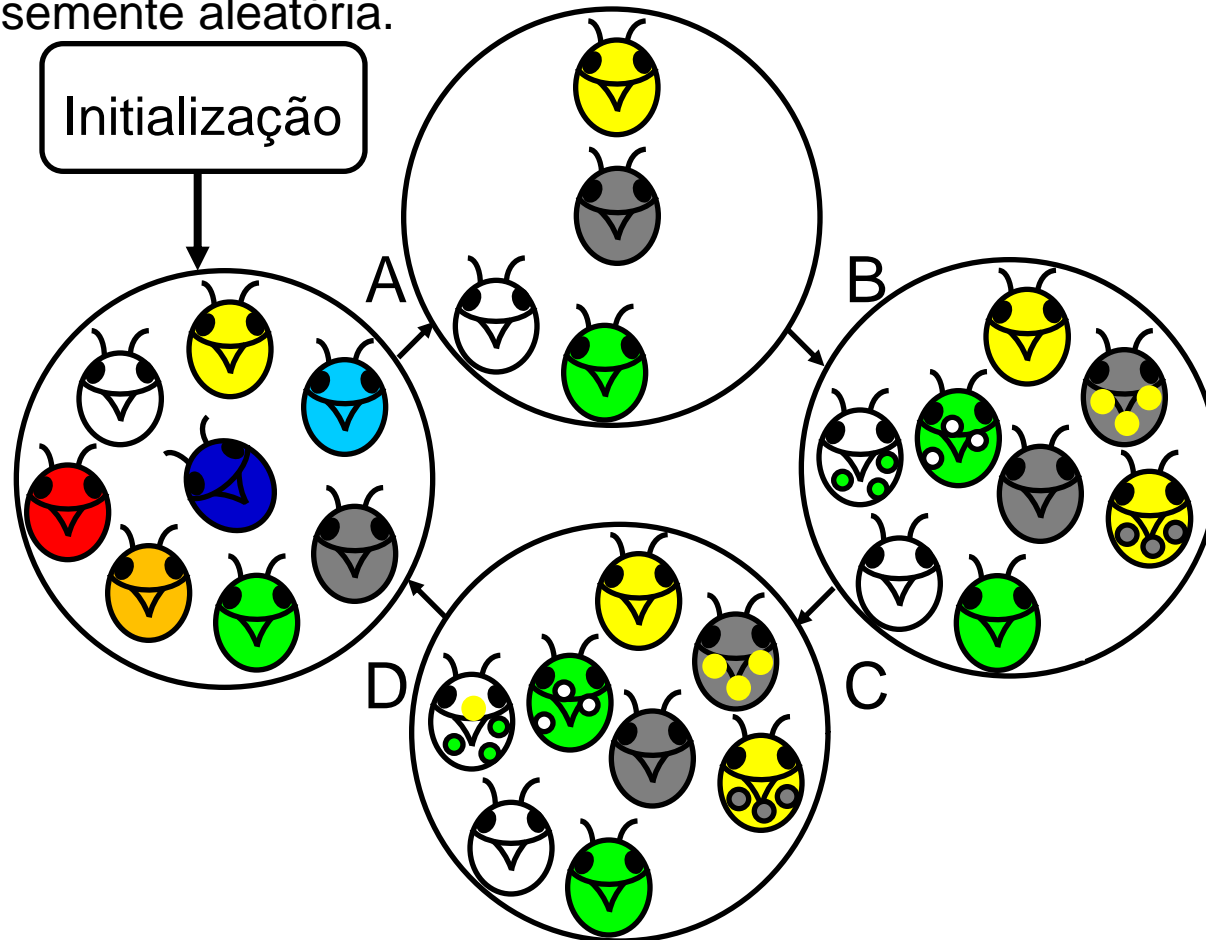




No algoritmo genético, a geração aleatória de indivíduos de uma dada população é uma etapa de importância fundamental. Esse processo é chamado de inicialização na figura abaixo. Com a biblioteca *NumPy*, temos a possibilidade de fixarmos uma semente aleatória, para garantir que os números aleatórios iniciais sejam mantidos, para uma dada semente aleatória.



Para definição da semente aleatória, usamos o comando abaixo. Nesse exemplo a semente aleatória é um número inteiro atribuído à variável `seed_in`.

```
np.random.seed(seed_in)
```

Essa linha deve ser inserida antes da geração do número aleatório. Por exemplo, o código abaixo (`random_with_seed.py`) gera um número aleatório entre 0 e 9.

```
# Import library
import numpy as np
# Set up seed
np.random.seed(31415)
# Generate random number
rs = np.random.randint(0,9)
print(rs)
```

Veja que a definição da semente aleatória aparece antes da definição do número aleatório, que ocorre na linha `rs = np.random.randint(0,9)`.

O código *random\_loop.py* repete a geração de número aleatório dentro de um loop *for*, como mostrado abaixo.

```
# Import library
import numpy as np
# Loop for generation of random numbers
for i in range(10):
    # Set up seed
    np.random.seed(31415)
    # Generate random number
    rs = np.random.randint(0,9)
print(rs)
```

Abaixo temos o resultado obtido com o programa *random\_loop.py*. Anterior a geração do número aleatório em cada iteração, temos a fixação da semente aleatória. O resultado é que temos sempre o mesmo número gerado.

Colocando-se a linha da semente como comentário no código *random\_loop.py*, os números passam a variar.

```
# Import library
import numpy as np
# Loop for generation of random numbers
for i in range(10):
    # Set up seed
    # np.random.seed(31415)
    # Generate random number
    rs = np.random.randint(0,9)
print(rs)
```

Como não temos semente aleatória, o código gera números distintos, como podemos ver no resultado abaixo.

```
0
1
3
2
3
3
1
6
8
4
```

Abaixo temos o código do novo método para gerar strings binárias com semente. A principal novidade é que temos um atributo do objeto para a semente, atribuído à variável `self.seed_in`. Essa variável é usada na linha `np.random.seed(self.seed_in)`.

```
def gen_bin_string_seed(self):
    """Method to generate binary strings"""
    # Import library
    import numpy as np
    # Set up seed for random number
    np.random.seed(self.seed_in)
    # Set up arrays of zeros
    self.max_array = np.zeros(self.max_iter)
    self.mean_array = np.zeros(self.max_iter)
    # Set up empty list
    bin_str_list = []
    # Generate binary string
    for i in range(self.pop_size):
        bin1 = np.random.randint(2, size=self.len_str)      # Binary (2) strings
        bin_str0 = str(bin1)
        bin_str0 = bin_str0.replace(" ", "")
        bin_str0 = bin_str0.replace("[", "")
        bin_str0 = bin_str0.replace("]", "")
        bin_str_list.append(bin_str0)
    # Update population
    self.current_pop = bin_str_list
    print("\nInitial Population")
    # Looping through initial population
    for bin_string in self.current_pop:
        print(bin_string)
```

