Genetic Algorithms

Jose M Sallan

20/11/2019

Introducing genetic algorithms

Genetic algorithms

Genetic algorithms are inspired by **evolution**:

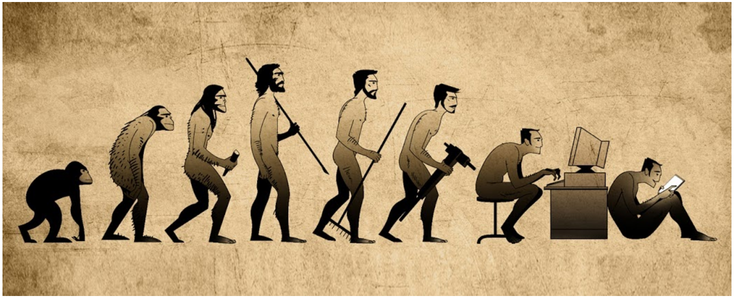


Figure 1: Evolution

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.
  - **Binary encoding:** a bit string.
  - **Real value encoding:** the value itself.
  - **Permutative encoding:** a permutation of n elements.
- 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:

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

```r
renderBit <- 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)
}
```

Bit string representation

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

```
b <- list(c(0,0,0), c(0,0,1), c(0,1,0), c(0,1,1), c(1,0,0), c(
x <- sapply(b, function(x) renderBit(x, -10, 10))
print(x, digits=3)
```

```
## [1] -10.00  -7.14  -4.29  -1.43   1.43   4.29   7.14  10.0
```

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:

$g_1 = b_1$

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

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 **rank selection** probability of mating is proportional to rank (from worse to best).

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

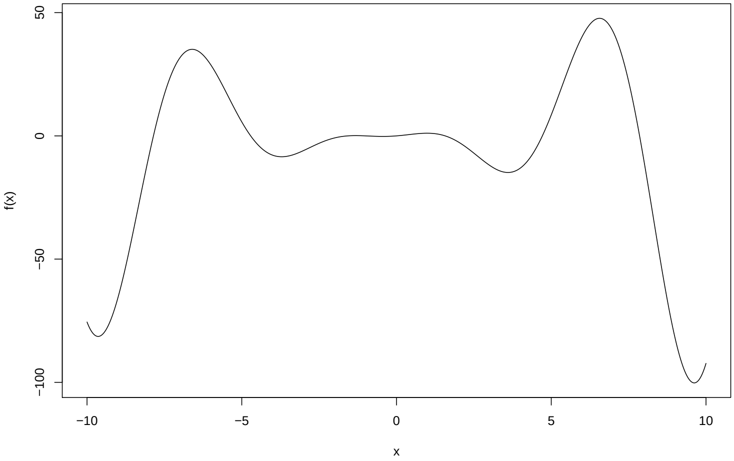
Some criteria for convergence (stopping):

- 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 GA using R

A simple example

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



We will solve this problem using the GA package of R, written by Luca Scrucca, usign binary encoding.

Defining the fitness function (binary encoding)

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

```r
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

To obtain the solution, we use `renderBit` function with the solution:

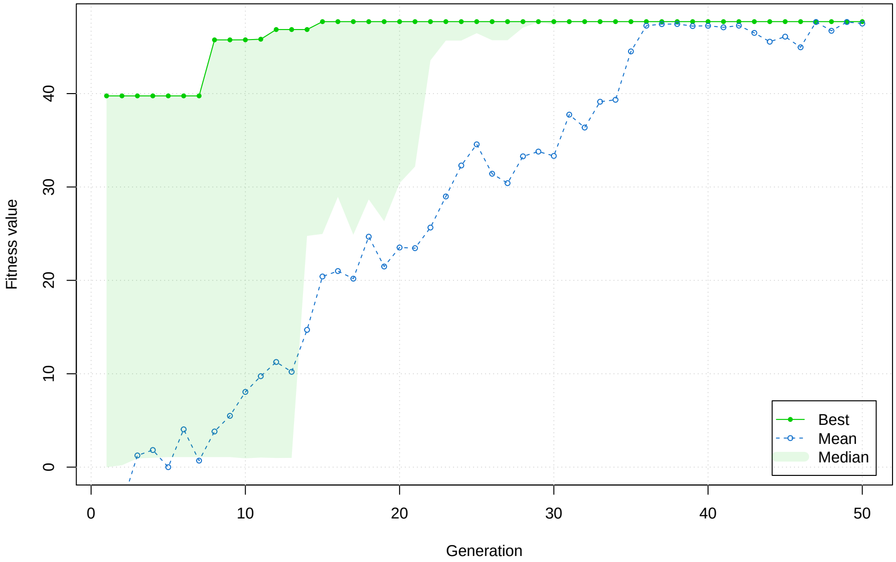
```
renderBit(g01.stdbinary@solution, -10, 10)
```

