Inteligência Artificial para Robótica Móvel

Métodos de Otimização Baseados em População

Professor: Marcos Maximo

Roteiro

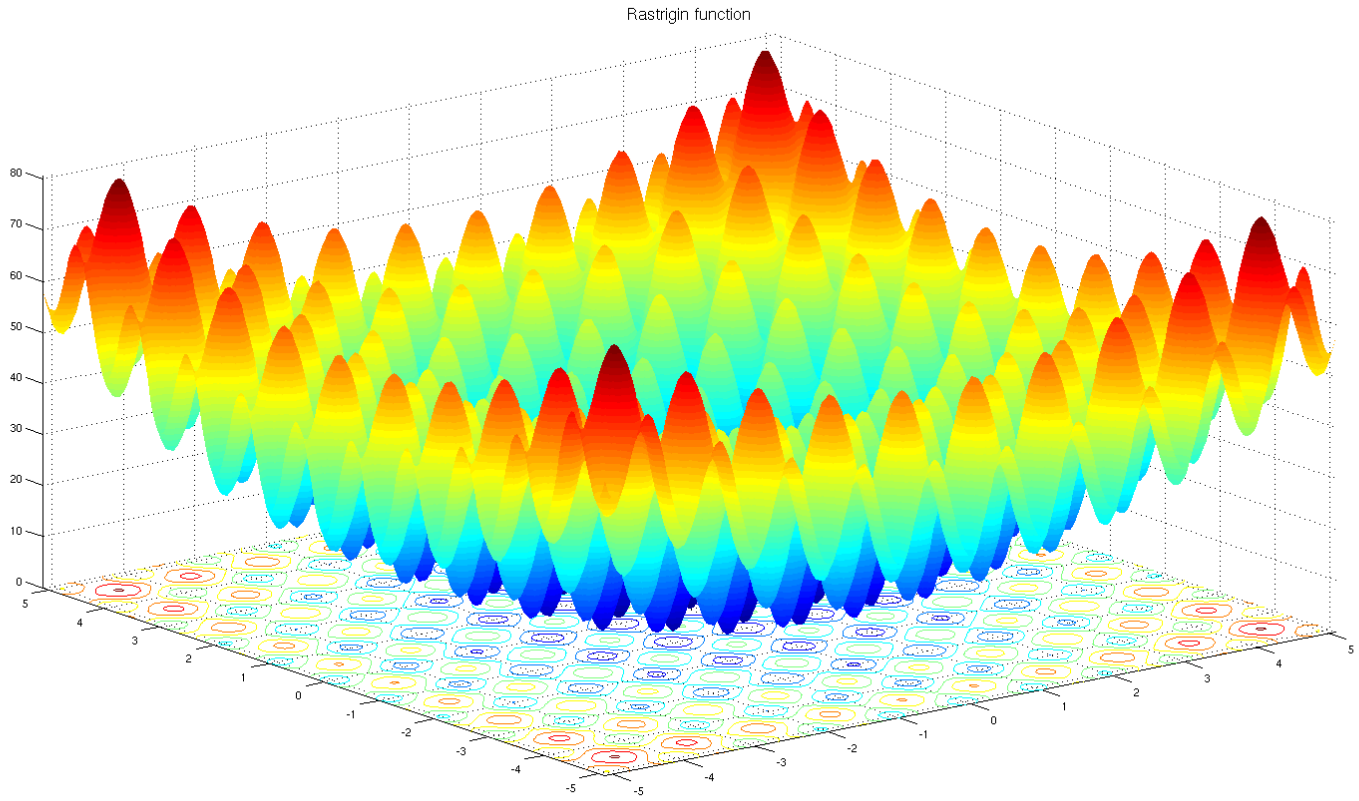
- Motivação.
- *Beam Search*.
- Método de Nelder-Mead.
- *Particle Swarm Optimization* (PSO).
- Algoritmos Genéticos.
- Estudo de Caso: Otimização de Caminhada de Robô Humanoide.

Motivação

Métodos Baseados em População

- Métodos de otimização vistos na aula anterior são muito “gulosos”, “exploitam” demais.
- Com isso, ficam facilmente presos em mínimos locais.
- Não funcionam bem para problemas “difíceis”, com muitos mínimos locais.
- Métodos focam apenas no **melhor** e *esquecem* que outras soluções próximas do ótimo também são promissoras.
- Ideia: manter população de soluções candidatas.
- Vantagem prática: costumam ser muito paralelizáveis.

Função de Rastrigin



Fonte: https://en.wikipedia.org/wiki/Rastrigin_function

Beam Search

Beam Search

- Português: Busca em Feixe Local (tradução do Norvig).
- Espécie de *Hill Climbing* usando população de P melhores.
- A cada iteração, adiciona todos os sucessores dos candidatos da população.
- Mas apenas o P melhores sobrevivem para a próxima (**sobrevivência dos mais aptos**).

Beam Search

```
# Assuming maximization
def beam_search(J, initial_population, population_size):
      population = initial_population
      while not check_stopping_condition():
            for candidate in population:
                  for neighbor in neighbors(candidate):
                        population.append(neighbor)
            population = sort_decreasing(population, J)
            population = population[0:population_size]
      return population[0]
```

Método de Nelder-Mead

Método de Nelder-Mead

- Outros nomes: *Downhill Simplex Method*, *Amoeba Method* etc.
- Método implementado na função `fminsearch` do MATLAB.
- Usa um *simplex* (população) de $n + 1$ pontos para um vetor $\mathbf{x}$ de dimensão n.

Método de Nelder-Mead (MATLAB)

Inicialização:

- Dado chute inicial $\mathbf{x}_0$, calcula demais pontos:

$$\mathbf{x}_i = \mathbf{x}_0, i = 0,1, \dots, n$$
$$\mathbf{x}_i(i) = \mathbf{x}_i(i) + 0.05 * \mathbf{x}_0(i)$$

- Se $\mathbf{x}_0 = \mathbf{0}$:

$$\mathbf{x}_i = \mathbf{0}$$
$$\mathbf{x}_i(i) = 0.00025$$

Método de Nelder-Mead (MATLAB)

Execução da iteração (minimização):

1. Reordenar pontos do *simplex* tal que: $J(\mathbf{x}_0) \leq J(\mathbf{x}_1) \leq \cdots \leq J(\mathbf{x}_n)$. Objetivo da iteração é substituir $\mathbf{x}_n$ (pior ponto).
2. *Refletir* o pior ponto:

$$\mathbf{r} = \mathbf{m} + (-1) * (\mathbf{x}_n - \mathbf{m})$$

em que $\mathbf{m}$ é o centro de massa dos n melhores: $\mathbf{m} = \sum_{i=0}^{n-1} \mathbf{x}_i \, /n$.

