Projeto de circuitos elétricos usando algoritmo genético

Jaedson Barbosa Serafim

2 de dezembro de 2022

Sumário

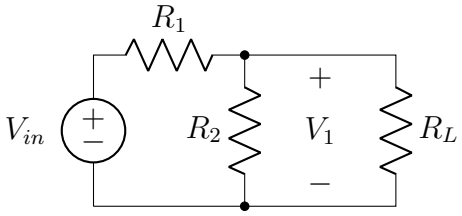

Figura 1: Circuito em análise.

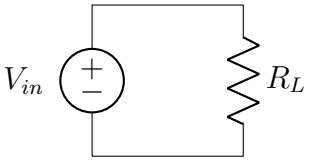

Figura 2: Circuito de máxima transferência de energia.

1 Introdução

Neste trabalho foi feito o cálculo dos valores de resistências para a máxima transferência de energia para o resistor R_L no circuito 1 usando algoritmo genético.

A vantagem deste circuito é que, dada a sua simplicidade, é possível facilmente saber qual a resposta para solução do problema. A máxima transferência de energia ocorre quando o resistor R_1 é substituído por um curto, que pode ser interpretado como uma resistência muito pequena, enquanto o resistor R_2 é substituído por um aberto, interpretável como uma resistência muito grande, obtendo assim o circuito equivalente 2.

A potência dissipada pelo resistor R_L no circuito equivalente 2 pode ser calculada usando a equação 1.

$$P_{max} = \frac{V_{in}^2}{R_L} \tag{1}$$

Substituindo V_{in} por $10V$ e R_L por 50Ω na equação 1 chegamos à máxima potência teórica para resistência R_L no circuito 1:

$$P_{max} = \frac{10^2}{50} = 2W \tag{2}$$

Nessa simplificação da figura 2, como não há outros elementos resistivos no circuito, toda a potência fornecida pela fonte é entregue à resistência, ou seja, o rendimento do circuito é de 100%.

Porém, e se não soubéssemos como fazer tais contas, e se sequer soubéssemos que circuito é esse? Nesse caso poderia ser feita essa busca pelos valores de R_1 e R_2 usando algoritmos genéticos e esse é o objetivo deste trabalho: projetar elementos de um circuito para atender a uma especificação.

2 Função de rendimento

A função da potência de saída do circuito 1 pode ser definida como sendo:

$$P_{out} = \frac{V_1^2}{R_L} \tag{3}$$

V_1 pode ser escrito em função de V_{in} e das resistências:

$$\begin{aligned} V_1 &= V_{in} * \frac{R_2 \parallel R_L}{R_1 + R_2 \parallel R_L} \\ &= V_{in} * \frac{1}{1 + \frac{R1}{R_2 \parallel R_L}} \\ &= V_{in} * \frac{1}{1 + \frac{R1}{\frac{R_2 * R_L}{R_2 + R_L}}} \\ &= V_{in} * \frac{1}{1 + R1 * \frac{R_2 + R_L}{R_2 * R_L}} \end{aligned} \tag{4}$$

Por fim, a potência de entrada do circuito 1 é definida por:

$$P_{in} = V_{in} * \frac{V_{in} - V_1}{R_1} \tag{5}$$

Usando as equações 3, 4 e 5 é possível então calcular a eficiência do circuito, ou seja, a relação entre a potência de saída e a potência de entrada, dada pela fórmula:

$$\eta = \frac{P_{out}}{P_{in}} \tag{6}$$

3 Análise inicial

Existem resistores com diversos valores de resistência, mas aqui a busca foi limitada de $1m\Omega$ a $100k\Omega$, que é um intervalo de busca bem abrangente caso fosse usada uma escala linear, por isso foi adotada uma escala logarítmica, para reduzir o intervalo de busca para $[-2, 5]$.

Código 1: Análise inicial do problema.

```
import matplotlib.pyplot as plt
from matplotlib import cm
import numpy as np

def calc_efficiency(R1, R2):
```

```
    Rl = 50
    Vin = 10
    V1 = Vin * (1 / (1 + R1 * (R2 + Rl) / (R2 * Rl)))
    Pout = V1 * V1 / Rl
    Pin = Vin * (Vin - V1) / R1
    return Pout / Pin

def plot_surface():
    R1 = np.logspace(-2, 2, num=255)
    R2 = np.logspace(-2, 2, num=255)
    ax = plt.figure().add_subplot(projection='3d')
    R1, R2 = np.meshgrid(R1, R2)
    surf = ax.plot_surface(R1, R2, calc_efficiency(R1, R2), cmap=cm.
        coolwarm)
    plt.xlabel('R1')
    plt.ylabel('R2')
    plt.colorbar(surf)
    plt.savefig('fig/surface.png')

def plot_efficiency_vs_r1():
    R1 = np.logspace(-2, 5, num=255)
    plt.figure()
    plt.plot(R1, calc_efficiency(R1, 100_000), label="100Ωk")
    plt.plot(R1, calc_efficiency(R1, 1_000), label="1Ωk")
    plt.plot(R1, calc_efficiency(R1, 1), label="Ω1")
    plt.legend(title='R2 values')
    plt.ylabel('Efficiency')
    plt.xscale('log')
    plt.xlabel('R1')
    plt.savefig('fig/efficiency_vs_r1.png')

def plot_efficiency_vs_r2():
    R2 = np.logspace(-2, 5, num=255)
    plt.figure()
    plt.plot(R2, calc_efficiency(100_000, R2), label="100Ωk")
    plt.plot(R2, calc_efficiency(1_000, R2), label="1Ωk")
    plt.plot(R2, calc_efficiency(1, R2), label="Ω1")
```