```
## [1] 6.560535
```

```
g01.stdbinary@fitnessValue
```

```
## [1] 47.70562
```

Performance (binary encoding)



Define fitness function (Gray encoding)

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

```r
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)

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

Results (Gray encoding)

```
solbin <- Gray2bin(g01.gray@solution)
renderBit(solbin, -10, 10)
```

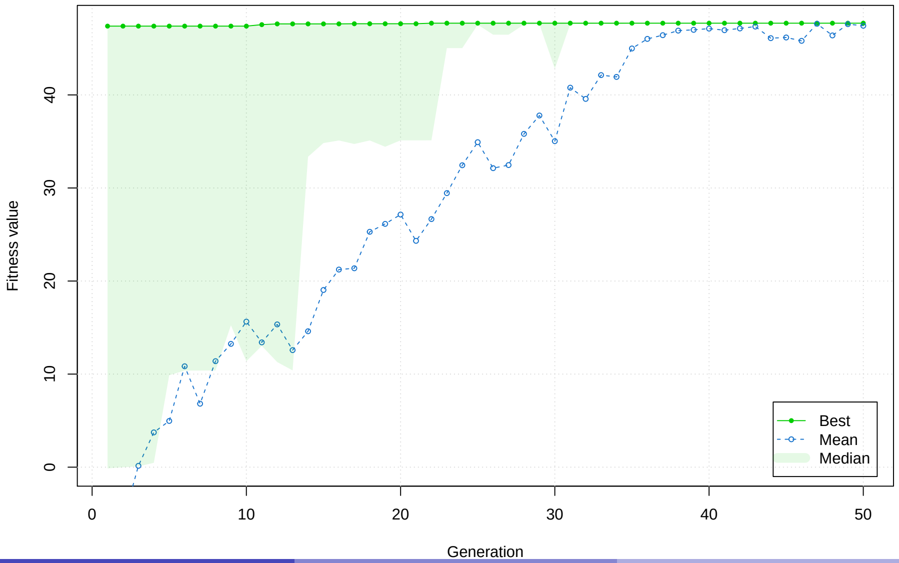
```
## [1] 6.560448
```

```
g01.gray@fitnessValue
```

```
## [1] 47.70562
```

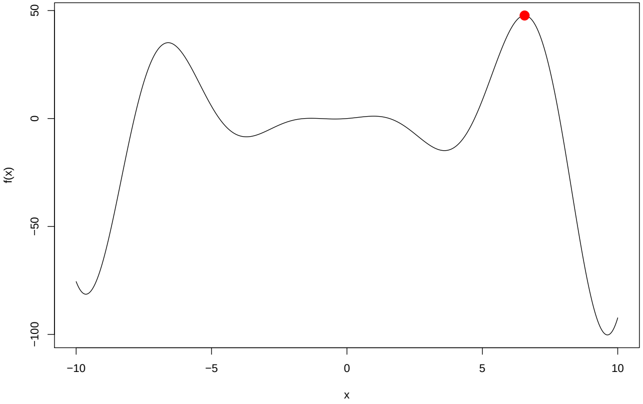
Performance (Gray encoding)