3. Se $J(\mathbf{x}_0) \leq J(r) \leq J(\mathbf{x}_{n-1})$, aceita r e termina a iteração (**reflexão**).
4. Se $J(\mathbf{r}) < J(\mathbf{x}_0)$, calcula **expansão**:

$$\mathbf{s} = \mathbf{m} + (-2) * (\mathbf{x}_n - \mathbf{m})$$

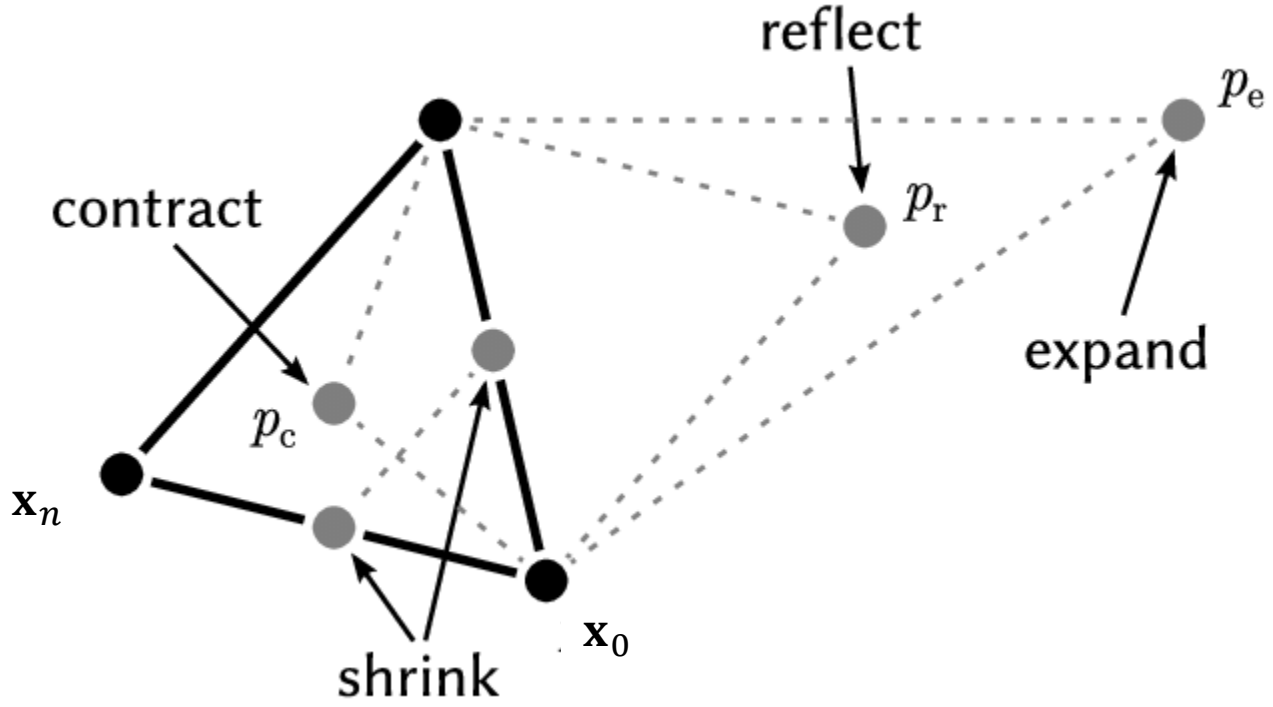
5. Se $J(\mathbf{s}) < J(\mathbf{r})$, aceita $\mathbf{s}$, caso contrário, aceita $\mathbf{r}$. Termina a iteração.

Método de Nelder-Mead (MATLAB)

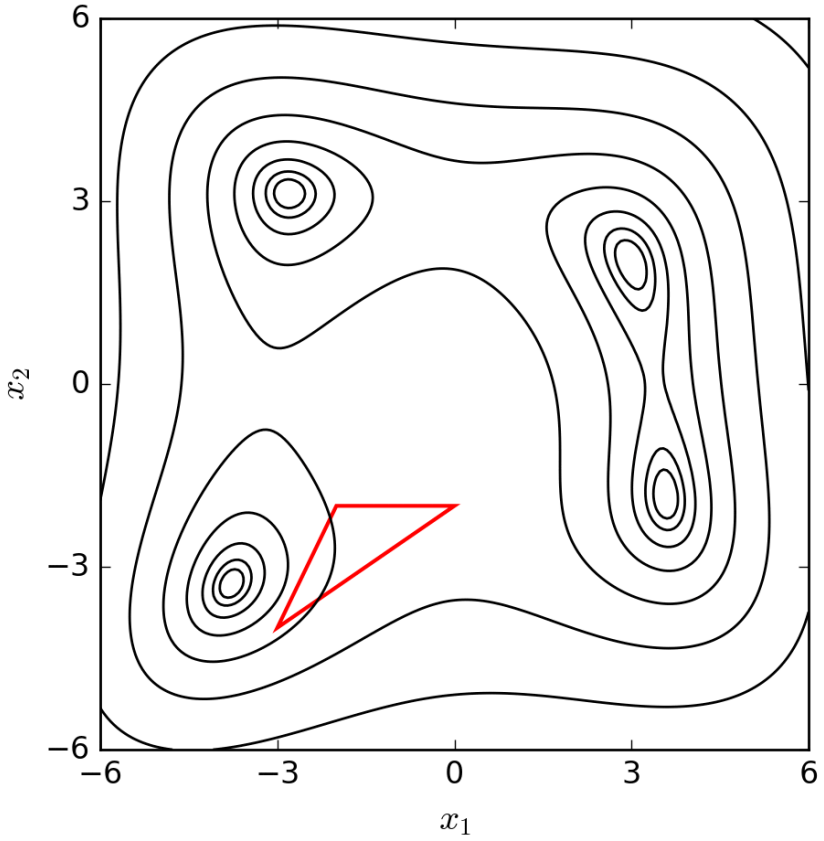
6. Se $J(\mathbf{r}) \geq J(\mathbf{x}_{n-1})$, fazer **contração** entre m e melhor entre r e $\mathbf{x}_n$.
   a. Se $J(\mathbf{r}) < J(\mathbf{x}_n)$: $\mathbf{c} = \mathbf{m} + (\mathbf{r} - \mathbf{m})/2$. Se $J(\mathbf{c}) < J(\mathbf{r})$, aceita $\mathbf{c}$ e termina iteração (**contract outside**). Caso contrário, ir para passo 7.
   b. Se $J(\mathbf{r}) \geq J(\mathbf{x}_n)$: $\mathbf{cc} = \mathbf{m} + (\mathbf{x}_n - \mathbf{m})/2$. Se $J(\mathbf{cc}) < J(\mathbf{x}_n)$, aceita $\mathbf{cc}$ e termina iteração (**contract inside**). Caso contrário, ir para passo 7.
7. Contrair todos os pontos no *simplex*:

$$\mathbf{x}_i = \mathbf{x}_0 + \frac{\mathbf{x}_i - \mathbf{x}_0}{2}$$

Método de Nelder-Mead



Método de Nelder-Mead



Fonte: https://en.wikipedia.org/wiki/Nelder%E2%80%93Mead_method#/media/File:Nelder-Mead_Himmelblau.gif

Método de Nelder-Mead

- Funciona muito bem para até 2-3 parâmetros.
- Muito dependente do chute inicial.
- Pessoalmente, uso muito para projeto de ganhos de controlador clássico (estilo PID) ou de *fit* (com função complicada).
- Em Controle, encontro chute inicial resolvendo versão simplificada do problema (sem atrasos, sem não-linearidades, ignorando dinâmicas mais rápidas etc.) usando técnicas clássicas de projeto.

