# Genetic Algorithms

Jose M Sallan

7/3/2019

# Introducing genetic algorithms

# Genetic algorithms

Genetic algorithms are inspired by **evolution**:

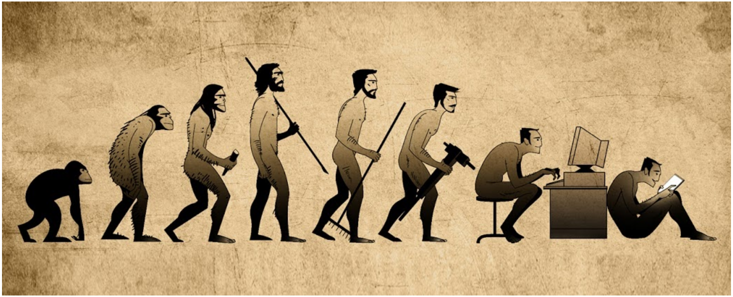


Figure 1: Evolution

# Evolution

Evolution is the change of heritable conditions of biological populations over successive generations:

- Heritable conditions are **encoded** in the **genotype**, and are displayed in the **phenotype**.
- Heritable conditions change by **variation** mechanisms: **crossover** of parent's genes and gene **mutation**.
- Only the individuals who fit to the environment are **selected** in the next generation.

Can we use the mechanism of evolution (variation and selection) to solve optimisation problems?

# Genetic algorithms and evolution

Genetic algorithm is a **optimisation metaheuristic** inspired in the process of natural selection and evolution.

- Metaheuristic: a template to create an algorithm to solve a specific problem.
- Other optimisation metaheuristics: particle swarm optimisation, simulated annealing, tabu search...

First introduced by John Holland (1975) to simulate evolutionary processes, later used for optimisation.

# The genetic algorithm flow

1. Define a **starting population** of candidate solutions
2. While there is no **convergence**, create a new generation:
   - Create a new generation member by **crossover** of members of previous generation.
   - **Mutate** generation members with a probability.
   - **Select** the members of the new generation, according to its **fitness**.

Frequently the selection process is embedded in crossover: only individuals fit to the environment are allowed to mate.

# The elements of a genetic algorithm

What do we need to build a genetic algorithm?

- A way to **encode** a candidate solution: how to obtain the genotype from the phenotype.
- Definition of **crossover**, **mutation** and **selection** operators. These operators work with the genotype of solutions.
- A **fitness function** that help us assess how does a population member fit to the environment.
- A **convergence** criterion to know when the algorithm stops.

# The standard genetic algorithm

# Binary encoding

A standard implementation of the genetic algorithm metaheuristic:

- A continuous **fitness function** to optimize (maximize or minimize) of $p$ variables, sometimes with **constraints**.
- A **region** to explore with the genetic algorithm, defined by an upper and lower bound of variables (additional constraints).
- A **bit string** representation of each variable:
  - Phenotype: a real number $x \in \mathcal{R}$.
  - Genotype: a binary string $\mathbf{b} \in \mathcal{B}^n$.
- Creation of new population with a **crossover** operator:
  - Uniform, one-point, two-point crossover
- **Proportional selection** of crossover elements:
  - The probability of selection is higher for elements of good value of fitness function.

# Bit string representation

Standard encoding mapping $\mathcal{B}^n \rightarrow [L, U] \subset \mathcal{R}$ of $n$ bits slices a continuous interval into $2^n - 1$ bins:

$$x = L + \frac{U - L}{2^n - 1} \sum_{i=0}^{n-i} b_{n-1} 2^i$$

For $n = 3$ bits we have eight marks of the [-10, 10] interval:

```
## [1] -10.00  -7.14  -4.29  -1.43   1.43   4.29   7.14  10.00
```

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Genotype | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| Phenotype | -10 | -7.14 | -4.29 | -1.43 | 1.43 | 4.29 | 7.14 | 10 |

# Bit string representation

The bit string has a precision equal to interval width:

$$\Delta x = \frac{U - L}{2^n - 1}$$

A high precision requires high number of bits, increasing computational cost.