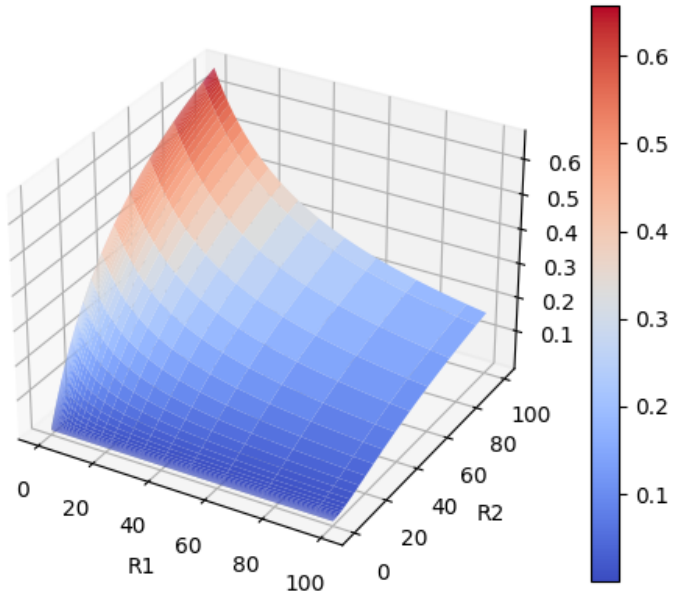

Figura 3: Eficiência em função de R_1 e R_2.

```
    plt.legend(title='R1 values')
    plt.ylabel('Efficiency')
    plt.xscale('log')
    plt.xlabel('R2')
    plt.savefig('fig/efficiency_vs_r2.png')

plot_surface()
plot_efficiency_vs_r1()
plot_efficiency_vs_r2()
```

Usando como base as fórmulas do capítulo 2 foi escrito o código 1, responsável por plotar algumas figuras que nos ajudam a entender melhor o problema.

A figura 3 demonstra que não existem máximos locais na função avaliada, apenas um máximo global, o que facilita a sua busca, embora este esteja nos extremos do domínio da busca, o que dificulta sua descoberta, ou seja, é bem difícil chegar ao ponto exato de máximo global, mas é fácil chegar próximo dele.

As figuras 4 e 5 foram plotadas com o eixo X em escala logarítmica e assim a curva exibida tornou-se "legível", comprovando que a melhor forma de buscar os melhores valores de R_1 e R_2 é analisando em potências de 10.

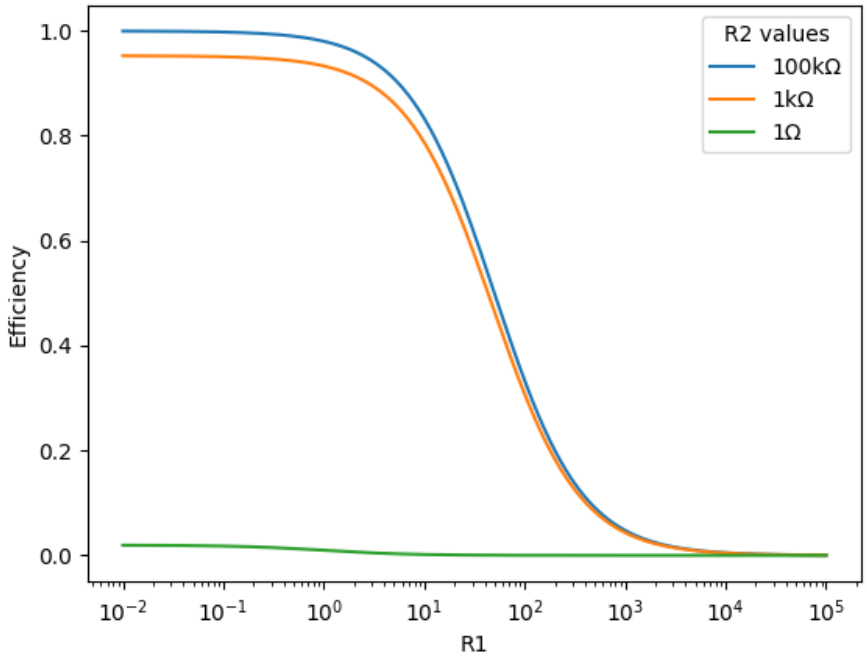

Figura 4: Curvas de eficiência em função de R_1 para 3 valores distintos de R_2.

