val<-min(pbest.val)
  gbest.loc<-pbest.loc[which.min(pbest.val),]

  #Prepare history
  hist.loc<-cbind(i=0,location)
  hist.pbest.val<-data.frame(matrix(NA,nrow=imax+1,ncol=(1+nrow(location))))%>%
    'colnames<-'(c("i",paste("pbest",c(1:(n^d)),sep="_")))
  hist.pbest.val$i<-c(0:imax)
  hist.pbest.val[1,-1]<-pbest.val
  hist.val<-hist.pbest.val
  hist.pbest.loc<-hist.loc
  hist.gbest.val<-hist.gbest.val<-data.frame(i=c(0:imax),gbest=rep(0,(1+imax)))
  hist.gbest.val[1,2]<-gbest.val
  hist.gbest.loc<-data.frame(matrix(NA,nrow=imax+1,ncol=(1+length(bnd$var))))%>%
    'colnames<-'(c("i",bnd$var))
```

```r
  hist.gbest.loc[,1]<-c(0:imax)
  hist.gbest.loc[1,-1]<-gbest.loc

  #Initial moving distance, we have d variables and n^d points
  dx<-matrix(runif(d*(n^d)),ncol=d,nrow=n^d)%>%'colnames<-'(bnd$var)
  #Predraw for random movements towards gbest and pbest
  dc<-matrix(runif(2*imax),ncol=2,nrow=imax)

  i<-1
  while(i<=imax) {
    #Update location
    gbl<-t(as.matrix(gbest.loc))%x%matrix(1,ncol=1,nrow=(n^d))
    dx<-w*dx+c1*dc[i,1]*(pbest.loc-location)+c2*dc[i,2]*(gbl-location)
    location<-location+dx
    #Check boundary, if violated then force to be on the boundary
    for(j in 1:length(bnd$var)) {
      location[ location[,j]>ub[j], j]<-ub[j]
      location[ location[,j]<lb[j], j]<-lb[j]
    }

    #Update value
    arg<-cbind(location,...)
    val<-apply(arg,MARGIN=1,
               FUN=function(x){
                       x<-as.list(x)
                       do.call(fun,x)})

    #Find and update pbest
    ipbest<-which(val<=pbest.val)
    pbest.val[ipbest]<-val[ipbest]
    pbest.loc[ipbest,]<-location[ipbest,]

    #Find and update gbest
    igbest<-which.min(pbest.val)
    gbest.val<-pbest.val[igbest]
    gbest.loc<-pbest.loc[igbest,]

    #Update history
    hist.val[(i+1),-1]<-val
    hist.loc<-rbind(hist.loc,cbind(i=i,location))
    hist.pbest.val[(i+1),-1]<-pbest.val
    hist.pbest.loc<-rbind(hist.pbest.loc,cbind(i=i,pbest.loc))
    hist.gbest.val[(i+1),-1]<-gbest.val
    hist.gbest.loc[(i+1),-1]<-gbest.loc
    i<-i+1
  }
  rst<-list(GBestVal=hist.gbest.val,GBestLoc=hist.gbest.loc,
          PBestVal=hist.pbest.val,PBestLoc=hist.pbest.loc,
          Val=hist.val,Loc=hist.loc)
  return(rst)
}
```

- Perform PSO to the previous example $f(x, y, z) = (x + y) * z$.

Hide