# Bit string representation

**Hamming distance** of two bit strings: number of different bits.

The Hamming distance of 001100 and 100110 is 3: bits 1, 3 and 5 are different.

Problem: contiguous bit string can have high Hamming distance (change of one bit can alter the value of $x$):

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| — | 1 | 2 | 1 | 3 | 1 | 2 | 1 |

# Bit string representation

**Alternative:** reflected binary code or Gray code (after Frank Gray). Can be obtained from standard binary:

$g_1 = b_1$

$g_n = b_n \oplus b_{n-1}, \;\; i = 2, \ldots, n$

Gray encoding for $n = 3$

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **b** | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| **g** | 000 | 001 | 011 | 010 | 110 | 111 | 101 | 100 |

Two consecutive strings encoded in Gray coding have Hamming distance equal to one.

about Gray encoding

# Crossover operators

A crossover operator obtains one offspring from two or more elements of the population. Crossovers for binary strings:

Element 1: 001011101001 Element 2: 110000111000

Uniform crossover with mask (at random): 011000110110, son 010011111001

One-point crossover with cut at $i = 5$: 001010111000

Two-point crossover with cut at $i = 3$ and $i = 10$: 001000111001

# Crossover and selection

Usually we give a higher chance to mate to elements with good values of fitness function.

In **proportional selection** probability of mating is proportional to fitness function (in MAX problems)

In **tournament selection** we pick the best-fitting individual from a subset of elements selected at random.

# Mutation operators

The offspring obtained from crossover can be mutated with a probability $p_m$. Objective: increasing variation and avoid convergence in a local optimum.

In binary encodings consists in negating some elements of the bit string.

Element to mutate: 001010111000

**Inorder mutation** (e.g., three elements from $i = 3$):

00**101**0111000

00**010**0111000

**Random mutation** (e.g., three positions):

00**1**010**1**110**0**0

00**0**010**0**110**1**0

# Elitism

We want to keep the best result obtained in all previous generations, as it is the output of the algorithm.

Adopting an **elitist strategy** means including this element in the next generation.

# Convergence

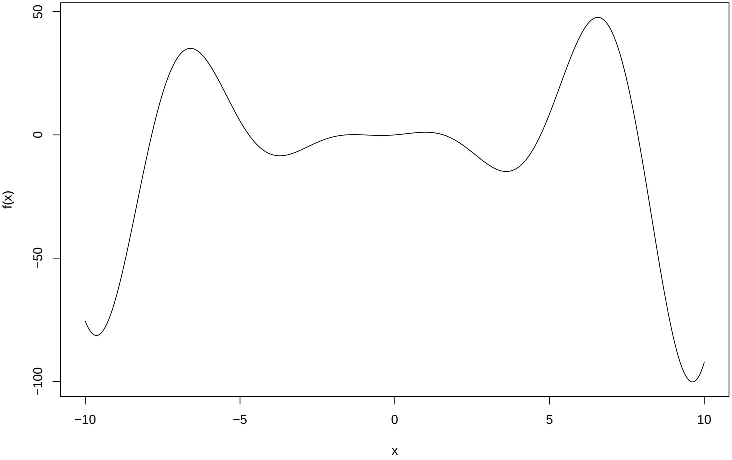
Some criteria for convergence (stopping):

- The algorithm reaches a specified number of generations.
- There is no improvement of the fitness function in a number of iterations.
- There is no variation in the last generation (all elements are equal).

# Solving a standard GA

# A simple example

We want to obtain the maximum in $-10 \leq x \leq 10$ of $(x^2 - x)cos(x)$:




Let's use a binary encoding implementation.