Particle Swarm Optimization (PSO)

Particle Swarm Optimization (PSO)

- Português: Otimização do Enxame de Partículas.
- Inspirado no movimento de migração dos pássaros. ♣ ♦ ♥ ♠
- Partícula: candidato à solução.
- Enxame: população.
- Número de partículas é hiperparâmetro.

Particle Swarm Optimization (PSO)

- Memorizar melhor posição de cada partícula $\mathbf{b}_i$.
- Memorizar melhor posição considerando todas as partículas $\mathbf{b}_g$.
- Atualização da velocidade da partícula:

$$\mathbf{v}_i = \omega \mathbf{v}_i + \varphi_p r_p (\mathbf{b}_i - \mathbf{x}_i) + \varphi_g r_g (\mathbf{b}_g - \mathbf{x}_i)$$

inertia weight *cognitive parameter* *social parameter*

$$r_p, r_g \sim U([0,1])$$

- Ideia: “empurrar” partículas na direção do “melhor”.
- Atualização da posição:

$$\mathbf{x}_i = \mathbf{x}_i + \mathbf{v}_i$$

Particle Swarm Optimization (PSO)

- Inicialização das partículas:
- Considere limites para cada dimensão:

$$\mathbf{l} \leq \mathbf{x} \leq \mathbf{u}$$

$$\mathbf{x}_i \sim U([\mathbf{l}, \mathbf{u}])$$

$$\mathbf{v}_i \sim U([-(\mathbf{u} - \mathbf{l}), (\mathbf{u} - \mathbf{l})])$$

Particle Swarm Optimization (PSO)

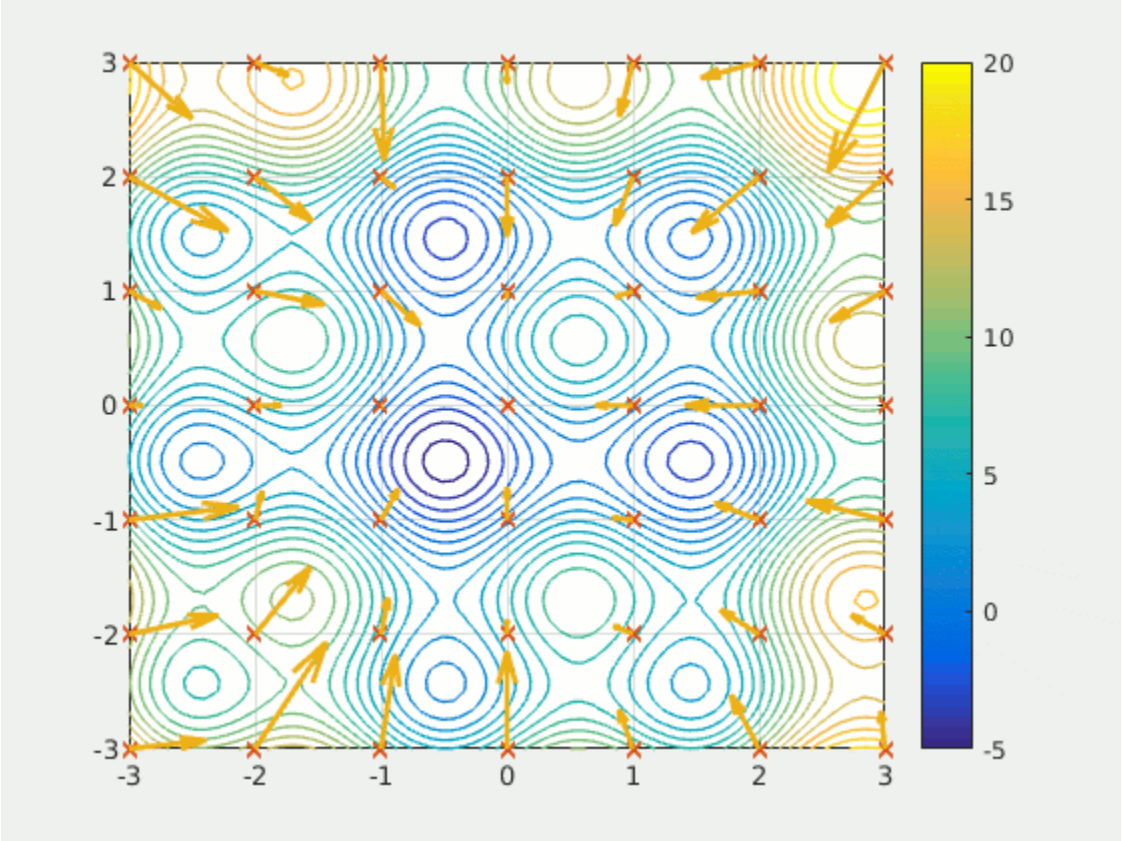
- É comum usar heurísticas para deixar posição e velocidade dentro de limites.
- A heurística mais simples é limitar posição e velocidade após o cálculo na iteração:

$$\mathbf{x}_i = \min(\max(\mathbf{x}_i, \mathbf{x}_{min}), \mathbf{x}_{max})$$
$$\mathbf{v}_i = \min(\max(\mathbf{v}_i, \mathbf{v}_{min}), \mathbf{v}_{max})$$

- Escolha comum para limites de velocidade:

$$\mathbf{v}_{min} = -(\mathbf{u} - \mathbf{l}), \mathbf{v}_{max} = (\mathbf{u} - \mathbf{l})$$

Particle Swarm Optimization (PSO)



Particle Swarm Optimization (PSO)

```
# Assuming minimization
def pso(J, hyperparams):
        particles = initialize_particles(hyperparams.num_particles,
                                         hyperparams.lb, hyperparams.ub)
        best_global = None # J(None) = inf
        while not check_stopping_condition():
                particles, best_iteration = update_particles(particles,
                                                 best_global, hyperparams)
                if J(best_iteration) < J(best_global):
                        best_global = best_iteration
        return best_global
```

Particle Swarm Optimization (PSO)

```
def initialize_particles(num_particles, lb, ub):
     particles = Particle[num_particles]
     for i in range(len(particles)):
          # random_uniform here operates on arrays
          particles[i].x = random_uniform(lb, ub)
          delta = ub – lb
          particles[i].v = random_uniform(-delta,
                                          delta)
```

Particle Swarm Optimization (PSO)

```
def update_particles(particles, best_global, hyperparams):
        w = hyperparams.w
        phip = hyperparams.phip
        phig = hyperparams.phig
        best_iteration = None
        for particle in particles:
                rp = random_uniform(0.0, 1.0)
                rg = random_uniform(0.0, 1.0)
                particle.v = w * particle.v + phip * rp * (particle.best – particle.x) +
                                                    phig * rg * (best_global – particle.x)
                particle.x = particle.x + particle.v
                if J(particle.x) < J(particle.best):
                        particle.best = particle.x
                        if J(particle.x) < J(best_iteration):
                                best_iteration = particle.x
```

Dicas para os Hiperparâmetros

- Trocamos chutar uns parâmetros por outros ☺.
- Recomendações baseadas em experiência **pessoal**.
- Número de partículas: usar muitas para ter mais “variedade”. Recomendação de 40 a 50.
- Mais parâmetros = mais partículas.
- $\omega < 1$.
- Tem gente que usa *schedule* no ω (e.g. $\omega = \frac{\omega_0}{1+\beta k}$).
- $\varphi_g > \varphi_p$.
- Costumava usar $\omega = 0{,}9$, $\varphi_p = 0{,}6$ e $\varphi_g = 0{,}8$.
- ω, φ_p e φ_g realizam *trade-off* entre *explotation* e *exploration*.