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



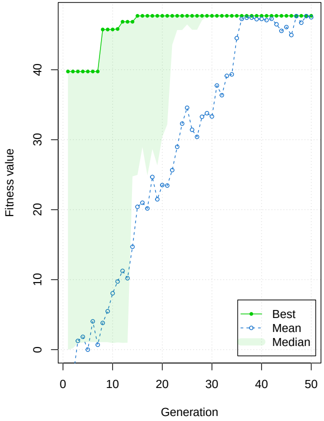
Conclusions of binary vs Gray comparison

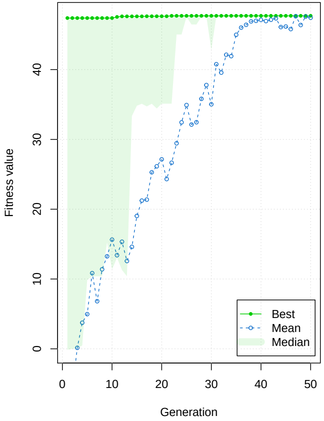
Both algorithms reach optimum. . .



Conclusions of binary vs Gray comparison

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





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

Constrained optimisation

Constrained optimisation

We want to minimise the function:

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

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:

$$f_1 = \mathrm{MAX}(x_1x_2 + x_1 - x_2 + 1.5, 0)$$

$$f_2 = \mathrm{MAX}(10 - x_1x_2, 0)$$

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
```

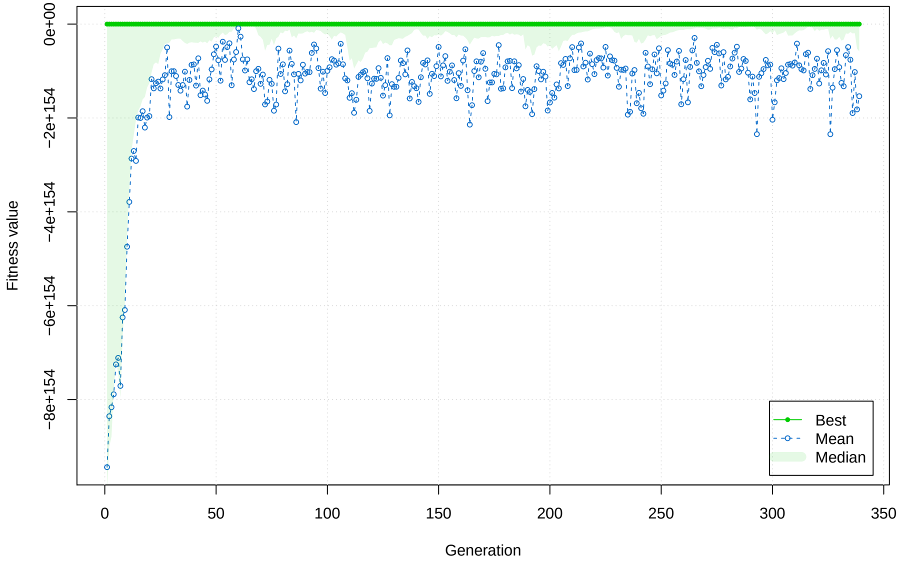
Solving constrained optimisation with MATLAB

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



Influence of parameters and operators on performance

Influence of parameters and operators on performance

If defaults do not work well, we can try to improve performance modifying genetic algorithm **parameters**:

- Population parameters: starting population, size
- Mutation and probability of performing mutation

We can also try to explore out of the default **operators**:

- selection
- crossover
- mutation
- elitism
- convergence

Population

The population helps the **exploration** of the search space:

- **Small population:** small search space, low time complexity, may need more generations to converge.
- **Large population:** widens search space, high time complexity, may need less generations to converge.

Seeding of initial population: including good solutions (e.g., obtained by local search).

Mutation

Mutation operator increases exploration: if we only perform crossover the genetic algorithm converges fast.

We can test variations on the probability of applying mutation p_{mut}:

- Raise p_{mut} in small population sizes.
- Define a p_{mut} for each population member: individuals with bad fitness function should have higher probability of mutation.
- Reduce p_{mut} as number of generations increases. This is how the standard MATLAB mutation function **@mutationgaussian** works.

Evaluation of operators can be performed with **experimental design**: see vignette **KPwithGApackage**. Some of this implementations are not supported by stanadard R or MATLAB functions.

Genetic algorithm extensions

Hybrid genetic algorithms

Consist of combining a **genetic algorithm** with a **local search** optimization algorithm.

Receives several names in literature:

- Hybrid genetic algorithms
- Memetic algorithms
- Genetic local search
- Cultural algorithms

Hybrid genetic algorithms implementation

The local search to use may depend on the function to optimize:

- MATLAB (real-valued representation):
  - **@fminunc** for unconstrained optimization (e.g., quasi Newton).