Uma vez estabelecida a semente, os números aleatórios serão mantidos. No final do método temos um loop *for* para exibir a população inicial. O método construtor foi modificado, de forma a acomodar o atributo *self.seed\_in*.

```
def gen_bin_string_seed(self):
    """Method to generate binary strings"""
    # Import library
    import numpy as np
    # Set up seed for random number
    np.random.seed(self.seed_in)
    # Set up arrays of zeros
    self.max_array = np.zeros(self.max_iter)
    self.mean_array = np.zeros(self.max_iter)
    # Set up empty list
    bin_str_list = []
    # Generate binary string
    for i in range(self.pop_size):
        bin1 = np.random.randint(2, size=self.len_str)      # Binary (2) strings
        bin_str0 = str(bin1)
        bin_str0 = bin_str0.replace(" ", "")
        bin_str0 = bin_str0.replace("[", "")
        bin_str0 = bin_str0.replace("]", "")
        bin_str_list.append(bin_str0)
    # Update population
    self.current_pop = bin_str_list
    print("\nInitial Population")
    # Looping through initial population
    for bin_string in self.current_pop:
        print(bin_string)
```

Abaixo temos o trecho inicial da definição da classe com destaque para o método construtor. Vemos em vermelho o texto que foi adicionado à classe *Simple\_GA02()* para a semente aleatória. Também foram adicionados atributos para os valores mínimo e máximo da faixa de *float* (*self.f\_min* e *self.f\_max*). Outro atributo adicionado refere-se ao fator de conversão para o tamanho das tartarugas. Esses atributos serão discutidos nos próximos slides.

```
class Simple_GA02(object):
    """A class to implement a simple GA"""

    # Constructor method
    def __init__(self, pop_size, max_iter, p_cross, len_str, p_mut, f_min, f_max, conv_f, seed_in):

        # Define attributes
        self.pop_size = pop_size                # Population size
        self.max_iter = max_iter                # Number of iterations
        self.p_cross = p_cross                  # Probability of crossover
        self.len_str = len_str                  # String length
        self.p_mut = p_mut                      # Probability of mutation_Coley_algo_03
        self.f_min = f_min                      # Minimum float
        self.f_max = f_max                      # Maximum float
        self.conv_f = conv_f                    # Conversion factor
        self.seed_in = seed_in                  # Seed for generation of random numbers
```



Nas últimas aulas implementamos o algoritmo 1 de Colley 1999 que acha o máximo da função  $x^2$  entre os inteiros 0 e 4095. Muitos problemas de otimização não se restringe-se ao conjunto dos inteiros. Para apresentarmos a solução do problema de maximização, vamos considerar que a faixa de números está entre 1,25 e 3,14. Os novos números são chamados de *floats* mínimo e máximo e são representados pelas variáveis  $f_{min}$  e  $f_{max}$ .

String binária	Integers	Floats
00000000000000	0	1.25
00000000000001	1	
00000000000010	2	
00000000000011	3	
00000000000100	4	
00000000000101	5	
00000000000110	6	
00000000000111	7	
0000000001000	8	
.....	.....	
1111111111110	4094	
1111111111111	4095	3.14

De uma forma geral, podemos dizer que para cada inteiro temos um equivalente *float*, colocando em forma matemática temos:

$$f_{\min} = \alpha_0 + \alpha_1 i_{\min}$$

$$f_{\max} = \alpha_0 + \alpha_1 i_{\max}$$

Onde os *f*'s representam os *floats* e os *i*'s representam os inteiros. As constantes  $\alpha$ 's são usadas para convertermos de inteiro para *float*. Considerando-se que  $i_{\min} = 0$  temos:

$$f_{\min} = \alpha_0$$

Calculando-se  $f_{\max} - f_{\min}$  chegamos:

$$f_{\max} - f_{\min} = \alpha_0 + \alpha_1 i_{\max} - \alpha_0$$

$$f_{\max} - f_{\min} = \alpha_1 i_{\max} \Rightarrow \alpha_1 = \frac{f_{\max} - f_{\min}}{i_{\max}}$$

Assim, o *float* equivalente ( $f$ ) a um inteiro qualquer  $i$  é dado por:

$$f = \alpha_0 + \alpha_1 i = f_{\min} + \alpha_1 i = f_{\min} + \left( \frac{f_{\max} - f_{\min}}{i_{\max}} \right) i$$

Agora temos uma equação geral onde podemos obter qualquer *float*, a partir do conhecimento da faixa de inteiros que dispomos e da definição da faixa de *floats* que queremos analisar.

$$f = f_{\min} + \left( \frac{f_{\max} - f_{\min}}{i_{\max}} \right) i$$

Considerando-se uma string binária de  $l$  bits, o número inteiro máximo que pode ser representado é dado por:

$$i_{\max} = 2^l - 1$$

Assim, substituindo-se esse valor na equação do *float*, temos:

$$f = f_{\min} + \left( \frac{f_{\max} - f_{\min}}{2^l - 1} \right) i$$

Um aspecto importante na aplicação de algoritmos genéticos é a acurácia. Por exemplo, quando consideramos a equivalência entre a faixa de 0 a 4095 com a faixa de 1,25 a 3,14. Vemos que entre dois números consecutivos em decimal, por exemplo, 1000 e 1001, temos os equivalentes em *float*:

$$f = 1,25 + \left( \frac{3,14 - 1,25}{4095} \right) 1000 = 1,71153846$$

$$f = 1,25 + \left( \frac{3,14 - 1,25}{4095} \right) 1001 = 1,71200000$$

Assim, entre 1,7115 e 1,7120 não temos representação *float*. Ou seja, temos aproximadamente 0,0005 de acurácia. Uma forma de aumentarmos a acurácia, é aumentando-se o tamanho das strings binárias. Uma forma de calcularmos a acurácia é dividirmos a faixa de *floats* ( $f_{max} - f_{min}$ ) pela quantidade de inteiros ( $i_{max}$ ).

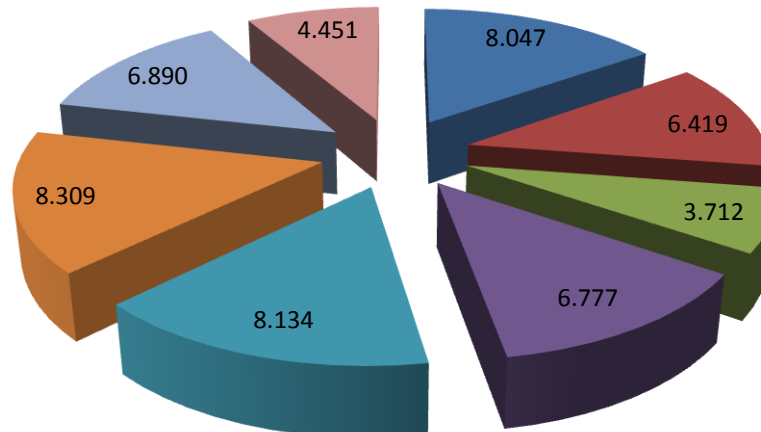
No novo código, criamos um método para implementar a equação de conversão de inteiro para *float*. Esse método é chamado *int2float()*. O código está destacado abaixo. Os parâmetros do método trazem o *float* mínimo (*f\_min*), o *float* máximo (*f\_max*) e o inteiro a ser convertido (*i*). Esses valores são usados para o cálculo do *f* que retorna para onde o método foi invocado, dentro do método *coley\_algo\_03()*.

```
def int2float(self,i):  
    """Method to convert integer to float"""  
  
    f = self.f_min + i*(self.f_max - self.f_min)/(2**(self.len_str) -1)  
  
    return f
```

Outra modificação foi na função *show\_pop()*, que passa a mostrar os valores *floats* para o indivíduo e a função ajuste, além da string binária.