Um problema do PSO

- Como já falado, em problemas de robótica $J(\mathbf{x})$ é estocástico.
- Às vezes, uma posição não muito boa dá sorte e obtém boa avaliação.

$$\mathbf{v}_i = \omega\mathbf{v}_i + \varphi_p r_p(\mathbf{b}_i - \mathbf{x}_i) + \varphi_g r_g(\mathbf{b}_g - \mathbf{x}_i)$$

- Enxame acaba convergindo para solução pouco robusta.

Algoritmos Genéticos

Algoritmos Genéticos

- Inglês: *Genetic Algorithms*.
- Por que no plural? **Muitas** variações...
- Baseados na Teoria da Evolução de Darwin. ♣ ♦ ♥ ♠
- Em geral, trabalha-se com maximização.

Algoritmos Genéticos

- Cromossomo: candidato à solução.
- Gene: uma parte do cromossomo (e.g. um bit ou uma dimensão).
- População: conjunto de candidatos.
- Geração: população na iteração.
- Função de *fitness* (aptidão).
- Mutação: altera um candidato aleatoriamente.
- Seleção: a cada geração, os mais aptos tem mais chance de se reproduzir.
- *Crossover*: filho é “mistura” dos pais (reprodução sexuada).
- Sobrevivência dos mais aptos: apenas os melhores passam para a próxima geração.

Cromossomo

- Considere problema com $\mathbf{x} \in \mathbb{R}^4$. Exemplo de cromossomo:

x_1	x_2	x_3	x_4
23.5	12.0	42.0	65.2

- Alguns preferem codificar direto em binário:

x_1	x_2	x_3	x_4
10110011	00101101	11110000	00010101

- Na codificação com real, cada x_i é um gene.
- Na codificação em binário, cada bit é um gene.

Mutação

- Com probabilidade de mutação p_m, altera um gene aleatório do cromossomo.

7.5	23.11	95.54	100.0	12.0	82.21	42.0	35.91

7.5	23.11	95.54	1.2	12.0	82.21	42.0	35.91

- Duas formas de implementar na Literatura:
  - p_m é probabilidade de ocorrer mutação no cromosso. Então, gene é escolhido aleatoriamente.
  - p_m é probabilidade de **cada** gene sofrer mutação.
- Alteração no cromossomo pode ser menos drástica:
  - Incrementar/decrementar gene de um pequeno valor aleatório.

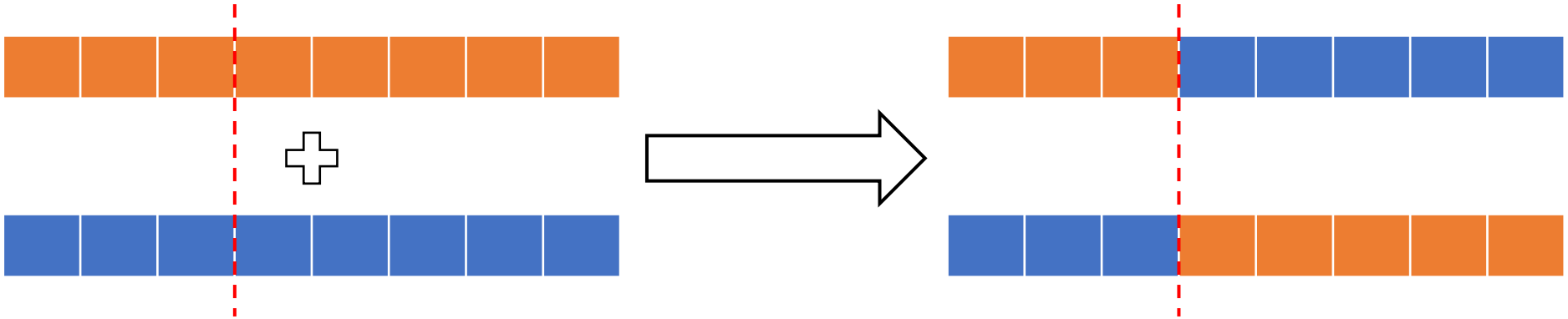
Seleção

- Vários esquemas, veremos **seleção por roleta**.
- Escolha de cada indivíduo proporcional à sua aptidão:

$$p_i = \frac{J(\mathbf{x}_i)}{\sum_j J(\mathbf{x}_j)}$$

- Em geral, usa-se escolha com reposição (i.e. pai pode ser escolhido novamente, inclusive pode procriar consigo mesmo).
- Número de pais escolhidos para procriar é hiperparâmetro.

Crossover



- Ponto de quebra escolhido aleatoriamente.
- Pode-se definir probabilidade de *crossover* p_c.
- Quando não acontece *crossover*, filhos são cópias dos pais.
- Também é possível usar vários pontos de quebra.

Sobrevivência dos Mais Aptos

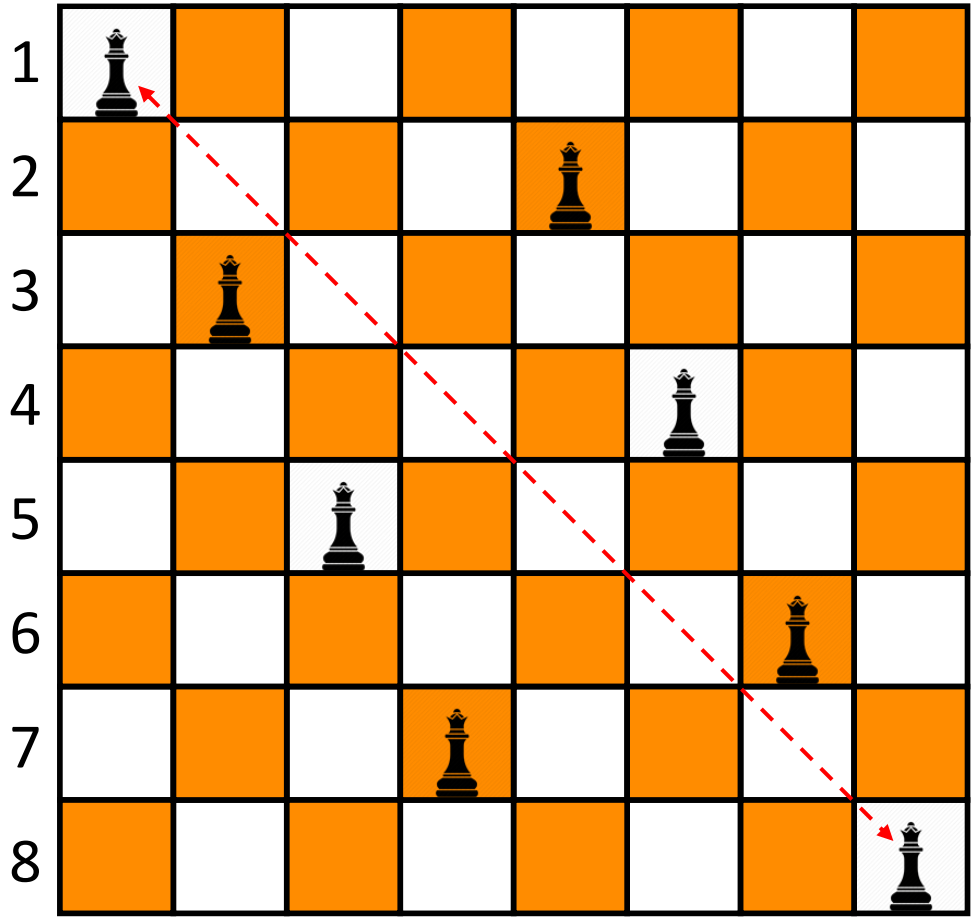
- Forma mais simples: define-se tamanho máximo da população P, então mantém-se o P melhores e mata-se os demais.
- Implementação: ordena e mantém os P melhores.
- Também há esquemas que usam probabilidade.