```r
bnd<-list(var=c("x","y"),lb=c(1,2),ub=c(3,4))
fnn<-function(x,y,z) {(x+y)*z} #Use z=10

rs<-PSO(fnn,bnd,imax=50,z=10)
print(rs$GBestVal[51,])
```

```
##     i gbest
## 51 50    30
```

Hide

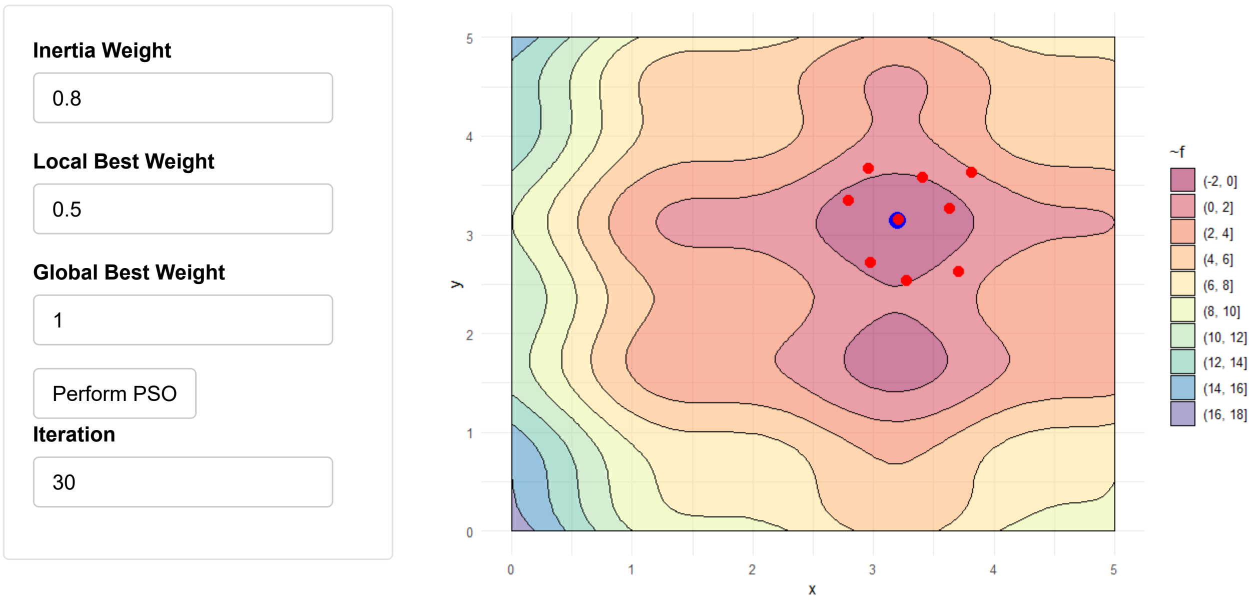
```r
print(rs$GBestLoc[51,])
```