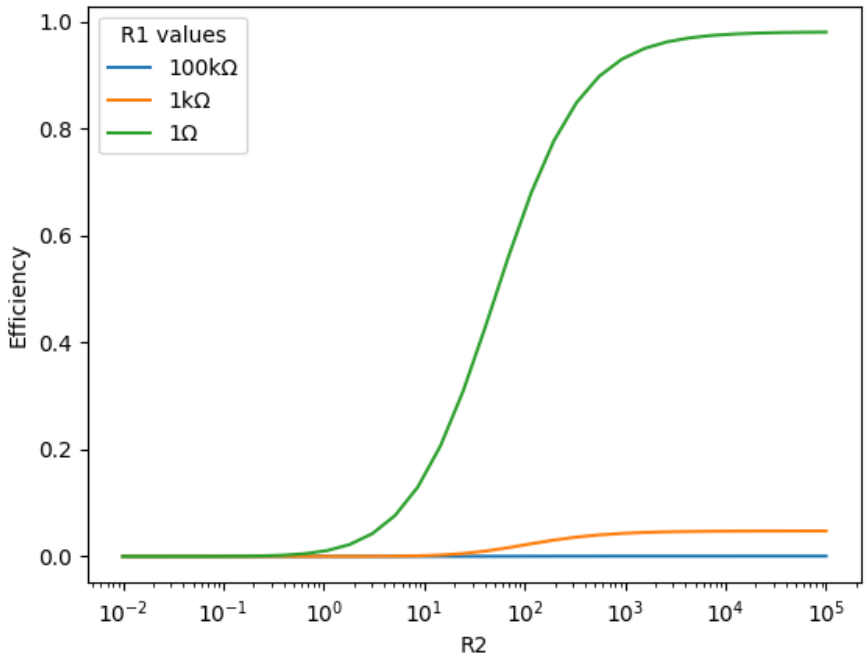

Figura 5: Curvas de eficiência em função de R_2 para 3 valores distintos de R_1.

Em relação à quantidade de bits, foi adotado apenas 8 bits e sua escolha foi puramente arbitrária, ela ocorreu devido a 1 byte ter 8 bits e assim, caso este problema estivesse sendo desenvolvido em alguma linguagem com tipagem estática poderia ser usado um número de tipo char (para C) ou u8 (para Rust).

Caso 8 bits pareçam pouco, vale mencionar que as figuras 4 e 5 foram geradas usando apenas os pontos representáveis por 8 bits, ou seja, usando apenas 255 pontos, como pode ser visto no código 1.

4 Função de *fitness*

Código 2: Simulação do circuito usando PySpice.

```
from PySpice.Spice.Netlist import Circuit

circuit = Circuit('Maxima transferencia de potencia')
Vin = 10
Rl = 50
circuit.V('input', 1, circuit.gnd, Vin)
R1 = circuit.R(1, 1, 2)
R2 = circuit.R(2, 2, circuit.gnd)
circuit.R(3, 2, circuit.gnd, Rl)
simulator = circuit.simulator(temperature=25, nominal_temperature
    =25)

def simulate(rs):
        r1 = 10**rs[0]
        r2 = 10**rs[1]
        R1.resistance = r1
        R2.resistance = r2
        analysis = simulator.operating_point()
        V1 = float(analysis['2'])
        Pout = V1 * V1 / Rl
        Pin = Vin * (Vin - V1) / r1
        return Pout / Pin
```

Muitas vezes é necessário usar ferramentas externas para a solução de um problema e aqui não é diferente, por isso a biblioteca PySpice é importada na primeira linha do código 2, para

simular o circuito e assim obtermos o valor da tensão V_1, usada para calcular as potências e assim conseguirmos o rendimento do circuito. "Por baixo dos panos", o que esta biblioteca faz é passar os parâmetros para outro programa, o Ngspice, e interpreta suas respostas.