# The GA R package

We will use the GA package of R, written by Luca Scrucca

Installing R and RStudio on Windows

Installing R and RStudio on mac

# Defining the fitness function (binary encoding)

As the genetic algorithm works with binary encoding, we need to provide a fitness function with binary input:

```
bin2real <- function(b, l, u){
  n <- length(b)
  s <- 0
  for(i in 0:(n-1)) s <- s + b[n-i]*2^i
  s <- l + (u-l)*s/(2^n - 1)
  return(s)
}
f.stdbin <- function(b, lbound=-10, ubound=10){
  x <- bin2real(b, lbound, ubound)
  return((x^2+x)*cos(x))
}
```

# Applying genetic algorithm (binary encoding)

First time you use this you need to install GA package:

```
install.packages("GA")
```

Perform GA and store results in `g01.stdbinary` variable:

```
library(GA)
g01.stdbinary <- ga(type = "binary", fitness = f.stdbin,
maxiter=50, lbound=-10, ubound=10, nBits=32, seed=1313)
```

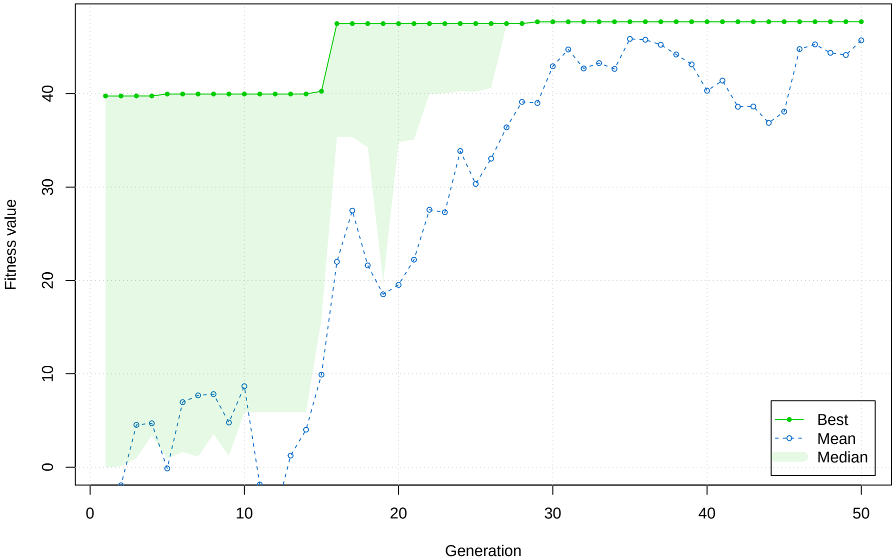
# Results (binary encodign)

```r
summary(g01.stdbinary)
```

```
## -- Genetic Algorithm -------------------
##
## GA settings:
## Type                  =  binary
## Population size       =  50
## Number of generations =  50
## Elitism               =  2
## Crossover probability =  0.8
## Mutation probability  =  0.1
##
## GA results:
## Iterations             = 50
## Fitness function value = 47.70562
## Solution =
##      x1 x2 x3 x4 x5 x6 x7 x8 x9 x10  ...  x31 x32
```

# Performance (binary encoding)

```
plot(g01.stdbinary)
```



# Define fitness function (Gray encoding)

We need an additional function to convert from Grey to binary:

```
Gray2bin <- function(g){
  n <- length(g)
  b <- logical(n)
  b[1] <- g[1]
  for(i in 2:n) b[i] <- ifelse(g[i]==0, b[i-1], !b[i-1])
  return(b)
}
f.gray <- function(g, lbound=-10, ubound=10){
  b <- Gray2bin(g)
  x <- bin2real(b, lbound, ubound)
  return((x^2+x)*cos(x))
}
```

# Genetic algorithm (Gray encoding)

```
g01.gray <- ga(type = "binary", fitness = f.gray,
maxiter=50, lbound=-10, ubound=10, nBits=32, seed=1313)
```