Exemplo: 8 Rainhas

- Exemplo: problema das 8 rainhas.
- Obter configuração do tabuleiro em que nenhum par de rainhas se ataca.
- Codificação:

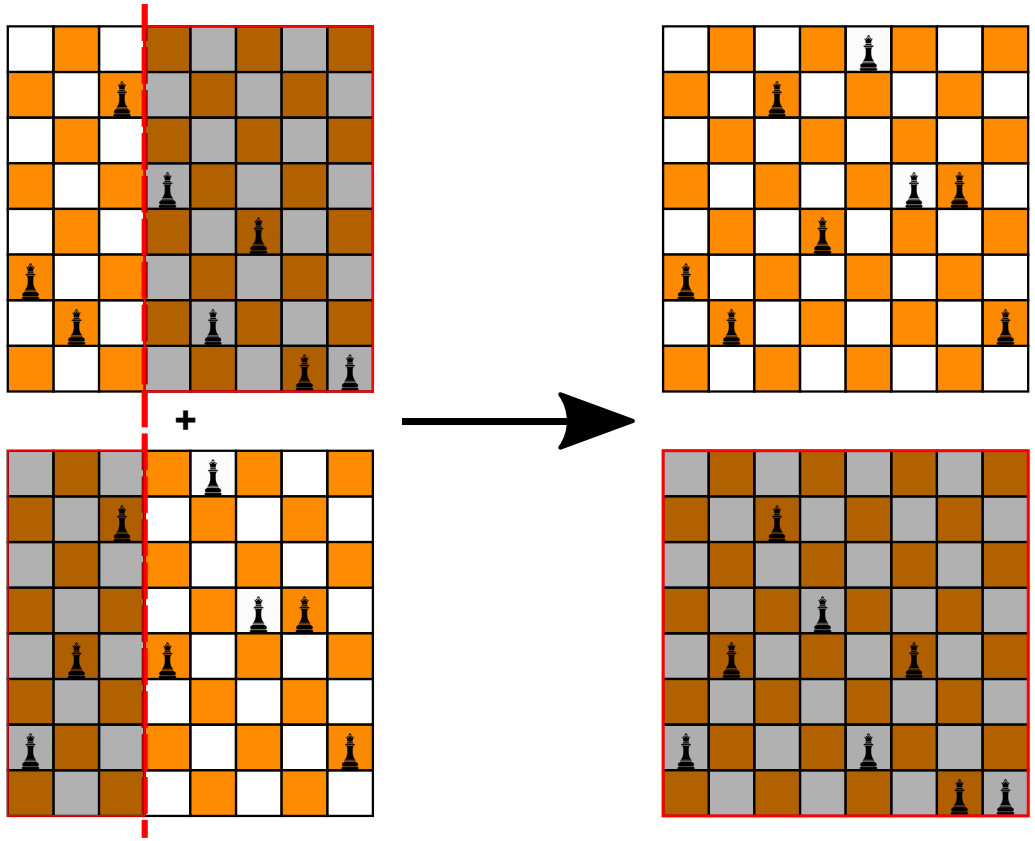
1	3	5	7	2	4	6	8

- Função de aptidão: número de pares que não estão se atacando. Na figura: 27.



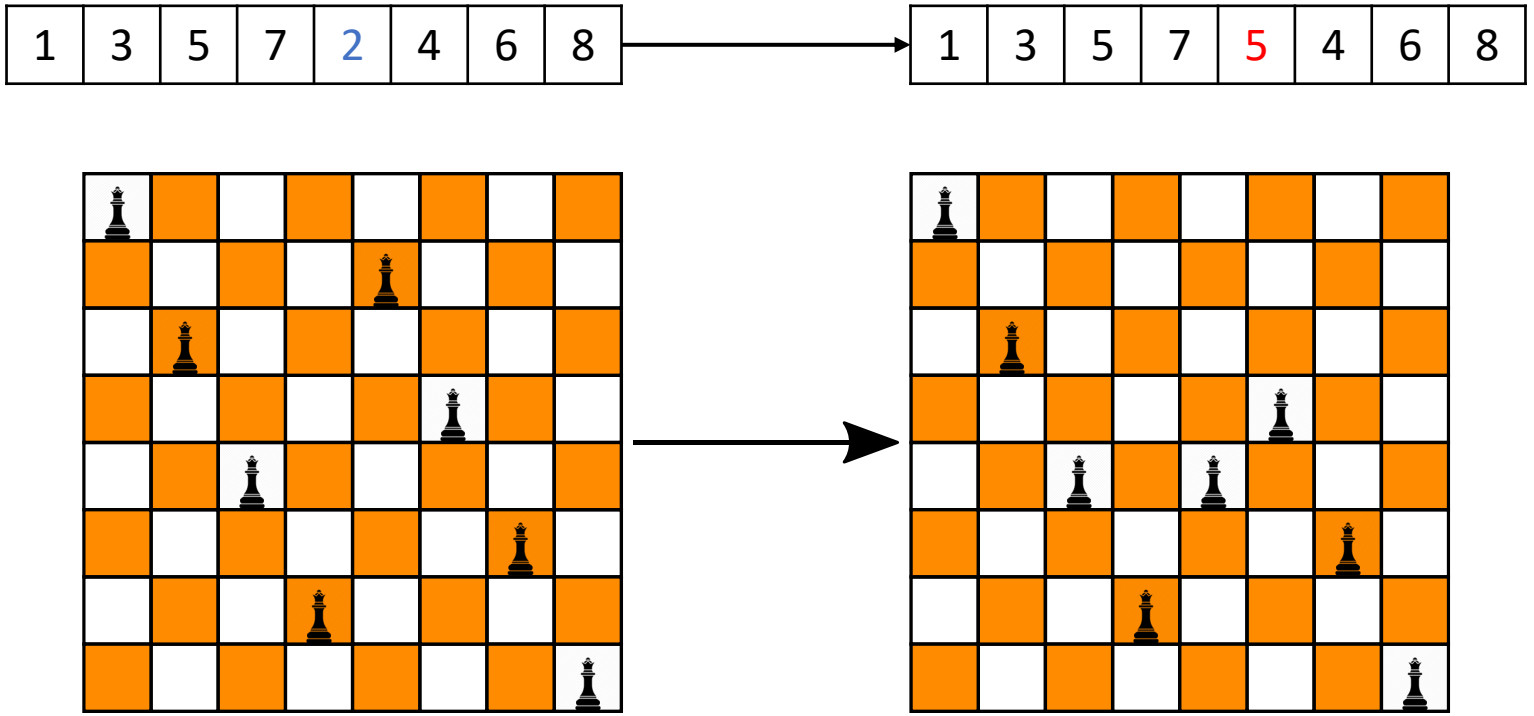
Exemplo: 8 Rainhas

- *Crossover*:



Exemplo: 8 Rainhas

- Mutação:

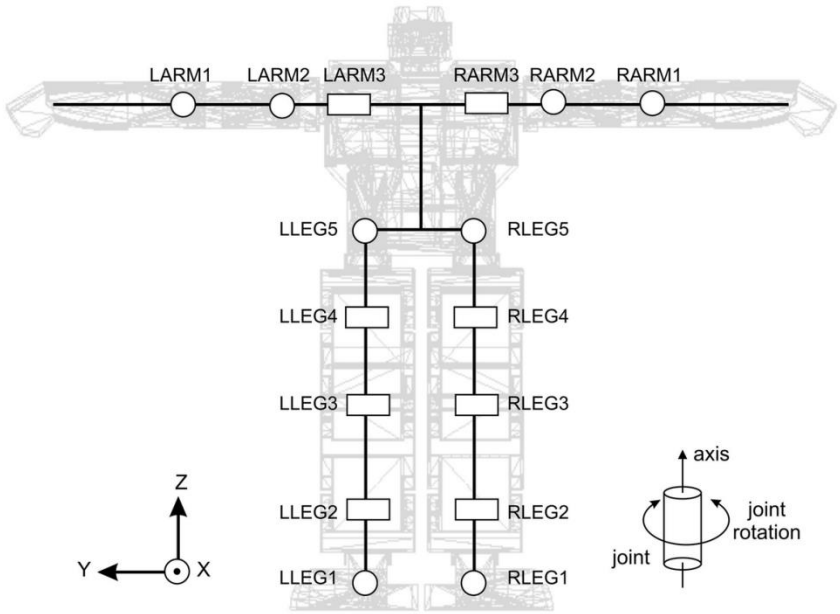


Algoritmos Genéticos

```
def genetic_algorithm(J, hyperparams):
      pop_size, pm, num_parents = unwrap_hyperparams(hyperparams)
      population = random_population(pop_size)
      fitnesses = evaluate(population, J)
      while not check_stopping_condition():
            parents = selection(population, fitnesses, num_parents)
            children = crossover(parents)
            population = parents U children
            population = mutation(population, pm)
            fitnesses = evaluate(population, J)
            population = survival(population, fitnesses, pop_size)
      return select_best(population)
```

Estudo de Caso: Otimização de Caminhada de Robô Humanoide

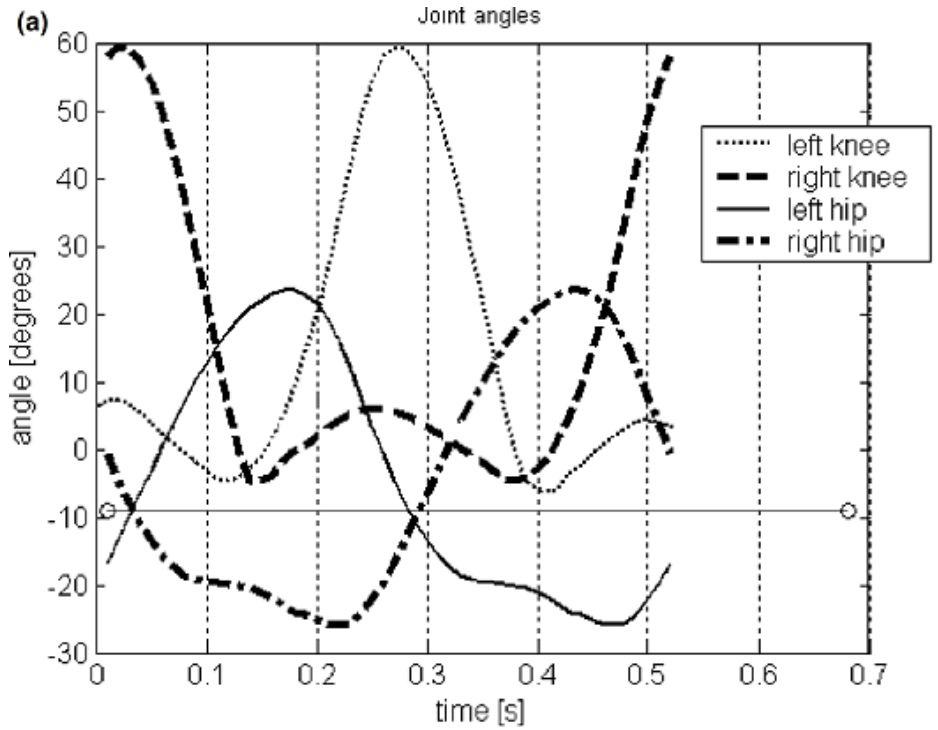
Problema

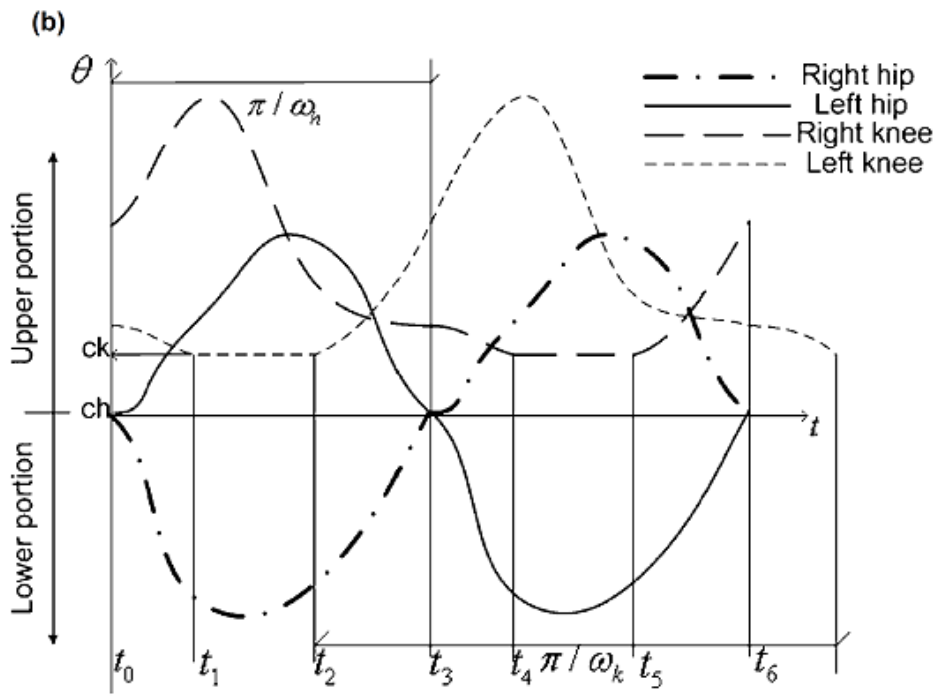


16 juntas!

Modelo de Caminhada

- Observando caminhada humana...