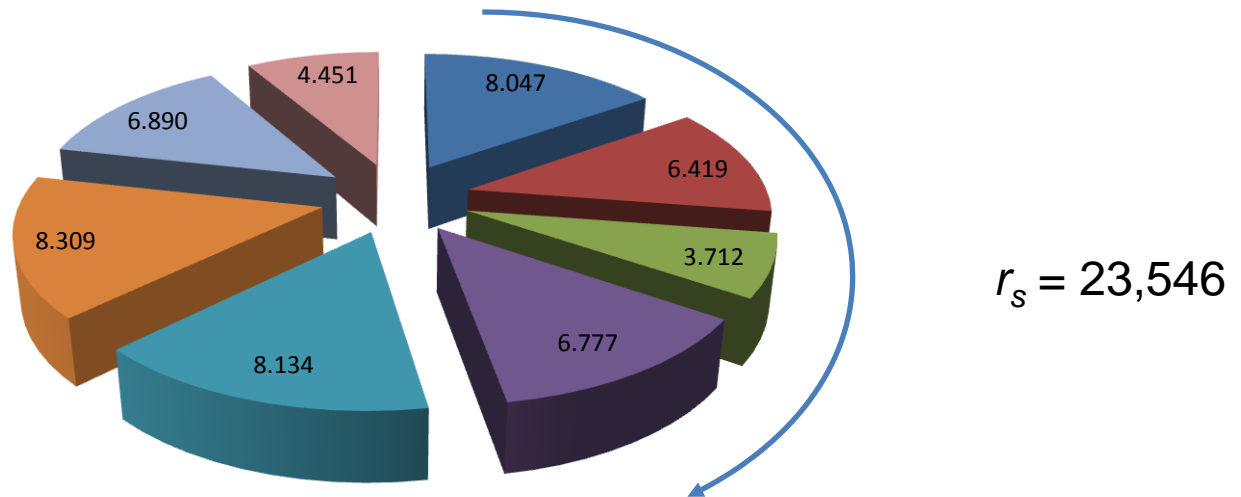
```
def show_pop(self, string2show, pop2show, f_in, function_in):  
    """Method to show population"""  
  
    # Show information concerning the population  
    print("\n"+string2show)  
  
    # Looping though the elements of the population  
    for i in range(len(pop2show)):  
        print(pop2show[i], "  %.3f"%f_in[i], "  %.3f"%function_in[i])
```

O operador seleção que usamos até o momento é relativamente simples de implementar, mas não é o mais comum no estudo de algoritmos genéticos. Uma abordagem mais robusta do operador seleção leva em conta o valor da função ajuste na seleção, mas não necessariamente elimina indivíduos com os valores mais baixos. Esta abordagem é chamada roleta de cassino.



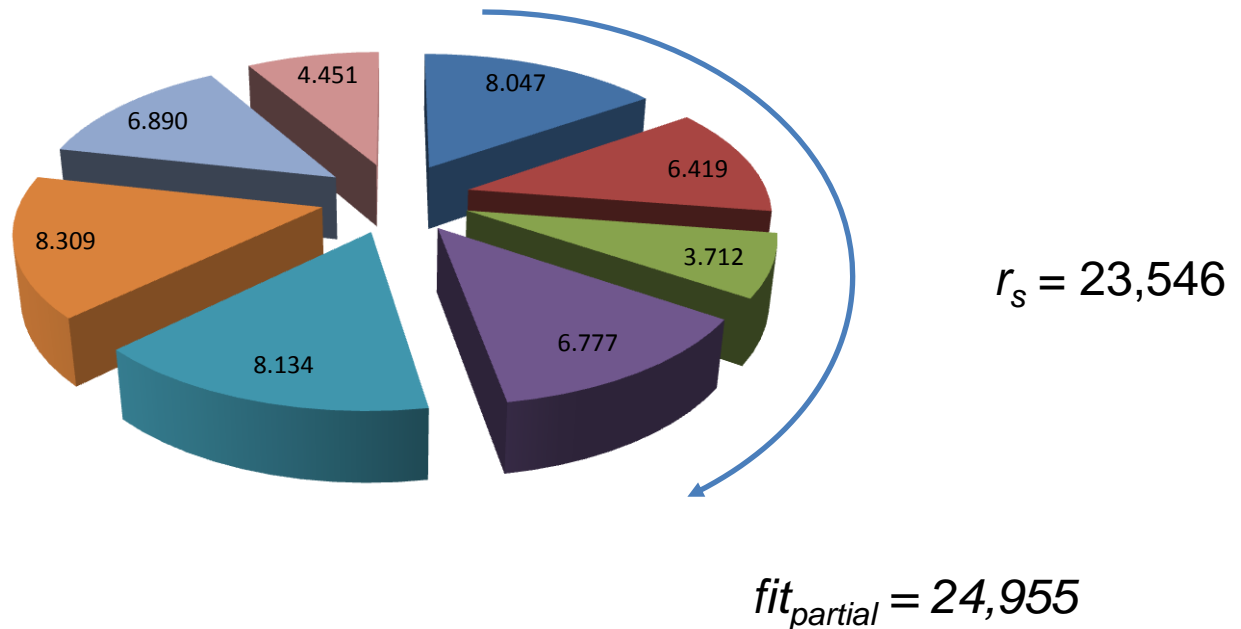


A abordagem de roleta de cassino soma todos os valores da função ajuste e atribui à variável  $fit_{sum}$ . Então é gerado um número aleatório ( $r_s$ ) entre zero e  $fit_{sum}$ . Em seguida um loop vai somando os valores da função ajuste de cada indivíduo da população, esses valores são somados à variável  $fit_{partial}$ . Quando a condição  $fit_{partial} > r_s$  for satisfeita, o loop é encerrado e o último indivíduo somado é considerado selecionado para a próxima geração. Esse processo se repete até que a população atinja o número de indivíduos estabelecido para a população.