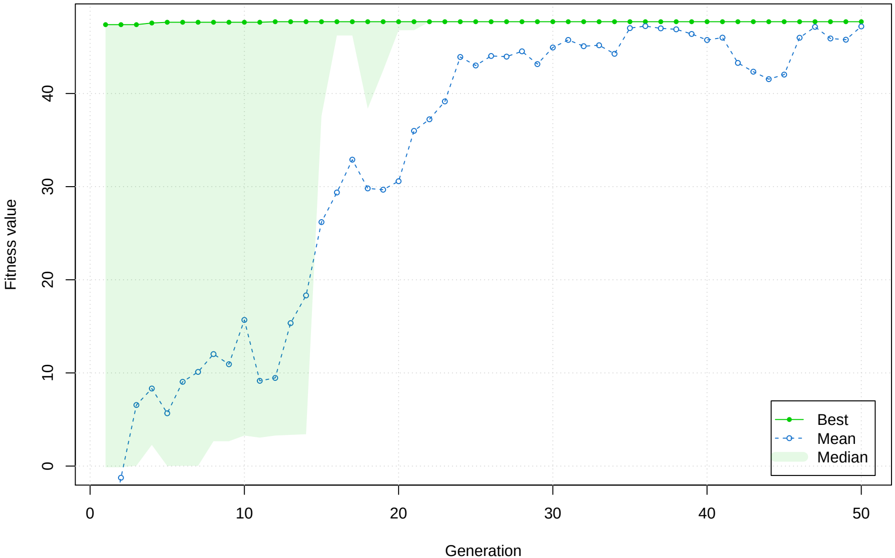
# Results (Gray encoding)

```r
summary(g01.gray)
```

```
## -- Genetic Algorithm -------------------
##
## GA settings:
## Type                  =  binary
## Population size       =  50
## Number of generations =  50
## Elitism               =  2
## Crossover probability =  0.8
## Mutation probability  =  0.1
##
## GA results:
## Iterations             = 50
## Fitness function value = 47.70562
## Solution =
##      x1 x2 x3 x4 x5 x6 x7 x8 x9 x10  ...  x31 x32
```

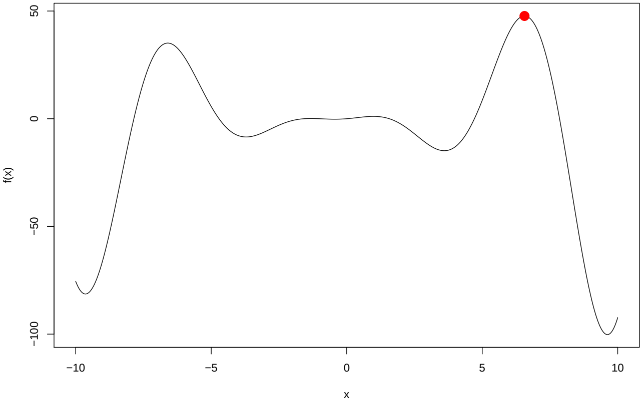
# Performance (Gray encoding)

```
plot(g01.gray)
```



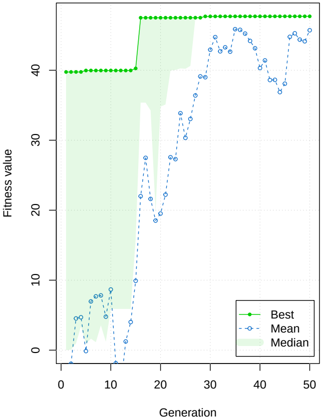
# Conclusions of binary vs Gray comparison

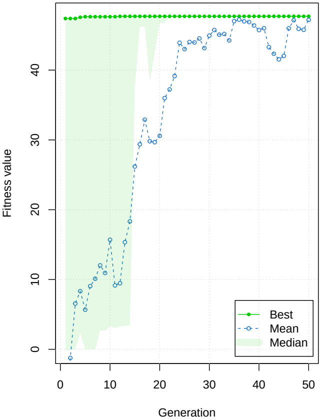
Both algorithms reach optimum...



# Conclusions of binary vs Gray comparison

. . . but Gray (right) is faster than standard binary (left)



# Real-valued encoding

# Real-valued encoding

For problems having decision variables $\mathbf{x} \in \mathcal{R}^n$, the most natural encoding is the floating-point or real-valued encoding.

In this encoding, the genotype is simply the phenotype (that is, the vector itself).