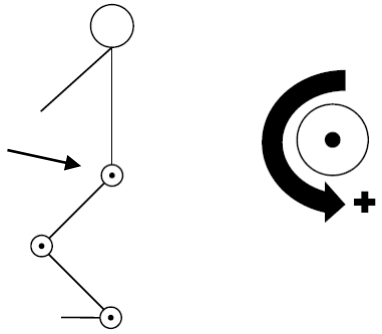




Modelo de Caminhada (Shafii et al, 2010)

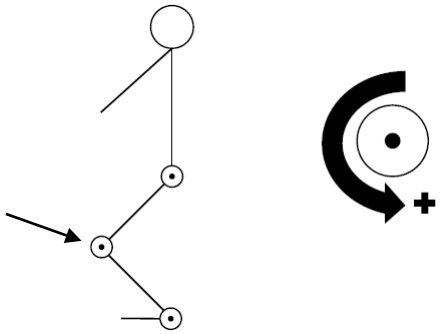
- Coxa (esquerda):

$$\theta_c(t) = \begin{cases} O_c + A\sin\left(\frac{2\pi t}{T}\right), t \in \left[0, \frac{T}{2}\right) + kT, k \in \mathbb{Z} \\ O_c + B\sin\left(\frac{2\pi t}{T}\right), t \in \left[\frac{T}{2}, T\right) + kT, k \in \mathbb{Z} \end{cases}$$



- Joelho (esquerdo):

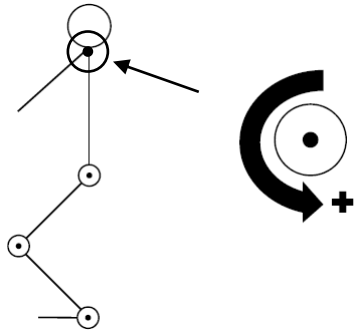
$$\theta_j(t) = \begin{cases} O_j + C\sin\left(\frac{2\pi(t - t_2)}{T}\right), t \in \left[0, \frac{T}{2}\right) + kT, k \in \mathbb{Z} \\ O_j, t \in \left[\frac{T}{2}, T\right) + kT, k \in \mathbb{Z} \end{cases}$$



Movimento de Braços (adaptado de Shafii et al, 2009)

- Ombro:

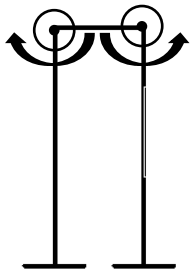
$$\theta_o(t) = \begin{cases} -D\_sin\left(\frac{2\pi t}{T}\right), t \in \left[0, \frac{T}{2}\right) + kT, k \in \mathbb{Z} \\ -D_+ sin\left(\frac{2\pi t}{T}\right), t \in \left[\frac{T}{2}, T\right) + kT, k \in \mathbb{Z} \end{cases}$$



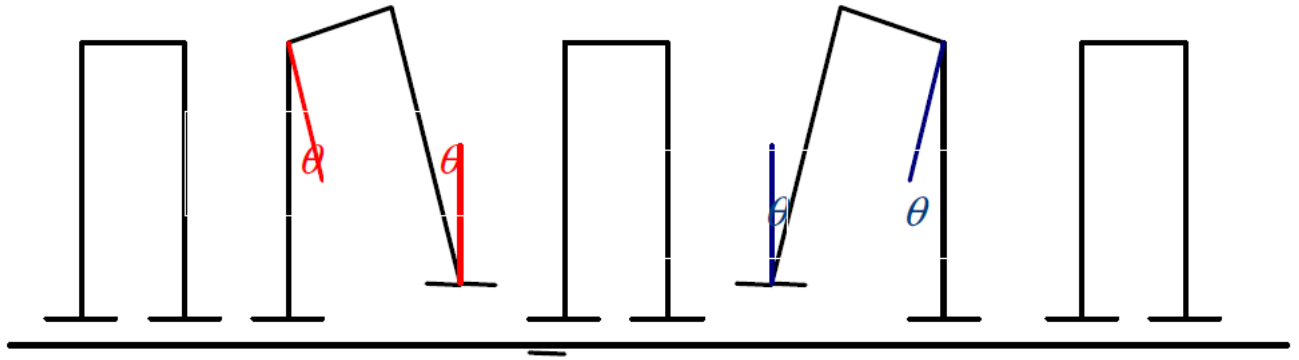
Movimento Coronal (Shafii et al, 2010)

- Coxa (plano coronal):

$$\theta_l(t) = \begin{cases} Esin\left(\frac{2\pi t}{T}\right), t \in \left[0, \frac{T}{2}\right) + kT, k \in \mathbb{Z} \\ 0, t \in \left[\frac{T}{2}, T\right) + kT, k \in \mathbb{Z} \end{cases}$$



- Sequência de movimentos:



Abordagem

- Total de 10 parâmetros para serem ajustados!
  - Trabalhoso ajustar testando “no braço”.
  - Usar algoritmos de otimização.
- Problema: robô vai quebrar antes de aprender a caminhar!
  - Usar simulação.

Simulação

Algoritmos de Otimização

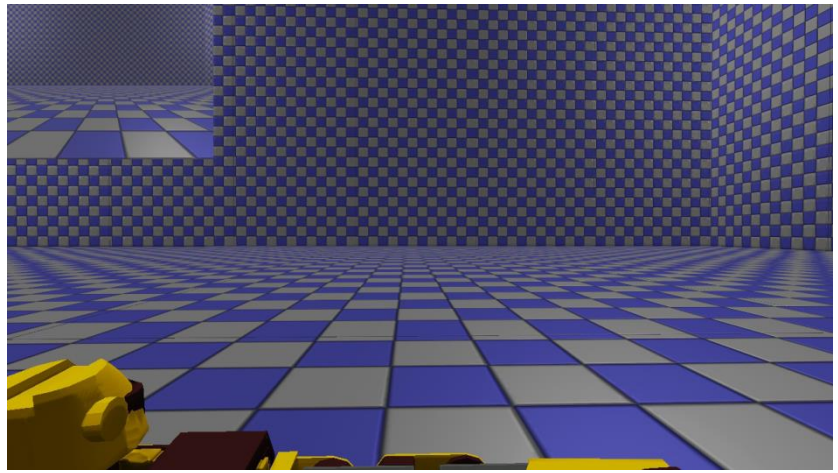
- *Particle Swarm Optimization* (PSO).
- Algoritmo Genético.