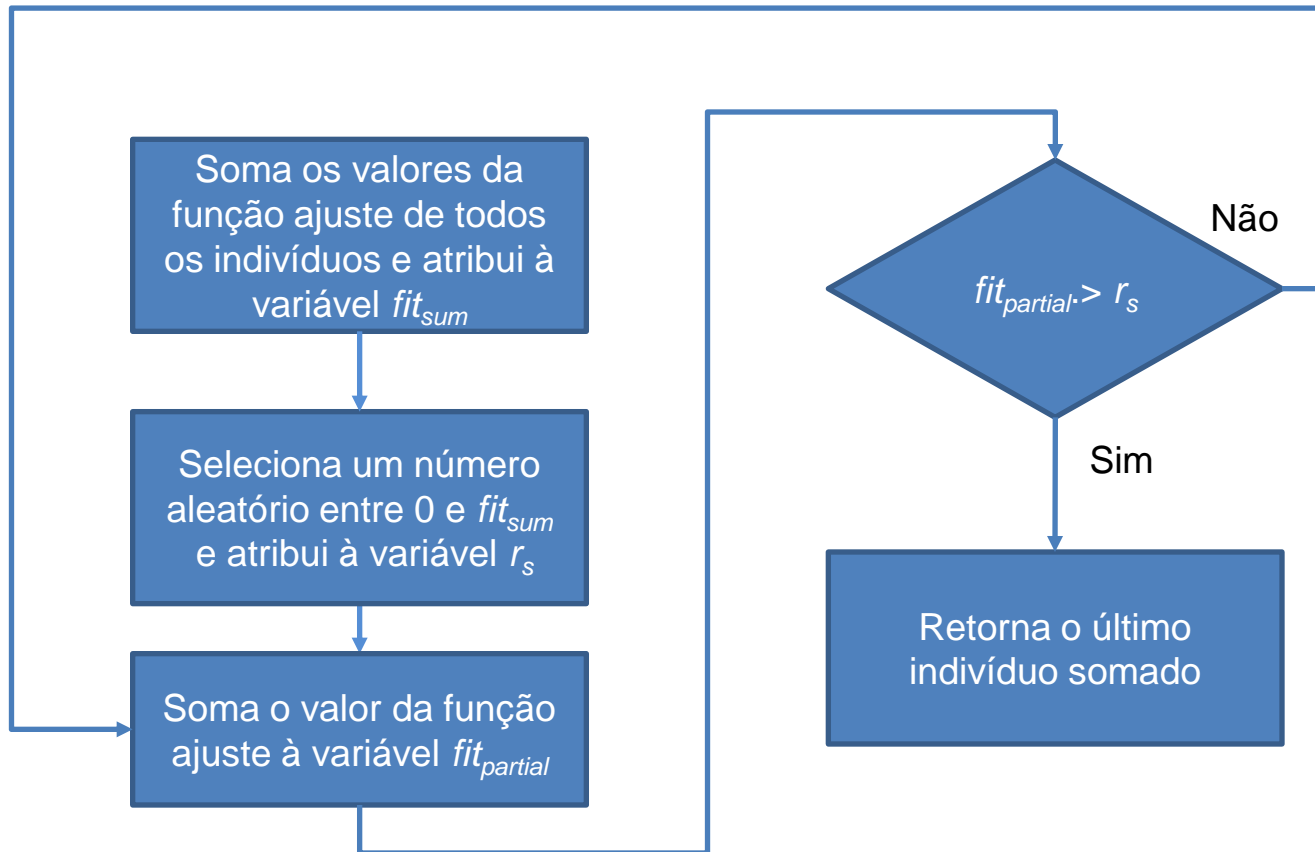


$$fit_{partial} = 24,955$$

Para ilustrar uma iteração do operador seleção com roleta de cassino, considere os valores da função ajuste mostrados no gráfico abaixo. Foi gerado um número aleatório entre zero e soma de todos os valores da função ajuste (52,739), por exemplo,  $r_s = 23,546$ . Em seguida somamos os valores de cada função ajuste e testamos a condição  $fit_{partial} > r_s$ , quando for satisfeita paramos a soma. O indivíduo que gerou o último valor da função ajuste (6,777) é selecionado.



Abaixo temos o fluxograma do algoritmo 2 de Coley 1999, que implementa a roleta de cassino para seleção de indivíduos.



Abaixo temos o método `roulette_wheel()`. Inicialmente somamos os valores da função ajuste de todos os indivíduos e atribuímos à variável `fit_sum`.

```
def roulette_wheel(self):  
    """Method for application of roulette wheel to a population"""  
    # Import library  
    import numpy as np  
    # Sum fitness function vaules  
    fit_sum = np.sum(self.fit)  
    # Set up empty list  
    new_pop = []  
    # Looping through all individuals  
    for j in range(self.pop_size):  
        # Generate random number  
        rs = np.random.uniform(0, fit_sum)  
        # Set up initial value for fit_partial  
        fit_partial = 0  
        # Looping through inidividual until fit_partial > rs  
        for i in range(self.pop_size):  
            fit_partial += self.fit[i]  
            # Check if condition is satisfied  
            if fit_partial > rs:  
                new_pop.append(self.current_pop[i])  
                break  
    # Update population  
    self.current_pop = new_pop
```

Agora dentro de um loop *for*, geramos uma número aleatório entre zero e a soma dos valores da função ajuste e atribuímos à variável *rs*.

```
def roulette_wheel(self):  
    """Method for application of roulette wheel to a population"""  
    # Import library  
    import numpy as np  
    # Sum fitness function vaules  
    fit_sum = np.sum(self.fit)  
    # Set up empty list  
    new_pop = []  
    # Looping through all individuals  
    for j in range(self.pop_size):  
        # Generate random number  
        rs =np.random.uniform(0,fit_sum)  
        # Set up initial value for fit_partial  
        fit_partial = 0  
        # Looping through inidividual until fit_partial > rs  
        for i in range(self.pop_size):  
            fit_partial += self.fit[i]  
            # Check if condition is satisfied  
            if fit_partial > rs:  
                new_pop.append(self.current_pop[i])  
                break  
    # Update population  
    self.current_pop = new_pop
```

Esse primeiro loop *for* varre toda população. Interno ao primeiro loop, temos um segundo loop *for* para realizarmos a função da roleta de cassino.

```
def roulette_wheel(self):  
    """Method for application of roulette wheel to a population"""  
    # Import library  
    import numpy as np  
    # Sum fitness function vaules  
    fit_sum = np.sum(self.fit)  
    # Set up empty list  
    new_pop = []  
    # Looping through all individuals  
    for j in range(self.pop_size):  
        # Generate random number  
        rs =np.random.uniform(0,fit_sum)  
        # Set up initial value for fit_partial  
        fit_partial = 0  
        # Looping through inividual until fit_partial > rs  
        for i in range(self.pop_size):  
            fit_partial += self.fit[i]  
            # Check if condition is satisfied  
            if fit_partial > rs:  
                new_pop.append(self.current_pop[i])  
                break  
    # Update population  
    self.current_pop = new_pop
```