  - **@fmincon** for constrainted optimization (e.g., interior point).
- Simulated annealing (SA).
- Tabu search (SA).

SA and TS are available for binary encoding and permutative representations (e.g., flowshop or tabu search problems).

Hybrid genetic algorithms implementation

There are several ways of combining genetic algorithms and local search:

- **The MATLAB way:** the local search function runs after the genetic algorithm termination.
- Performing a local search optimization to all the population every fixed number of generations.
- Performing local search with a probability that can depend on fitness (optimize the best individuals).

Hybrid genetic algorithms in MATLAB

See scripts in the documentation:

- **rosenbrock.m**
- **rosenbrock_constraints2.m**
- **rosenbrock_ga2.m**
- **rosenbrock_fmincon.m**

Islands model genetic algorithm

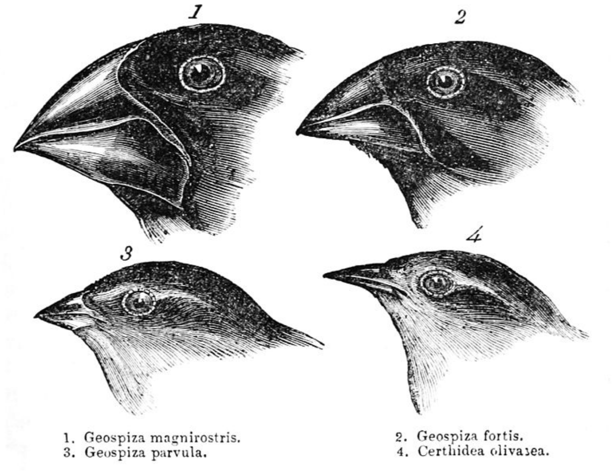


Figure 2: Darwin finches

Islands model genetic algorithm

Instead of having one single population, the islands model maintains **several populations (islands)**.

Every specific number of generations, **migration** takes place: each population is seeded with individuals of other populations.

This model can be adequate to foster variation among populations with good solutions obtained in other exploitation processes.

Islands model genetic algorithm

Other names that islands model has:

- multi-island model.
- migration model.

This model can be adequate for parallel computing.

Islands model genetic algorithm

Currently the islands model is not implemented in MATLAB: although there are some **Migration** options available, there is no advantage in using them.

The **GA** package in R supports islands model and parallel computing. See **Schwefel** vignette for an example.

Conclusions

Conclusions

The genetic algorithm is a metaheuristic useful in many optimitzation contexts:

- Real-valued functions (real-value or binary (tradicional) encodings).
- Binary encodings.
- Permutative encodings (flowshop problem, travelling salesman problem).

Works with **unconstrained** and **constrained** optimization.

Genetic algorithms are a **brute-force** method: use it when you want high-quality solutions even with high computational cost (number of function evaluations, time spent).

In MATLAB, implemented in the **ga** solver.

For real-valued functions, try first **fminunc** or **fmincon** solvers.

Conclusions

If algorithm performance is not good, you might want to go beyond solver defaults:

- Increase population size (sometimes effective, but costly).
- Increase mutation rate.

Try different operators:

- Selection
- Crossover
- Mutation
- Elitism
- Convergence

Try alternative implementations:

- Hybrid genetic algorithms
- Islands model of genetic algorithms

Thanks for your attention!

Contact:

Jose M Sallan mailto:jose.maria.sallan@upc.edu

Materials available at:

https://github.com/jmsallan/optimization