5 Algoritmo genético

5.1 Implementação

Código 3: Algoritmo genético com codificação binária.

```
from simulation import simulate
from numpy.random import randint, rand, choice
from numpy import divide, sum
from functools import reduce

# define range for resistor
r_range = [-2, 5]
# define range for input
bounds = [r_range, r_range]
# define the total iterations
target = 0.99
# bits per variable
n_bits = 8
# maximum numer of iterations
n_iter_max = 25
# number of simulations
n_simulations = 5

# decode bitstring to numbers
def decode(bounds, n_bits, bitstring):
        decoded = list()
        largest = 2**n_bits-1
        for i in range(len(bounds)):
                # extract the substring
                start, end = i * n_bits, (i + 1) * n_bits
                # convert bitarray to integer
```

```
                integer = reduce(lambda a, b: a * 2 + b, bitstring[
                    start:end])
                # scale integer to desired range
                value = bounds[i][0] + (integer/largest) * (bounds[
                    i][1] - bounds[i][0])
                # store
                decoded.append(value)
        return decoded

# tournament selection
def tournament_selection(pop, scores):
        # first random selection
        selection_ix = randint(len(pop))
        for ix in randint(0, len(pop), 2):
                # check if better (e.g. perform a tournament)
                if scores[ix] > scores[selection_ix]:
                        selection_ix = ix
        return pop[selection_ix]

# roulette selection
def roulette_selection(pop, scores):
        # first normalize scores
        scores = divide(scores, sum(scores))
        # then choose a random index based on the probability
            vector
        selection_ix = choice(len(pop), p=scores)
        # finally returns the element from that index
        return pop[selection_ix]

# crossover two parents to create two children
def crossover(p1, p2, r_cross):
        # children are copies of parents by default
        c1, c2 = p1.copy(), p2.copy()
        # check for recombination
        if rand() < r_cross:
                # select crossover point that is not on the end of
                    the string
```

```
                pt = randint(1, len(p1)-2)
                # perform crossover
                c1 = p1[:pt] + p2[pt:]
                c2 = p2[:pt] + p1[pt:]
        return [c1, c2]

# mutation operator
def mutation(bitstring, r_mut):
        for i in range(len(bitstring)):
                # check for a mutation
                if rand() < r_mut:
                        # flip the bit
                        bitstring[i] = 1 - bitstring[i]

# genetic algorithm
def genetic_algorithm(objective, selection_alg, bounds, n_bits,
   target, n_pop, r_cross, r_mut):
        # number of generations needed to find the answer
        n_iter = 0
        # history of best scores
        scores_history = list()
        # initial population of random bitstring
        pop = [randint(0, 2, n_bits*len(bounds)).tolist() for _ in
           range(n_pop)]
        # keep track of best solution
        best = pop[0]
        best_eval = objective(decode(bounds, n_bits, pop[0]))
        # enumerate generations
        while best_eval < target and n_iter < n_iter_max:
                n_iter += 1
                # decode population
                decoded = [decode(bounds, n_bits, p) for p in pop]
                # evaluate all candidates in the population
                scores = [objective(d) for d in decoded]
                # check for new best solution
                scores_best = max(scores)
                scores_history.append(scores_best)
```

```
                if scores_best > best_eval:
                        i = scores.index(scores_best)
                        best, best_eval = pop[i], scores[i]
                # select parents
                selected = [selection_alg(pop, scores) for _ in
                   range(n_pop)]
                # create the next generation
                children = list()
                for i in range(0, n_pop, 2):
                        # get selected parents in pairs
                        p1, p2 = selected[i], selected[i+1]
                        # crossover and mutation
                        for c in crossover(p1, p2, r_cross):
                                # mutation
                                mutation(c, r_mut)
                                # store for next generation
                                children.append(c)
                # replace population
                pop = children
        best_decoded = decode(bounds, n_bits, best)
        return [best_decoded, best_eval, scores_history]

# perform the genetic algorithm search
def main(selection_alg_name, n_pop, r_cross, r_mut):
        r_mut /= float(n_bits) * len(bounds)
        selection_alg = tournament_selection if selection_alg_name
           == 'tournament' else roulette_selection
        results = list()
        for _ in range(n_simulations):
                result = genetic_algorithm(simulate, selection_alg,
                    bounds, n_bits, target, n_pop, r_cross, r_mut)
                results.append(result)
        return results
```

O algoritmo genético desenvolvido neste trabalho é na verdade um aprimoramento do criado por Brownlee (2021), que embora funcionasse razoavelmente bem, não atendia a todas as solicitações deste projeto. Seguindo a ordem cronológica do projeto, as principais melhorias im-

plementadas estão logo abaixo descritas.

1. Correção da variável *largest* da função *decode* pois, para uma quantidade de n bits, o maior valor possível não é 2^n, mas sim $2^n - 1$.

2. Adicionada seleção pelo método da roleta, que pode ser escolhida passando-a como sendo o parâmetro *selection_alg* da função *genetic_algorithm*, tarefa desempenhada pela função *main*, que a escolhe caso tenha sido chamada com seu parâmetro *selection_alg_name* igual a *roulette*[1].

3. Adicionado histórico de melhor *fitness* de cada geração.

4. Substituída quantidade fixa de gerações por comparação de *fitness* com valor alvo, ou seja, ao invés de sempre executar a mesma quantidade de gerações, agora o algoritmo é interrompido tão logo a função de *fitness* de qualquer indivíduo da população retorne um valor igual ou maior ao alvo definido anteriormente.

5. Corrigido erro com a variável *best* na função *genetic_algorithm* que ocorria quando o primeiro indivíduo da população inicial atendia ao critério de parada, pois seu valor inicial era igual a 0, ou seja, de tipo inteiro, enquanto a função de decodificação na penúltima linha da função esperava um vetor de bits. A solução foi simples, bastou substituir o 0 pelo primeiro indivíduo da população. Cabe ressaltar que este erro só ocorria por causa da alteração do critério de parada.

6. Adicionado novo critério de parada: o número de iterações agora precisa ser inferior a um valor limite. Esse critério é importante pois quando são feitos testes com taxa de mutação igual a zero é possível que o algoritmo genético jamais chegue a uma solução, daí a razão de fazer essa limitação.

5.2 Especificações

Esse código tem implementada a forma mais simples de algoritmo genético, sempre priorizando a aleatoriedade do processo e a simplicidade do algoritmo, assim alguns parâmetros são fixos em todos as execuções deste algoritmo a partir da função *main*, são eles:

- intervalo de valores (*r_range*) igual a $[-2, 5]$;
- alvo (*target*) igual a $0,99$, ou seja, 99% da energia fornecida pela fonte é dissipada na resistência R_L;

[1]Na verdade, qualquer valor passado diferente de *tournament* acarretará no uso da seleção pelo método da roleta, pois não foi implementada nenhuma lógica de validação de parâmetros

- codificação binária da população usando 8 bits para representar cada variável em análise, totalizando 16 bits por indivíduo dado que são 2 variáveis;
- simulação limitada a 25 iterações;
- 5 simulações por chamada da função *main*, garantindo resultados mais precisos; e
- taxa de mutação dividida pela quantidade de bits de cada indivíduo, assim o valor enviado à função é referente à probabilidade de um indivíduo ter uma mutação em algum bit.