No segundo loop vamos somando os valores da função ajuste até que este ultrapasse o número aleatório *rs*. Ao passar, adicionamos o último elemento à nova população.

```
def roulette_wheel(self):  
    """Method for application of roulette wheel to a population"""  
    # Import library  
    import numpy as np  
    # Sum fitness function vaules  
    fit_sum = np.sum(self.fit)  
    # Set up empty list  
    new_pop = []  
    # Looping through all individuals  
    for j in range(self.pop_size):  
        # Generate random number  
        rs = np.random.uniform(0, fit_sum)  
        # Set up initial value for fit_partial  
        fit_partial = 0  
        # Looping through inividual until fit_partial > rs  
        for i in range(self.pop_size):  
            fit_partial += self.fit[i]  
            # Check if condition is satisfied  
            if fit_partial > rs:  
                new_pop.append(self.current_pop[i])  
                break  
    # Update population  
    self.current_pop = new_pop
```

O processo é repetido até que a nova população, atribuída à variável *new\_pop*, volte a ter o número original de indivíduos. Por último, a população é atualizada.

```
def roulette_wheel(self):
    """Method for application of roulette wheel to a population"""
    # Import library
    import numpy as np
    # Sum fitness function vaules
    fit_sum = np.sum(self.fit)
    # Set up empty list
    new_pop = []
    # Looping through all individuals
    for j in range(self.pop_size):
        # Generate random number
        rs =np.random.uniform(0,fit_sum)
        # Set up initial value for fit_partial
        fit_partial = 0
        # Looping through inividual until fit_partial > rs
        for i in range(self.pop_size):
            fit_partial += self.fit[i]
            # Check if condition is satisfied
            if fit_partial > rs:
                new_pop.append(self.current_pop[i])
                break
    # Update population
    self.current_pop = new_pop
```



O uso da roleta no operador seleção pode levar a perda do melhor indivíduo de uma dada geração. Assim, é comum usarmos a opção de elitismo no algoritmo genético. Com elitismo o melhor indivíduo do operador seleção é salvo e mantido para próxima geração. Abaixo temos o método `elitism()`. Nesse método identificamos o maior valor da função ajuste na linha `max_fit = np.max(my_array)`.

```
def elitism(self):
    """Method for elitism"""
    # Import libraries
    import numpy as np
    import test_function as tf
    # Call bin2dec_array()
    dec = self.bin2dec_array(self.current_pop)
    # Call int2float()
    my_float = self.int2float(dec)
    # Instantiating an object of the Function() class and assigns it to the variable fit0
    fit0 = tf.Function(my_float)
    # Invoking the fitness_func() method
    my_fit = fit0.fitness_func()
    # Set up an array
    my_array = np.array(my_fit)
    # Find the maximum for the fitness function and assigns it to max_fit
    max_fit = np.max(my_array)
    # Set up an empty list
    new_pop = []
    # Find the position for the max_fit in the my_array
    pos_array = np.where( my_array == max_fit)
    # Get the integer for index
    my_index = int(pos_array[0][0])
    # Assign elite member to self.elite
    self.elite = self.current_pop[my_index]
    # Show elite member
    print("Elite member: ", self.elite, max_fit)
```

O valor atribuído à variável *max\_fit* é usado para identificar o índice do elemento do array que tem o máximo. Isso é feito em *pos\_array = np.where( my\_array == max\_fit)*. Como podemos ter mais de um índice com o máximo, é atribuído à variável *pos\_array* um array, onde nos interessa só uma das posições. Na linha *my\_index = int(pos\_array[0][0])* atribuímos o índice do elemento máximo à variável *my\_index*.

```
def elitism(self):
    """Method for elitism"""
    # Import libraries
    import numpy as np
    import test_function as tf
    # Call bin2dec_array()
    dec = self.bin2dec_array(self.current_pop)
    # Call int2float()
    my_float = self.int2float(dec)
    # Instantiating an object of the Function() class and assigns it to the variable fit0
    fit0 = tf.Function(my_float)
    # Invoking the fitness_func() method
    my_fit = fit0.fitness_func()
    # Set up an array
    my_array = np.array(my_fit)
    # Find the maximum for the fitness function and assigns it to max_fit
    max_fit = np.max(my_array)
    # Set up an empty list
    new_pop = []
    # Find the position for the max_fit in the my_array
    pos_array = np.where( my_array == max_fit)
    # Get the integer for index
    my_index = int(pos_array[0][0])
    # Assign elite member to self.elite
    self.elite = self.current_pop[my_index]
    # Show elite member
    print("Elite member: ", self.elite, max_fit)
```

Agora obtemos elemento elite com a linha `self.elite = self.current_pop[my_index]`. Essa variável será usada no método `include_elite`.

```
def elitism(self):
    """Method for elitism"""
    # Import libraries
    import numpy as np
    import test_function as tf
    # Call bin2dec_array()
    dec = self.bin2dec_array(self.current_pop)
    # Call int2float()
    my_float = self.int2float(dec)
    # Instantiating an object of the Function() class and assigns it to the variable fit0
    fit0 = tf.Function(my_float)
    # Invoking the fitness_func() method
    my_fit = fit0.fitness_func()
    # Set up an array
    my_array = np.array(my_fit)
    # Find the maximum for the fitness function and assigns it to max_fit
    max_fit = np.max(my_array)
    # Set up an empty list
    new_pop = []
    # Find the position for the max_fit in the my_array
    pos_array = np.where(my_array == max_fit)
    # Get the integer for index
    my_index = int(pos_array[0][0])
    # Assign elite member to self.elite
    self.elite = self.current_pop[my_index]
    # Show elite member
    print("Elite member: ", self.elite, max_fit)
```

O método `include_elite()` usa o elemento `self.elite` para substituir um elemento aleatório da população. Assim garantimos que o elemento elite está na população ao final da geração. O método `include_elite()` será evocado logo após a evocação do método do operador mutação.