# Crossover operators

The most used crossover operators are:

- Vector-level:
  - Whole arithmetic crossover
- Variable-level:
  - Local arithmetic crossover
  - Blend crossover
  - Uniform crossover

# Crossover operators

**Whole arithmetic crossover:** from two parents $\mathbf{x}^1$ and $\mathbf{x}^2$ we can obtain two offspring:

$\alpha\mathbf{x}^1 + (1-\alpha)\mathbf{x}^2$

$(1-\alpha)\mathbf{x}^1 + \alpha\mathbf{x}^2$

with $\alpha \in [0,1]$

**Local arithmetic crossover:** we perform a similar crossover at the variable level.

$\alpha x_i^1 + (1-\alpha)x_i^2$

$(1-\alpha)x_i^1 + \alpha x_i^2$

again with $\alpha \in [0,1]$

# Crossover operators

**Blend crossover:** we construct the offspring selecting each variable randomly from the interval:

$[x_i^1 - \alpha(x_i^2 - x_i^1), x_2^1 + \alpha(x_i^2 - x_i^1)]$

with $x_i^2 > x_i^1$.

Usually $\alpha = 0.5$ yields good results. If necessary, variables of offspring should be adjusted to upper or lower bounds.

If $\alpha = 0$ we have **uniform crossover**.

# Mutation operators

The most usual mutation operator is to pick a value within a given radius of the population member.

It is frequent to reduce the radius as generations go (nonuniform mutation), similarly to genetic algorithm.

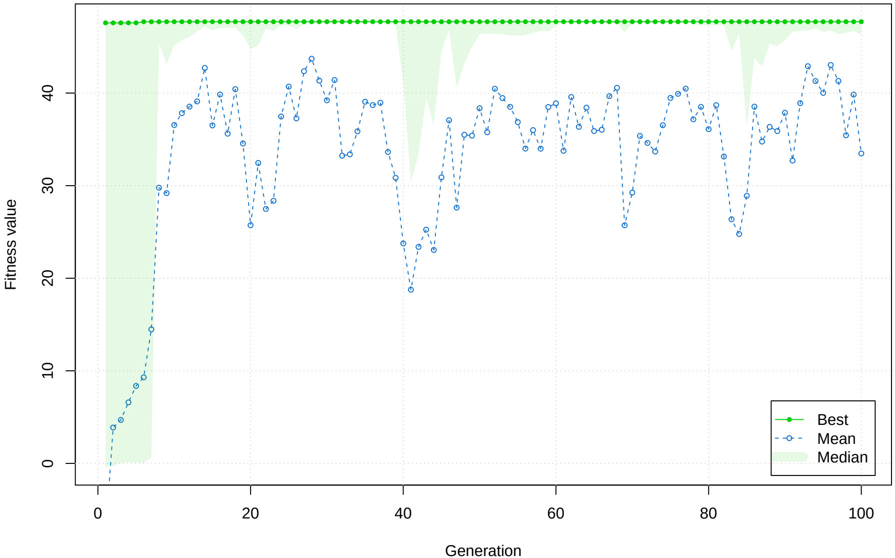
# Genetic algorithm (real-valued encoding)

```
f <- function(x)  (x^2+x)*cos(x)
lbound <- -10; ubound <- 10
g01.real <- ga(type = "real-valued", fitness = f,
lower = lbound, upper = ubound, seed=1313)
```

This implementation also finds the optimum.