Processo de Otimização

- Experimento:

1. Iniciar robô.
2. Esperar robô andar 20 segundos ou cair.
3. Calcular desempenho.

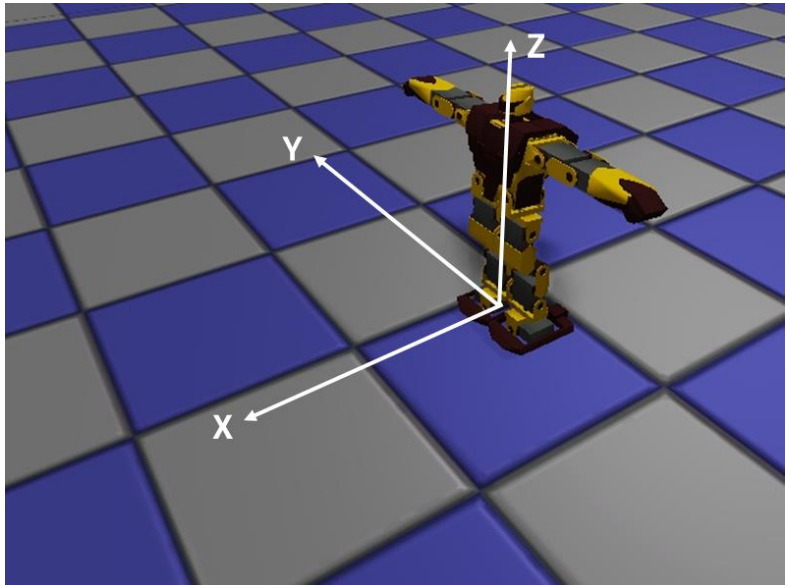
- Mapa usado:



Processo de Otimização

- Medida de desempenho (maximização):

$$D = (x - x_o) - |y - y_o| + 0{,}1 \times \Delta t - \sum P_i$$



Punição	Significado	Valor
P_1	Queda	50
P_2	Posição inicial instável	80
P_3	Não se moveu	60

Resultado (simulação)

Transferência para Robô Real

- Não funcionou de primeira.
- Ajustes “no braço” ☺. ♣ ♦ ♥ ♠
- Trabalho de ajuste certamente muito menor do que se tivesse começado do zero.

Resultado (robô real)

Resultados da Otimização

- Genético encontrou soluções melhores que PSO.
- Usei apenas meu computador pessoal.
- Simulador (*Unreal*) não permitia rodar mais rápido que tempo real.
- Assim, cálculo de $J(\mathbf{x})$ podia demorar até 20 s.
- Genético nunca chegou a convergir.
- PSO convergia em cerca de 6h… 8h...
- Claro que esses resultados dependem dos hiperparâmetros.

Experiência Prática

- Na prática, função de qualidade requer tentativa e erro.
- No começo, só usava queda como punição: diversas posições iniciais instáveis eram “bem avaliadas” porque robô se jogava para frente.
- Depois, robô aprendia a marcar passo para não cair, daí punição por “não se mover”.
- Otimização **maximiza o bizu**.

Punição	Significado	Valor
P_1	Queda	50
P_2	Posição inicial instável	80
P_3	Não se moveu	60

Experiência Prática

- Problema do PSO: uma caminhada rápida, mas pouco estável na sorte recebia pontuação muito alta.
- Fazia PSO convergir para solução pouco estável.
- Usava média de 3, mas problema ainda acontecia.
- É importante salvar estado da otimização: computador pode desligar.
- Salvar histórico também é interessante.

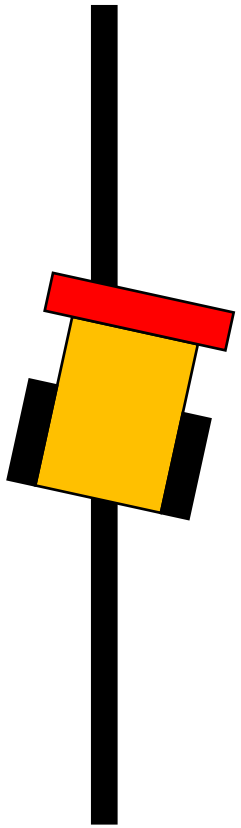
Para Saber Mais

- *Beam search* e Algoritmo Genético: capítulo 4 do livro Inteligência Artificial (2ª edição) de Russell & Norvig.
- PSO:
  - Wikipedia
  - *Particle Swarm Optimization for Single Continuous Space Problems: A Review* - https://www.mitpressjournals.org/doi/10.1162/EVCO_r_00180
- Nelder-Mead: documentação da função `fminsearch` do MATLAB.
- Estudo de caso:

Stable and fast model-free walk with arms movement for humanoid robots. MROA Maximo, EL Colombini, CHC Ribeiro. International Journal of Advanced Robotic Systems.

Laboratório 4

Laboratório 4



- Se a linha estiver à esquerda, girar para a esquerda.
- Se estiver à direita, girar para a direita.
- Quanto mais distante do centro, girar com mais intensidade.

Laboratório 4

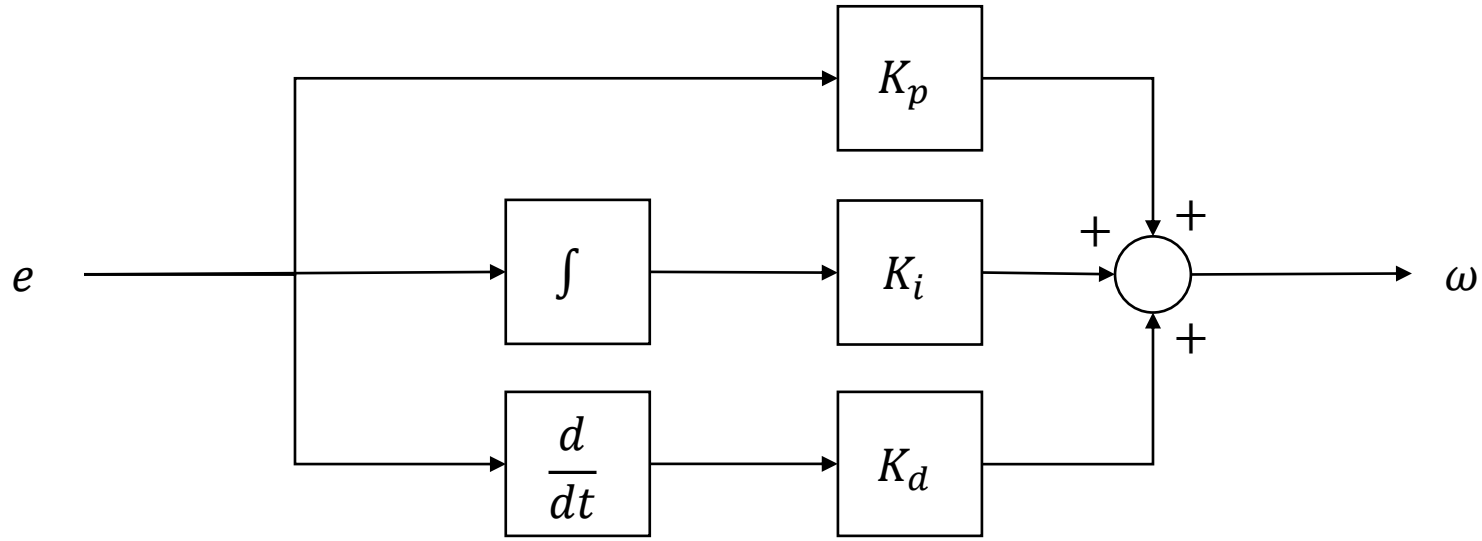
- Otimizar controlador de robô seguidor de linha.
- Uso de *Particle Swarm Optimization* (PSO).
- Robô possui *array* de 7 sensores para detectar linha.
- Erro da linha calculada como centro de massa das medidas:

$$e = \frac{\sum_i x_i I_i}{\sum_i I_i}$$

- Estratégia de controle:
  - Velocidade linear constante.
  - PID para seguir linha.

Laboratório 4

- Controlador PID:



- P: proporcional, I: integrativo, D: derivativo.

Laboratório 4

- Intuição de PID:
  - P: deixa mais rápido.
  - D: reduz oscilações (amortece).
  - I: remove erro em regime (robô faz curvas mais centralizado na linha).