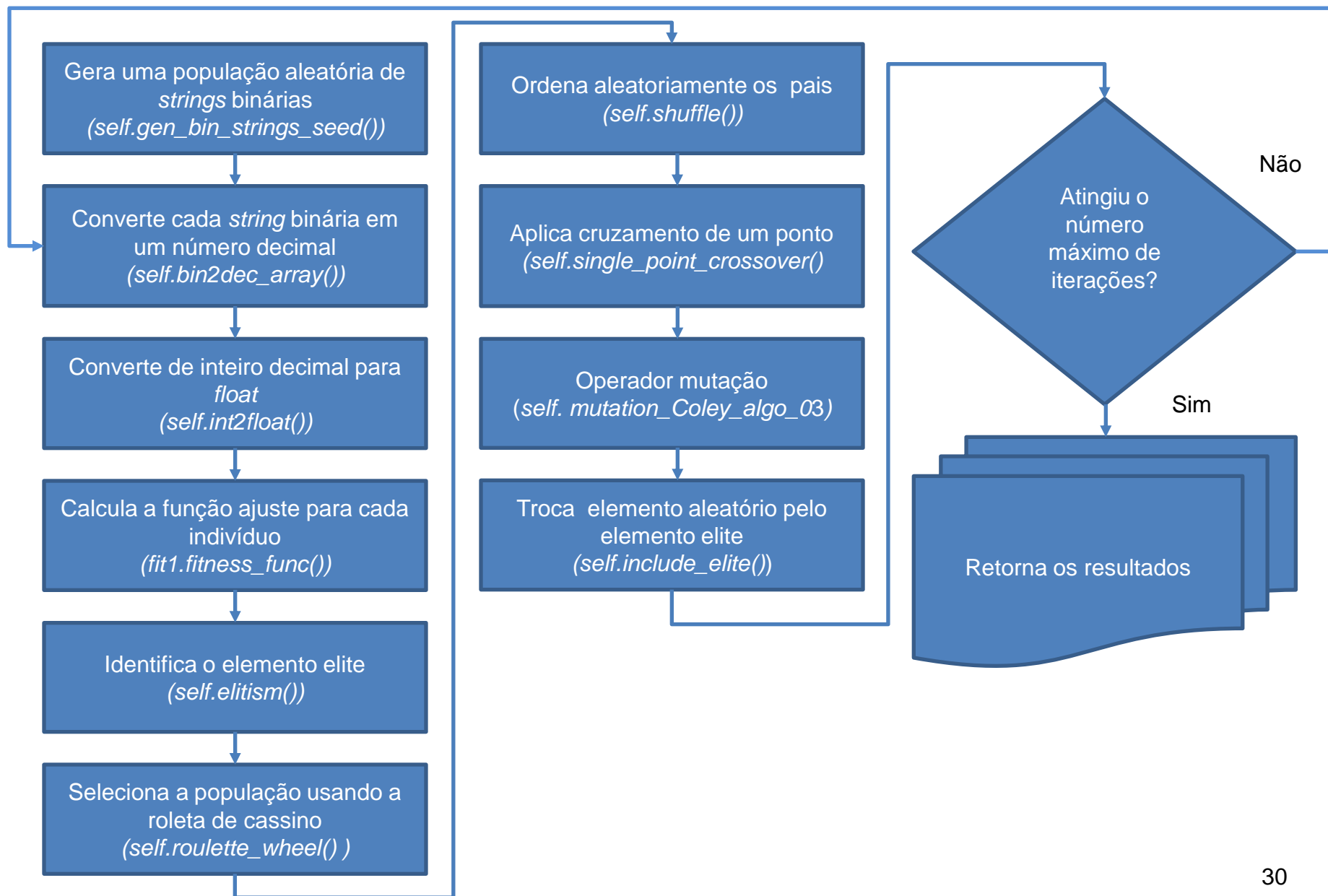
```
def include_elite(self):
    """Method to include elite member in the current population"""
    # Import library
    import numpy as np
    # Generate random number
    rs = np.random.randint(0, self.pop_size)

    # Set up empty list
    new_pop = []

    # Looping through population and change for elite member at random position
    for i in range(self.pop_size):
        if i != rs:
            new_pop.append(self.current_pop[i])
        else:
            new_pop.append(self.elite)
    # Update current_pop
    self.current_pop = new_pop
```

Nesse método geramos um número aleatório entre zero e o tamanho da população e atribuímos à variável *rs*. Em seguida criamos uma lista vazia que é atribuída à variável *new\_pop*. Na sequência, temos um loop *for* que varre a população e testa se o índice do elemento da população é diferente do número interior aleatório. Se for mantém o elemento, caso contrário o elemento é substituído pelo elemento elite. Por último, a população é atualizada, garantindo-se a presença do elemento elite.

```
def include_elite(self):  
    """Method to include elite member in the current population"""  
    # Import library  
    import numpy as np  
    # Generate random number  
    rs = np.random.randint(0, self.pop_size)  
  
    # Set up empty list  
    new_pop = []  
  
    # Looping through population and change for elite member at random position  
    for i in range(self.pop_size):  
        if i != rs:  
            new_pop.append(self.current_pop[i])  
        else:  
            new_pop.append(self.elite)  
    # Update current_pop  
    self.current_pop = new_pop
```



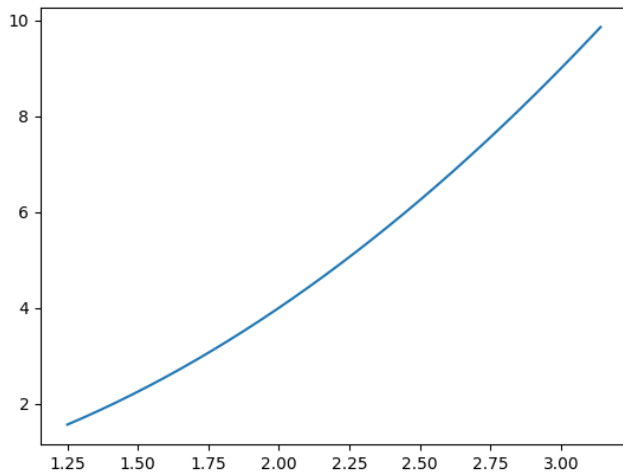
Até agora implementamos um algoritmo genético para achar o máximo da função  $f(x) = x^2$ , uma função relativamente simples. Normalmente as funções para os quais estamos interessados em achar o mínimo ou o máximo são mais complexas. A fim de padronizar testes de eficiência de algoritmos de otimização, são usadas funções matemáticas que apresentam maiores variações. Há diversas funções que são consideradas padrões para testes, principalmente para funções de mais de uma variável. Iremos aqui parte nos restringir a funções de uma variável.

Para facilitar a codificação, criamos uma classe a parte do código *lga01.py* onde está implementada a função ajuste. Abaixo temos o código *test\_function.py*. A linha em vermelho traz a codificação da função  $f(x) = x^2$ , que é a nossa função teste. Resumindo, usamos como função ajuste uma função teste, como objetivo de avaliar o desempenho do algoritmo.

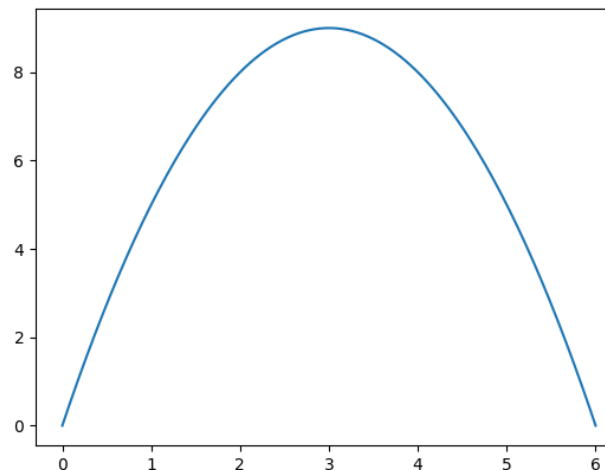
```
class Function(object):  
    """Class to calculate an one-variable mathematical function"""  
    def __init__(self,x):  
        self.x = x  
    # Definition of fitness_func()  
    def fitness_func(self):  
        """Method to calculate fitness function"""  
        import numpy as np  
  
        f = self.x**2          # [1.25,3.14]      conv_factor = 0.45      max = 3.141  
        return f
```

Abaixo temos o gráfico de algumas funções que podem ser implementadas em `test_function.py`.