```
##     i x y
## 51 50 1 2
```

- Try out the function

$$f(x,y) = (x-3.14)^2 + (y-2.72)^2 + \sin(3x+1.41) + \sin(4y-1.73)$$

This function has several local minima and one global minimum (somewhere near $x = 3.15, y = 3.2$, I do not check).



**Inertia Weight**

0.8

**Local Best Weight**

0.5

**Global Best Weight**

1

Perform PSO

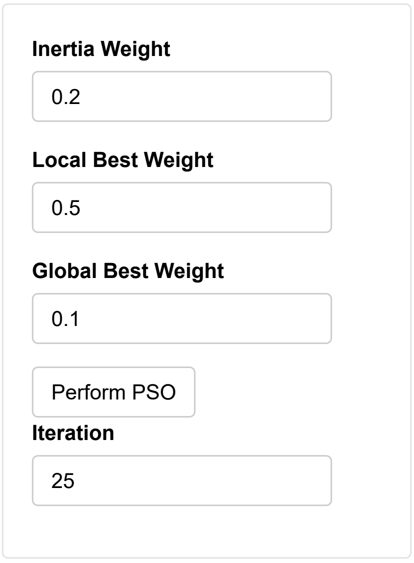
**Iteration**

30

- Try out Himmelblau's function

$$f(x,y) = (x^2 + y - 11)^2 + (x + y^2 - 7)^2$$

It has 4 local / global minima all taking values of $0$: $\{(3,2),\ (-2.81, 3.13),\ (-3.78, -3.28),\ (3.58, -1.85)\}$.



**Inertia Weight**

0.2

**Local Best Weight**

0.5

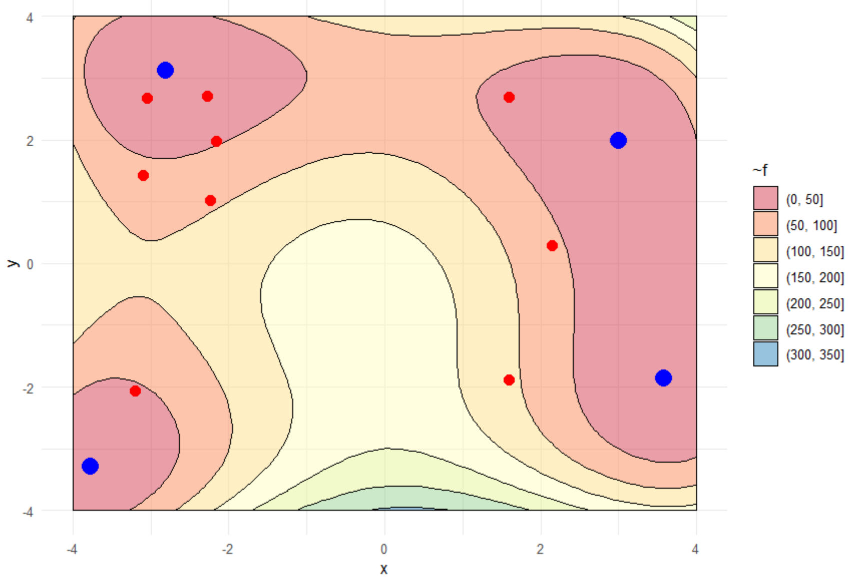
**Global Best Weight**

0.1

Perform PSO

**Iteration**

25



### 1.3.2 Discussions

- Higher inertia weight leads to more **exploration**: particles tend to be scattered with active movements instead of converging to a point.
- Higher social weight leads to quick convergence of particles; **exploitation**.
- Higher cognitive weight leads to a “prejudice” in the sense that the particles tend to stick with its own location. Not much movements.
- Exploration and exploitation needs to be balanced:
    - Too much exploitation can lead to getting stuck at a “wrong” solution. Other free particles may have better answers, but the swarm does not listen.
    - Too much exploration could lead to inconclusive outcome as particles are less likely to converge.
- Use the weights to balance exploitation and exploration:
    - Exploration: inertia and cognitive.
    - Exploitation: social.
    - Consider to treat the inertia weight in a similar sense to temperature in simulated annealing. Start with lots of inertia for exploration, and then gradually let social and cognitive weight to takeover for more exploitation.
- Does PSO converge to an optima? The answer is **NO GUARANTEE**.
- Improvement: the Adaptive PSO developed in 2009
    - Wrap PSO with an additional **Fuzzy + Elitist Learning** algorithm to adjust the weights for particles.
        - After running a round of PSO, compute the distances between all particle pairs, and compute the **evolutionary factor** for each particle.
        - The evolutionary factor captures how far away that a particle is from the whole swarm. More distant particle means more exploration than exploitation.
        - Classify particles into 4 groups according to the evolutionary factor, and use group-specific formula to reassign weights.
        - Perform elitist learning for the group classified as “Convergence”: Given the location of the particles, perform an algorithm similar to simulated annealing to update the locations for these particles.
        - Run PSO again with the reassigned weights.

        - Keep repeating PSO and learning until we run out of iterations.
    - Ensures convergence to at least a local optimal.
- PSO is computationally expensive as we need to compute $n$ times for each iteration.
- Either do **parallel computation** (say assign each particle to a core) or run on GPUs.

### 1.4 Nelder-Mead (just to mention real quick)

- Use vertices of a simplex to search for solution.
    - **Simplex:** the minimum measurable object in a space.
        - 1-d space: a simplex is a line formed by 2 points.
        - 2-d space: a simplex is a triangle formed by 3 points.
    - Form a simplex, evaluate function value for each of the vertices.
    - Compute the centroid of the simplex.