> O AG básico descarta a geração anterior e considera para a futura apenas os descendentes obtidos. A técnica Elitista consiste em reintroduzir o indivíduo melhor avaliado de uma geração para a seguinte, evitando a perda de informações importantes presentes em indivíduos de alta avaliação e que podem ser perdidas durante os processos de seleção e cruzamento. Algumas técnicas controlam o número de vezes que o indivíduo pode ser reintroduzindo, o que contribui para evitar convergência a máximos locais. (BENTO; KAGAN, 2008, p. 6)

Apesar do operador de elitismo muitas vezes contribuir para uma maior velocidade de convergência, ele cria alguns problemas como a maior probabilidade de estagnação em um máximo local, como pôde ser percebido no último trabalho, além de também acrescentar uma nova camada de complexidade à solução e dificultar o seu processamento em paralelo para conjuntos de dados muito extensos. Além de que, uma menor taxa de cruzamento aumenta a probabilidade dos bons indivíduos continuarem dentre a população e a correta escolha desse parâmetro faz com que o algoritmo não seja tão penalizado pela ausência do operador de elitismo. Em outras palavras, a taxa de elitismo é igual a zero em todas as execuções.

6 Simulações

Código 4: Simulações e plotagem de gráficos.

```
from index import main
import matplotlib.pyplot as plt
import numpy as np
from statistics import mean

r_cross_opts = (0.6, 0.7, 0.8, 0.9, 1)
r_mut_opts = (0.5, 1, 1.5, 2)
results_pop_4_roulette = list()
results_pop_4_tournament = list()
```

```
results_pop_8_roulette = list()
results_pop_8_tournament = list()

for r_cross in r_cross_opts:
  row_results_pop_4_roulette = list()
  row_results_pop_4_tournament = list()
  row_results_pop_8_roulette = list()
  row_results_pop_8_tournament = list()

  for r_mut in r_mut_opts:
    print((r_cross, r_mut))
    row_results_pop_4_roulette.append(main('roulette', 4, r_cross,
      r_mut))
    row_results_pop_4_tournament.append(main('tournament', 4,
      r_cross, r_mut))
    row_results_pop_8_roulette.append(main('roulette', 8, r_cross,
      r_mut))
    row_results_pop_8_tournament.append(main('tournament', 8,
      r_cross, r_mut))

  results_pop_4_roulette.append(row_results_pop_4_roulette)
  results_pop_4_tournament.append(row_results_pop_4_tournament)
  results_pop_8_roulette.append(row_results_pop_8_roulette)
  results_pop_8_tournament.append(row_results_pop_8_tournament)

X, Y = np.meshgrid(r_mut_opts, r_cross_opts)

def plot_colormesh(name, func, vmax = 25):
  calc_z = lambda results: np.array([[func([len(sim_result[2]) for
    sim_result in col]) for col in row] for row in results])
  fig, ax = plt.subplots(2, 2, sharex=True, sharey=True)

  fig.set_figwidth(8)
  fig.set_figheight(12)

  meshopts = {'cmap': 'coolwarm', 'vmin': 0, 'vmax': vmax}
```

```
    surf = ax[0][0].pcolormesh(X, Y, calc_z(results_pop_4_roulette),
        **meshopts)
    ax[0][0].set_xlabel('mutation rate')
    ax[0][0].set_ylabel('crossover rate')
    ax[0][0].set_title('Population=4 (Roulette)')

    surf = ax[0][1].pcolormesh(X, Y, calc_z(results_pop_4_tournament)
        , **meshopts)
    ax[0][1].set_xlabel('mutation rate')
    ax[0][1].set_ylabel('crossover rate')
    ax[0][1].set_title('Population=4 (Tournament)')

    surf = ax[1][0].pcolormesh(X, Y, calc_z(results_pop_8_roulette),
        **meshopts)
    ax[1][0].set_xlabel('mutation rate')
    ax[1][0].set_ylabel('crossover rate')
    ax[1][0].set_title('Population=8 (Roulette)')

    surf = ax[1][1].pcolormesh(X, Y, calc_z(results_pop_8_tournament)
        , **meshopts)
    ax[1][1].set_xlabel('mutation rate')
    ax[1][1].set_ylabel('crossover rate')
    ax[1][1].set_title('Population=8 (Tournament)')

    fig.colorbar(surf, ax=ax, orientation='horizontal', pad=0.07)
    plt.savefig(f'fig/{name}.png')

plot_colormesh('mean_colors', mean)
plot_colormesh('max_colors', max)
plot_colormesh('min_colors', min)

plt.figure()

quant_normal = 0
quant_primeira = 0
quant_nunca = 0
```

```
for result in (results_pop_4_roulette, results_pop_4_tournament,
    results_pop_8_roulette, results_pop_8_tournament):
  for row in result:
    for col in row:
      for sim_result in col:
        scores_history = sim_result[2][0:25]
        if len(scores_history) == 0:
          quant_primeira += 1
          continue
        if sim_result[1] < 0.99:
          quant_nunca += 1
        if len(scores_history) < 25:
          last_score = scores_history[len(scores_history) - 1]
          scores_history += [last_score] * (25 - len(scores_history
              ))
        plt.plot(range(1,26), scores_history, linewidth=0.1, color=
            'black')
        quant_normal += 1
plt.xlabel('Number of generations')
plt.ylabel('Delivered power')
plt.xscale('log')
plt.savefig('fig/generations.png')
print(quant_normal, quant_primeira, quant_nunca)
```

Resumidamente, o código 4 chama a função *main* com 5 diferentes taxas de cruzamento, definidas na variável *r_cross_opts*, e 4 diferentes taxas de mutação, armazenadas em *r_mut_opts*, e armazena os resultados em 4 diferentes vetores, cada um com um tipo de seleção e um tamanho populacional diferente. Por fim, esses resultados são plotados nos gráficos exibidos no próximo capítulo.