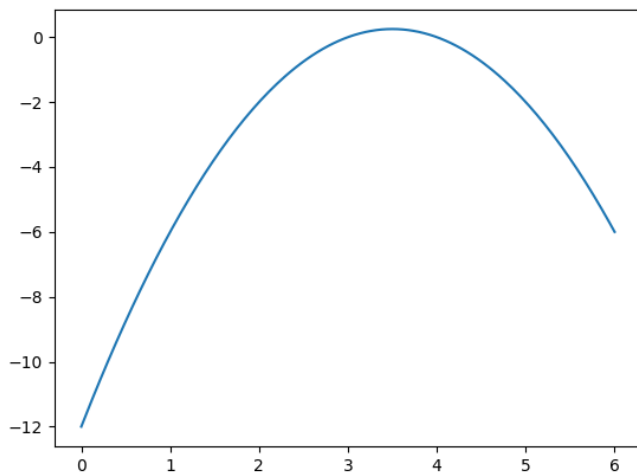
$$f(x) = x^2$$



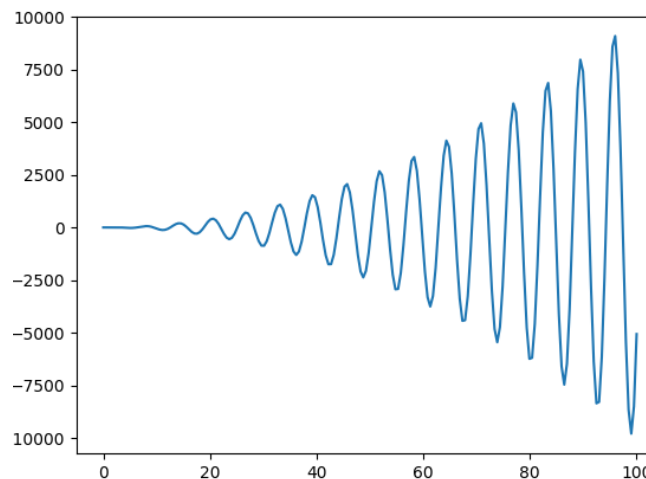
$$f(x) = 6x - x^2$$



$$f(x) = 7x - x^2 - 12$$



$$f(x) = x^2 \cdot \sin(x)$$





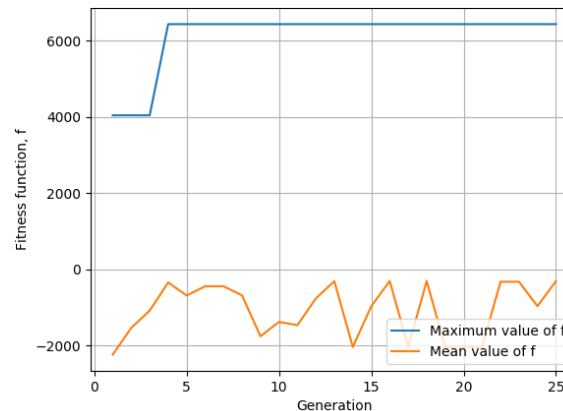
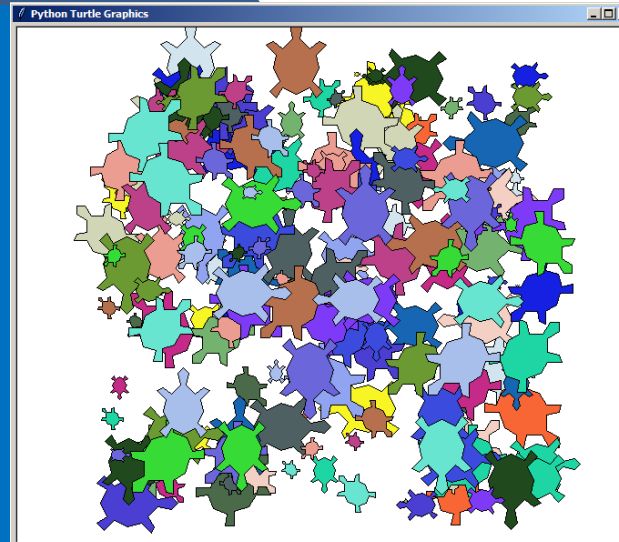
Usamos os seguintes parâmetros para o programa *lga01.py*.

```
pop_size = 8 # Population size  
max_iter = 25 # Number of iterations  
p_cross = 0.8 # Probability of crossover  
len_str = 12 # Length of strings  
p_mut = 0.01 # Probability of mutation_Coley_algo_03  
seed = 314159 # Seed for generation of random number  
min_f = 0.0 # Minimum float  
max_f = 100.0 # Maximum float  
conv_factor = 0.0005 # Conversion factor (fitness function to dimensions)
```

Foi usada a seguinte função teste,  $f(x) = x^2 \cdot \sin(x)$ , com intervalo entre 0 e 100.

Abaixo temos os resultados obtidos com os parâmetros do slide anterior para o programa *lga01.py*.

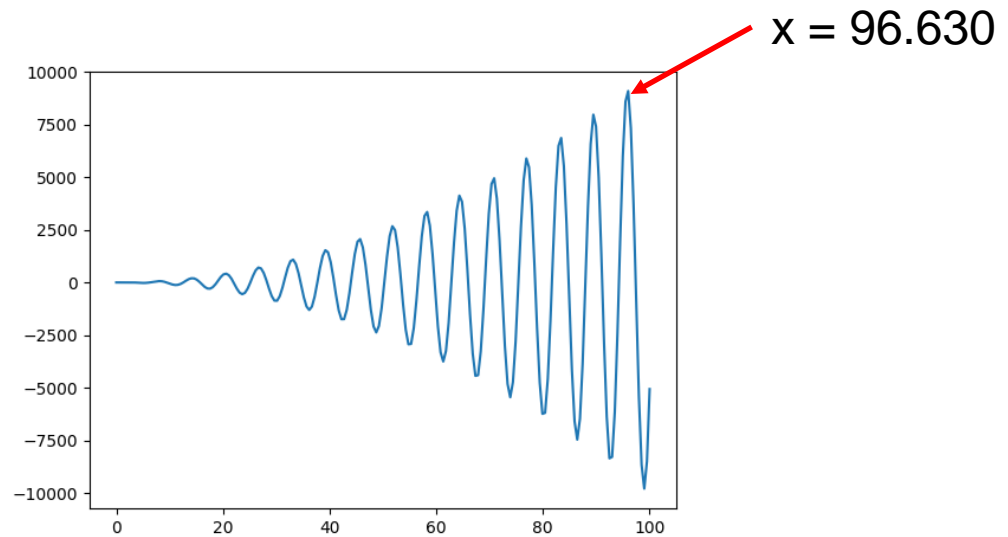
```
Population for generation 25
111101110101 96.630 6428.212
111011101011 93.260 -7260.322
110111010111 86.520 -7426.064
101110101111 73.040 -3765.778
011101011111 46.081 1834.788
111101110101 96.630 6428.212
110101111101 84.322 3412.940
101011111011 68.645 -2135.512
```



O melhor resultado é obtido para  $x = 96.630$ , que é o máximo indicado na função abaixo.