    - Use 1 out of 4 strategies to search for a new vertex according to how the performances of each vertices compared with the centroid.
    - The process gives us a new simplex pointing towards a supposedly better direction.
    - Repeat until the vertices of the simplex are close enough.
- Strategies includes **reflection**, **expansion**, **contraction** and **shrink**. See wikipedia: https://en.wikipedia.org/wiki/Nelder%E2%80%93Mead_method (https://en.wikipedia.org/wiki/Nelder%E2%80%93Mead_method)
- Widely used, and is almost always one of the default algorithms provided by various programming languages. For example, `optimize` in R use Nelder-Mead by default.
- The first few iterations are efficient in the sense that one quickly locate a rough area that possibly contains the optimal.
- Does NOT guarantee a convergence to optimal. One might find something neither a maximum nor a minimum.

## 2 Concluding Remarks

- No perfect algorithms! Totally entrusting an algorithm most likely wastes your time.
- My workflow:
    1. Do a grid search or PSO on the space. For example, if searching in a 8-d space I'd use 6 points on each dimension to form a $6^8$ grid for the grid search.
    2. Use the best or more results from the previous stage as initial points to perform simulated annealing.
    3. Use the best result from annealing as an initial point and check with gradient based method.
- Jobs to be done before using my workflow:
    - Experiment the algorithm you want with the function of your interest. But use a small amount of iteration only!
    - Helps you to get a quick grasp about your objective function, and about how you should specify the parameters in the algorithm.
    - Helps you to detect bugs or ill-behaving parts of your function. You don't want to see error messages after 8 hours of iterations, especially when the iteration is almost completed.

## 3 Assignments

- Consider a system of equations

$$I_i = \frac{\left(\frac{w_i}{\varphi_i}\right)^{1-\sigma}}{\left(\frac{w_i}{\varphi_i}\right)^{1-\sigma} + \left(\frac{w_j}{\varphi_j}\right)^{1-\sigma}}(I_i + I_j)$$

$$I_j = \frac{\left(\frac{w_j}{\varphi_j}\right)^{1-\sigma}}{\left(\frac{w_j}{\varphi_j}\right)^{1-\sigma} + \left(\frac{w_i}{\varphi_i}\right)^{1-\sigma}}(I_j + I_i)$$

$$0 = \frac{I_i}{\varphi_i}\left(\frac{\sigma}{\sigma-1}\right)^{-\sigma}\left[\left(\frac{w_i}{\varphi_i}\right)^{-\sigma} + \left(\frac{w_j}{\varphi_j}\right)^{-\sigma}\right] - 1$$

$$0 = \frac{I_j}{\varphi_j}\left(\frac{\sigma}{\sigma-1}\right)^{-\sigma}\left[\left(\frac{w_j}{\varphi_j}\right)^{-\sigma} + \left(\frac{w_i}{\varphi_i}\right)^{-\sigma}\right] - 1$$

Assume further that $\sigma = 3$, $\varphi_i = 1$ and $varphi_j = 5$. One can solve for $\{I_i, I_j, w_i, w_j\}$ from this system of equation by reformulating it into a minimization problem. Your mission is as follows:

1. Figure out how this equation system can be reformulated into a loss function so that we can choose the interested variables to minimize it.

**Hint:** Euclidean distance from a point $(x, y)$ to the origin $(0, 0)$ is defined as $d = \sqrt{x^2 + y^2}$.

2. Use simulated annealing to solve this problem.

3. Use PSO to solve this problem.

4. Compare your solutions. Also compare the iterations needed to get the solution.

## 4 Reference

- Zhan, Z-H., Zhang, J., Li, Y, and Chung, H.S-H. (2009). "Adaptive Particle Swarm Optimization," *IEEE Transactions on Systems, Man, and Cybernetics*, 39, 1362–1381.