7 Resultados

Com apenas 2 minutos de processamento foi possível chegar aos resultados que serão discutidos neste capítulo. Começamos com a figura 6, ela mostra o nível extremo de aleatoriedade deste algoritmo e sua incrível capacidade de quase sempre convergir ao ponto correto em poucas iterações. As poucas população que não o fizeram, só não conseguiram por causa do limite imposto de apenas 25 iterações, mas é interessante notar que todas estavam muito próximas de chegar ao

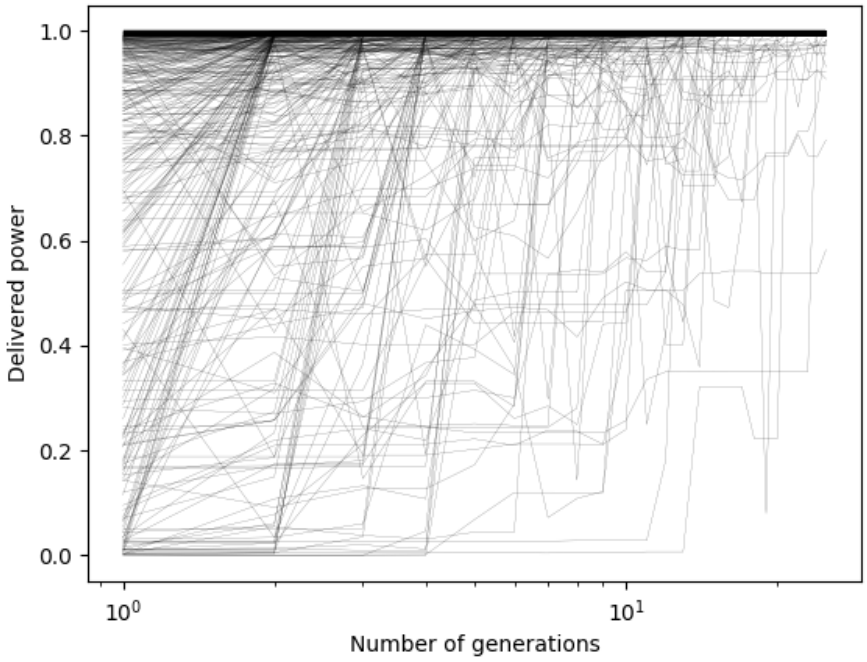

Figura 6: Melhores *fitness* de 400 simulações aleatórias diferentes com diferentes parâmetros.

resultado final.

A quantidade de simulações é igual ao produtório do número de:

- simulações para cada chamada da função *main*: 5;
- taxas de mutação: 4, são elas: 50%, 100%, 150%, 200% por indivíduo ou respectivamente 3,125%, 6,25%, 9,375% e 12,5% por bit;
- taxas de cruzamento: 5, são elas: 60%, 70%, 80%, 90% e 100%; e
- conjuntos de teste: 4, são eles: população de 4 indivíduos com seleção de roleta, população de 4 indivíduos com seleção de torneio, população de 8 indivíduos com seleção de roleta e população de 8 indivíduos com seleção de torneio.

Daí vem a quantidade de 400 simulações, das quais 389 convergiram, ou seja, é uma taxa de sucesso de 97,25%, mesmo com a extrema limitação de apenas 25 gerações.

Na figura 7 é possível visualizarmos o pior resultado das 5 simulações para diferentes tamanhos de população, taxas de cruzamento e mutação com cada um dos métodos de seleção. Dele podemos extrair algumas conclusões:

- a mais óbvia é que uma maior população permite uma convergência mais rápida, afinal existem mais possibilidades de encontrar uma solução se a quantidade de testes é maior;