# Performance (real-valued encoding)

```
plot(g01.real)
```



# The Rastrigin function

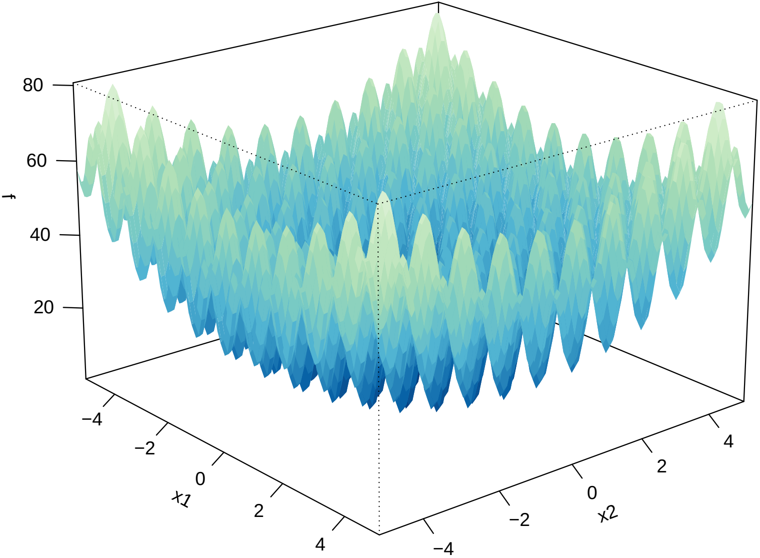
The **Rastrigin function** is a performance test for optimisation algorithms, as it has a large search space and many local minima. For two variables:

$f = 20 + x_1^2 + x_2^2 - 10(cos 2\pi x_1 + cos 2\pi x_2)$

with $x_i \in [-5.12, 5.12]$

Note that this is a minimization problem, so we must use $-f$ as fitness function.

# The Rastrigin function



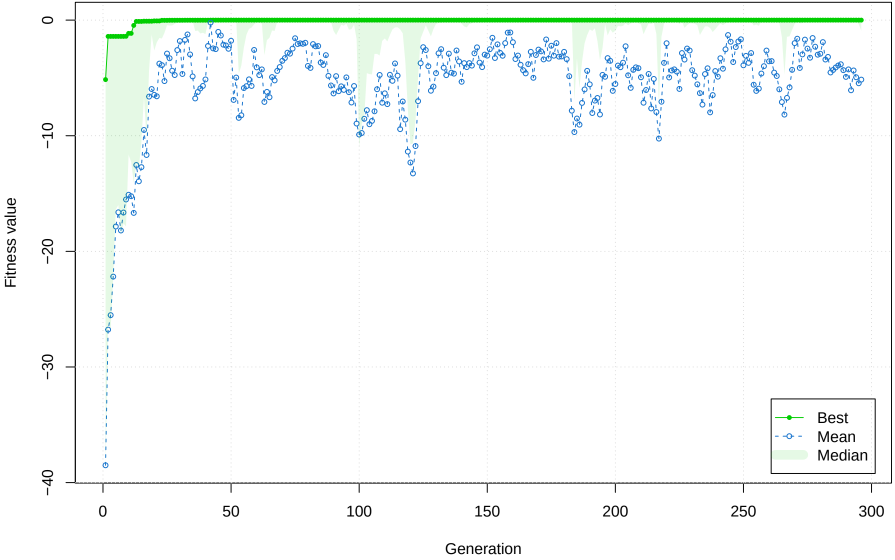
# Solving the Rastrigin problem

```
ga.rastrigin <- ga(type = "real-valued",
         fitness =  function(x) -Rastrigin(x[1], x[2]),
         lower = c(-5.12, -5.12), upper = c(5.12, 5.12),
         popSize = 50, maxiter = 1000, run = 100, seed=1313)
```

Set two convergence criteria: maximum number of generations (1000) and maximum number of runs without improvement (100).

The GA finds the optimum $(0, 0)$.

# Performance (Rastrigin problem)



# Using MATLAB for genetic algorithms

# Solving optimisation problems with ga in MATLAB

You can use MATLAB to solve optimisation (minimisation) problems with genetic algorithm using the **ga** solver.

To use the **ga** solver, you need to install the **Global optimisation Toolbox**, an extension of the Optimisation Toolbox.