Population for generation 25

<b>111101110101</b>	<b>96.630</b>	<b>6428.212</b>
111011101011	93.260	-7260.322
110111010111	86.520	-7426.064
101110101111	73.040	-3765.778
011101011111	46.081	1834.788
111101110101	96.630	6428.212
110101111101	84.322	3412.940
101011111011	68.645	-2135.512



# Projeto 1 - Algoritmo Genético

Programa: *lga02.py*

Data de entrega: 10-05-2017

Modifique o código do programa *lga01.py* de forma que o programa leia os parâmetros do algoritmo genético a partir de um arquivo de entrada (*lga.in*). O arquivo de entrada deve ter as seguintes informações:

```
pop_size  
max_iter  
p_cross  
len_str  
p_mut  
seed  
min_f  
max_f  
conv_factor  
elitism_in  
draw_turtle  
plot_func
```

Os parâmetros *elitism\_in*, *draw\_turtle* e *plot\_func* são variáveis booleanas que podem ser *True* ou *False*. O parâmetro *elitism\_in* indica se o elitismo será considerado. O parâmetro *draw\_turtle* é uma variável booleana e indica se as tartarugas serão desenhadas ou não. O último parâmetro (*plot\_func*) indica se será gerado o gráfico da função ajuste. Esses três últimos parâmetros não estão no código *lga01.py* e devem ser incorporados no novo código.

Além disso, o novo código irá gerar um arquivo *lga.log* com as informações de cada geração (população, valor em *float* e valor da função ajuste). É similar ao que o programa *lga01.py* mostra na tela para cada geração, só que agora será gravado num arquivo *.log*. No final do arquivo *lga.log*, deve ser gravada uma linha com as informações do melhor indivíduo (string binária, valor em *float* e valor da função ajuste). O programa também gera um arquivo CSV, chamado *lga.csv*, com as seguintes informações:

<i>Generation</i>	<i>Elite Member</i>	<i>Mean Fitness Function</i>	<i>Maximum Fitness Function</i>
-------------------	---------------------	------------------------------	---------------------------------

...

O novo programa deve chamar-se *lga02.py* e tem que ser testado para as seguintes funções:

$$f(x) = x^2, \text{ no intervalo } [1.25, 3.14] \text{ e } conv\_factor = 0.45,$$

$$f(x) = 6x - x^2, \text{ no intervalo } [0, 6], \text{ e } conv\_factor = 0.45,$$

$$f(x) = 7x - x^2 - 12, \text{ no intervalo } [0, 6] \text{ e } conv\_factor = 0.6,$$

$$f(x) = x^2 \cdot \sin(x), \text{ no intervalo } [0, 100] \text{ e } conv\_factor = 0.0005.$$

O programa deve ser entregue até o dia 10-05-2017 com um relatório sucinto descrevendo o desempenho do programa com as funções descritas no slide anterior. Devem ser testados diversos valores dos parâmetros de entrada do arquivo *lga.in*. Procurem o conjunto de parâmetros que gerem os melhores resultados para a localização do máximo das funções descritas no slide anterior.

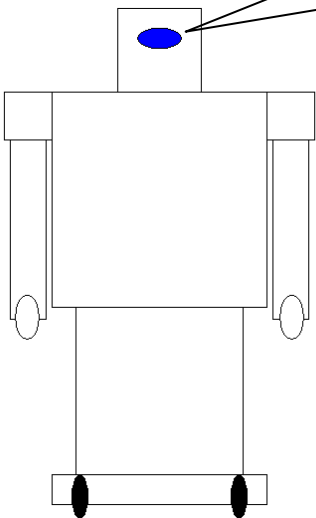
A seguir temos a lista de programas da presente aula.

Lista de programas:

*hexadec.py*  
*lga01.py*  
*random\_loop.py*  
*random\_with\_seed.py*  
*test\_function.py*  
*touche.py*



This text was produced in a HP Pavillon dm4 notebook with 6GB of memory, a 500 GB hard disk, and an Intel® Core® i7 M620 CPU @ 2.67 GHz running Windows 7 Professional. Text and layout were generated using PowerPoint 2007. Turtles drawings on slide 34 were generated with Turtle library. Plots on slides 32, 34, and 35 were generated by Matplotlib. The pizza charts on slides 16, 17, and 18 were generated with Excel 2007. The image on the first slide was taken from <http://briandcolwell.com/2017/02/artificial-intelligence-the-big-list-of-things-you-should-know/.html> on March 26<sup>th</sup> 2017. This text uses Arial font.



- BRESSERT, Eli. **SciPy and NumPy**. Sebastopol: O'Reilly Media, Inc., 2013. 57 p. [Link](#)
- DAWSON, Michael. **Python Programming, for the Absolute Beginner**. 3ed. Boston: Course Technology, 2010. 455 p.
- COLEY, David A. **An Introduction to Genetic Algorithms for Scientists and Engineers**. Singapore: World Scientific Publishing Co. Pte. Ltd., 1999. 227 p.
- EIBEN, A. E., SMITH, J. E. **Introduction to Evolutionary Computing. Natural Computing Series**. 2nd ed. Berlin: Springer-Verlag, 2015. 287 p. [Link](#)
- HACKELING G. **Mastering Machine Learning with scikit-learn**. Birmingham: Packt Publishing Ltd., 2014. 221 p. [Link](#)
- HETLAND, Magnus Lie. **Python Algorithms. Mastering Basic Algorithms in the Python Language**. 2ed. Nova York: Springer Science+Business Media LLC, 2010. 294 p. [Link](#)
- IDRIS, Ivan. **NumPy 1.5. An action-packed guide dor the easy-to-use, high performance, Python based free open source NumPy mathematical library using real-world examples. Beginner's Guide**. Birmingham: Packt Publishing Ltd., 2011. 212 p. [Link](#)
- LUTZ, Mark. **Programming Python**. 4ed. Sebastopol: O'Reilly Media, Inc., 2010. 1584 p. [Link](#)
- TOSI, Sandro. **Matplotlib for Python Developers**. Birmingham: Packt Publishing Ltd., 2009. 293 p. [Link](#)

Última atualização: 28 de maio de 2017.