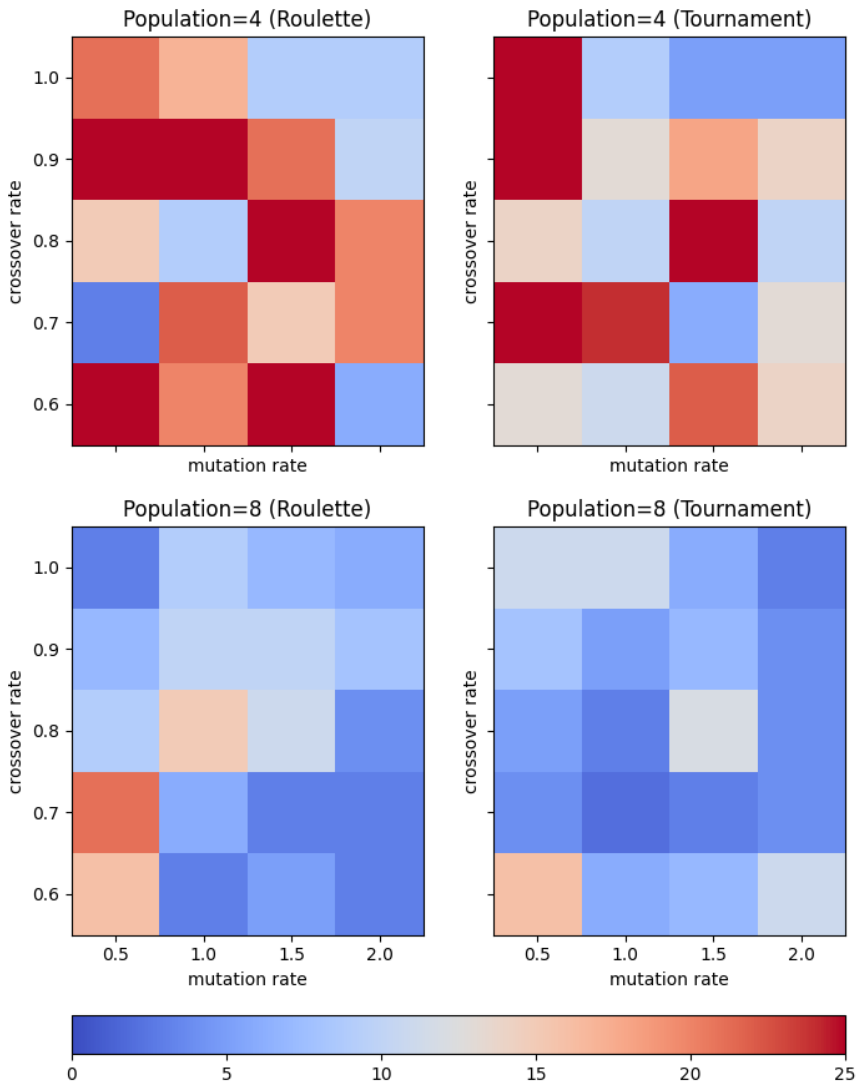

Figura 7: Quantidade máxima de gerações para convergir em cada uma dos 4 diferentes conjuntos de teste.

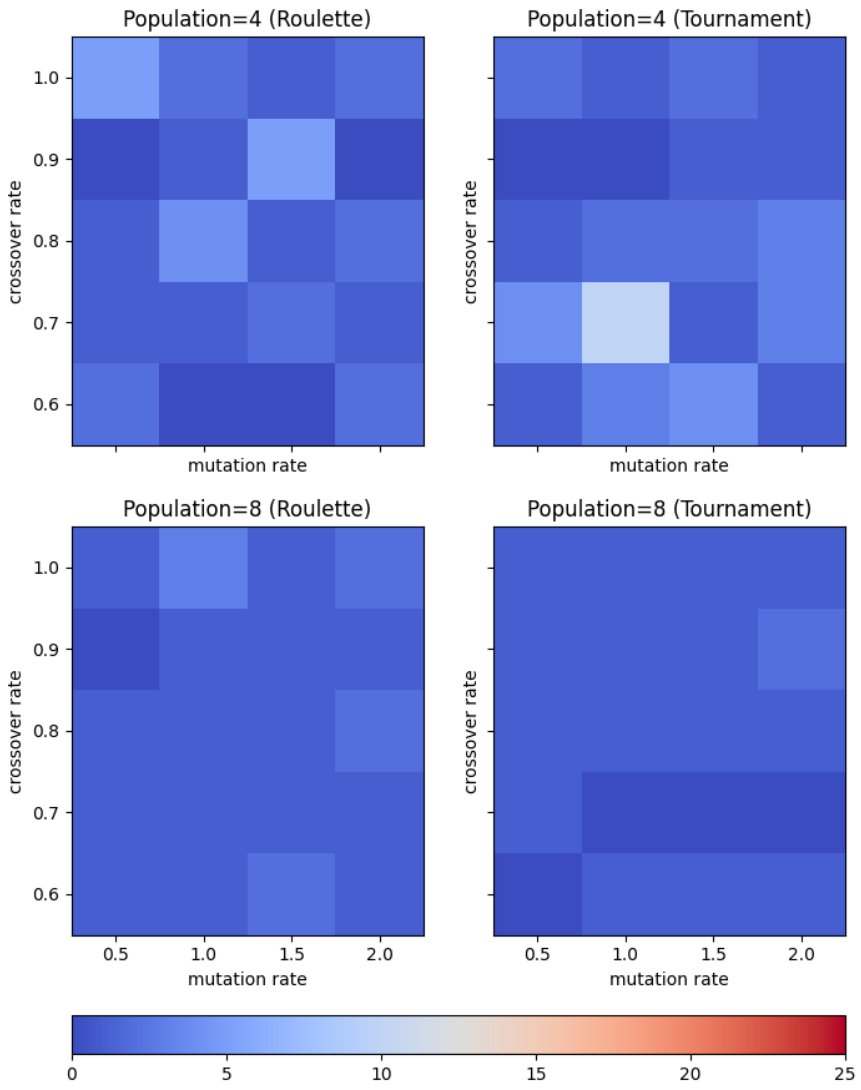

Figura 8: Quantidade mínima de gerações para convergir em cada uma dos 4 diferentes conjuntos de teste.