# Solving optimisation problems with ga in MATLAB

The generic structure of the **ga** function in MATLAB is:

```
x = ga(fun,nvars,A,b,Aeq,beq,lb,ub,nonlcon,options)
```

with:

- **fun:** function to minimize (required).
- **nvars:** number of variables (required).
- **A, b:** linear inequalities $Ax \leq b$.
- **Aeq, beq:** linear equalities $Ax = b$.
- **lb, ub:** low and upper bounds.
- **nonlcon:** function returning nonlinear constraints, being $c(x) \leq 0$ or $ceq(x) = 0$.
- **options:** a set of options for the ga created with **optimoptions**.

# Solving optimisation problems with ga in MATLAB

The output of the **ga** can be of the form:

```
[x,fval,exitflag,output,population,scores] = ga(...)
```

with:

- **x:** value of the solution.
- **fval:** value of the function to optimize.
- **exitflag:** identifier of the reason the algorithm stopped.
- **output:** information about algorithm performance.
- **population, scores:** matrix with the final population and scores vecctor of that final population.

# Solving optimisation problems with ga in MATLAB

To create the **options** variable we use the **optimoptions** function:

```
options = optimoptions('ga','Param1', value1,
                       'Param2', value2, ...);
```

To change population size:

```
options = optimoptions('ga', 'PopulationSize', 100)
```

To get an interactive plot of algorithm evolution:

```
options = optimoptions('ga', 'PlotFcn', 'gaplotbestf')
```

# Solving optimisation problems with ga in MATLAB

You can find reference of genetic algorithms in MATLAB in the following links:

**ga** function

options

# Minimising the Rastrigin function with MATLAB

In MATLAB we can access the Rastrigin function with **@rastriginsfcn**

**Exercise:** find the (unrestricted) minimum of the Rastrigin function with genetic algorithm in MATLAB with the following parameters:

- Maximum number of generations: 1000.
- Maximum number of runs without improvement (stalls): 300.
- Plot the evolution of best value and mean value of generation **gaplotbestf**.

# Solving a maximisation problem

Reconsider the problem of maximising $(x^2 - x)cos(x)$ in the interval $-10 \leq x \leq 10$.

Write a script with the fitness function and other script with the specification of the **ga** function, which plots algorithm evolution.

# Constrained optimisation

# Constrained optimisation

We want to minimise the function:

$f = 100(x_1^2 - x_2^2)^2 + (1 - x_1^2)^2$

subject to the following constraints and bounds:

$x_1 x_2 + x_1 - x_2 + 1.5 \leq 0$

$10 - x_1 x_2 \leq 0$

$0 \leq x_1 \leq 1$

$0 \leq x_2 \leq 13$

# Constrained optimisation

Crossover and mutation operators may generate population members that do not satisfy constraints (although in general they discard solutions out of variable bounds).

The way to discard these solutions is to generate a **fitness function with penalties**: solutions that do not satisfy constraints have bad values of fitness function.

Note that fitness function != objective function.

# Constrained optimisation

The fitness function to our problem will be (both constraints are $\leq$ inequalities) minimising:

$F = f + Mf_1 + Mf_2$

Where:

$f_1 = \text{MAX}(x_1 x_2 + x_1 - x_2 + 1.5, 0)$

$f_2 = \text{MAX}(10 - x_1 x_2, 0)$

# Solving constrained optimisation with MATLAB

We need to create two scripts with files representing objective function and constraints:

```
function y = cam_function (x)
y = 100*(x(1)^2 - x(2))^2 + (1 - x(1))^2;
```

```
function [c, ceq]=cam_constraints (x)
c(1) = x(1)*x(2) + x(1) - x(2) + 1.5;
c(2) = 10 - x(1)*x(2);
ceq = [];
end
```

# Solving constrained optimisation with MATLAB

Then we solve the problem doing:

```
lb = [0; 0];
ub = [1; 13];
nonlcon = @cam_constraints;
fun = @cam_function;
options = optimoptions('ga',
'ConstraintTolerance', 1e-6, 'PlotFcn', 'gaplotbestf')
[x, fval, exitflag, output] =
ga(fun, 2, [],[],[],[], lb, ub, nonlcon, options)
```

You will find the scripts in the **cam** folder in the **examples** folder.

# Performance of the R implementation