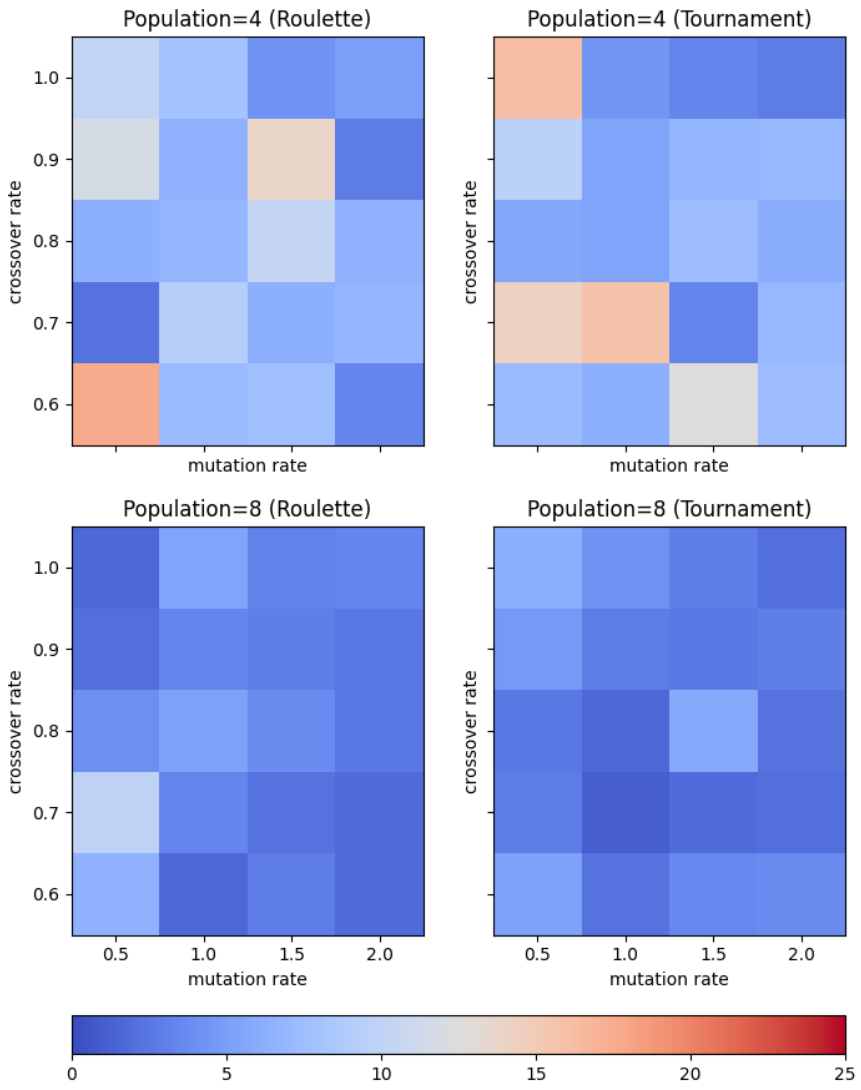

Figura 9: Quantidade média de gerações para convergir em cada uma dos 4 diferentes conjuntos de teste.

- taxas de cruzamento mais baixas levaram à convergência mais rápida na população de 8 indivíduos enquanto ocorre o contrário na população de 4 indivíduos, provavelmente devido à eliminação de indivíduos com bom *fitting*; e
- no geral, ambas formas de seleção produzem resultados consistentes, porém a vantagem de performance da seleção por torneio é um bom ponto a se levar em consideração na hora de escolher entre um e outro.

Na figura 8 é possível tirar a conclusão de que com qualquer um dos parâmetros adotados é possível chegar a um resultado satisfatório dependendo da sorte de ter uma boa população inicial.

Por fim, na figura 9 é possível confirmar o que a figura 6 já dava indícios: no geral foram necessários menos de 5 iterações para se chegar ao resultado esperado, ou seja, quase sempre com menos de 30 simulações foi possível chegar a um resultado que necessitaria milhares de simulações, mais de 65 mil, caso todos os valores possíveis fossem ser analisados por meio da força bruta, ou seja, uma redução média de 99,95% na quantidade de simulações necessárias para chegar à solução ótima.

8 Conclusão

Embora o problema a ser resolvido tenha sido relativamente simples, com ele foi possível aprender um pouco mais sobre como algoritmos genéticos podem ser usados para resolver problemas de engenharia de forma eficiente.

Referências

BENTO, E. P.; KAGAN, N. Algoritmos genéticos e variantes na solução de problemas de configuração de redes de distribuição. *Sba: Controle & Automação Sociedade Brasileira de Automatica*, v. 19, n. 3, p. 302–315, set. 2008. ISSN 0103-1759. Disponível em: <http://www.scielo.br/scielo.php?script=sci_arttext&pid=S0103-17592008000300006&lng=pt&tlng=pt>.

BROWNLEE, J. *Simple Genetic Algorithm From Scratch in Python*. 2021. Disponível em: <https://machinelearningmastery.com/simple-genetic-algorithm-from-scratch-in-python/>.