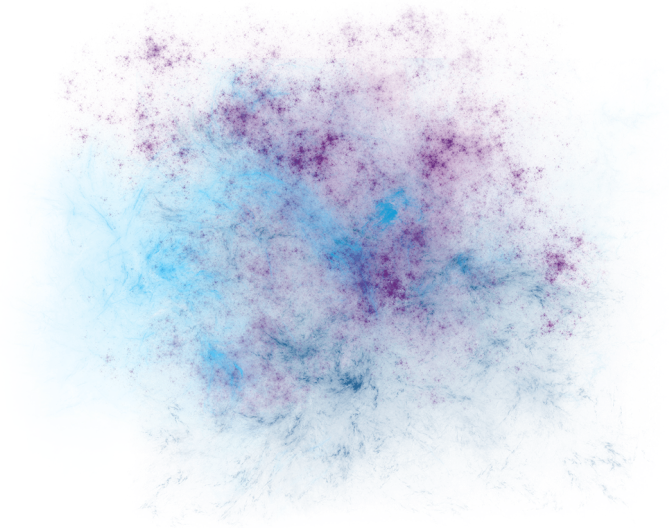
ANTÔNIO CARLOS DURÃES DA SILVA
JOEL WILL BELMIRO

GitHub: https://github.com/duraes-antonio/IA_trab5_FP

INTELIGÊNCIA ARTIFICIAL – 5º TRABALHO:
APLICAÇÃO DO ALGORÍTIMO DE FILTRO DE PARTÍCULAS NO RASTREAMENTO DE OBJETOS



1 EXPLICAÇÃO TEÓRICA DO ALGORITMO

Os algorítimos de filtros de partículas pertencem a classe de métodos Sequenciais de Monte Carlo. O uso desses filtros pertence principalmente à atividade de estimar o estado de um elemento em determinado instante. Por tratar de estimativas e aproximações, seu custo computacional é inferior a outros métodos que trabalham com resultados determinados e são, portanto, mais inflexíveis e até mais custosos, como as avariações do filtro de Kalman [1].

2 PROBLEMA PROPOSTO

A aplicação escolhida para teste do algoritmo foi o rastreamento de um objeto em um vídeo, mais especificamente, uma bola de basquete.

O problema resume-se, essencialmente, na distinção do objeto-alvo de outros itens presentes no cenário, na captura do máximo possível da área do corpo do objeto

para que seja possível calcular as coordenadas de seu centro de massa com precisão satisfatória.

3 IMPLEMENTAÇÃO

Para melhor divisão de tarefas, organização e consequente qualidade do código, a implementação foi particionada em 3 arquivos principais:

- “main.py”: A aplicação, responsável por capturar e validar brevemente os argumentos de entrada (número de partículas, velocidade, caminho do vídeo, etc), por chamar as funções do opencv para filtro de imagens e por coordenar como o resultado do algorítimo será exibido (cores, box, contorno)
- “util_imagem_opencv.py”: Um conjunto de funções para manipulação de imagem (blur, conversão RGB para HSV, cálculo do centro de massa) e aplicação de filtros do Opencv
- “particula.py”: Classe responsável por conter o filtro de partículas e seus métodos.

O foco do detalhamento da implementação será nos métodos e modelagem do algorítimo de filtro de partículas.

3.1 GRUPO DE PARTÍCULAS

Modelagem: O grupo de partículas é um array contendo N sub-arrays com x e y, representando uma partícula. Os pesos estão em um array a parte. Essa modelagem simples, com pouca orientação a objeto foi utilizada com intenção de maximizar ao máximo o desempenho do algoritmo, evitando travamentos durante a exibição do vídeo.

Instanciação: Para instanciar um grupo de partículas é necessário passar como argumentos: O número desejado, a velocidade mínima, a velocidade máxima, a posição máxima para X e para Y, além de um centro de massa inicial (que é opcional, se nada for passado, é usado a coordenada [0, 0]).

```
def __init__(
      self, n: int, v_min: float, v_max: float, x_max: int, y_max: int,
      centro: Tuple[int, int] = (0, 0)):

   self.__centro_ant: Tuple[int, int] = centro
   self.__vmin = v_min
   self.__vmax = v_max
   self.__n = n

   # Inicialize os N pesos (1 p/ cada partícula) com valor 1
   self.__pesos = array([1.0] * n)

   # Inicialize N partículas, cada uma com dois eixos
   self.__partics = empty((n, 2))

   # Sorteie N partículas, cada uma composta por um valor randômico de
   # x [0 até x_max] e randômico de y [0 até y_max]
   self.__partics[:, 0] = uniform(low=0, high=x_max, size=n)
   self.__partics[:, 1] = uniform(low=0, high=y_max, size=n)
```

3.2 PREDIÇÃO

A etapa de predição trabalha com o centro de massa do objeto no instante atual (t) e também no instante anterior (t-1). A ideia básica é calcular a distância entre a posição atual do objeto e sua posição anterior, e somar tal valor com uma velocidade randômica entre o mínimo e máximo (o uso de ruídos randômicos também foram utilizados).

Para encontrar a diferença em graus entre a antiga e a nova posição do objeto, a função arco tangente foi utilizada, também empregou-se o uso das funções cosseno e seno, a primeira para cálculo do incremento do eixo x, e a segunda para incremento de y.

```
def __predicao(
      self, c_massa_ant: Tuple[int, int], c_massa_atual: Tuple[int, int]):

   # Calcule a dist. euclidiana entre o antigo e o novo centro de massa
   dist = linalg.norm(array([c_massa_ant]) - array([c_massa_atual]), axis=1)

   # Sorteie a velocidade entre o mínimo e o máximo
   desvios = array(
      [uniform(self.__vmin, self.__vmax + 1) for i in range(self.n)])

   dist_com_ruido = dist + (randn(self.n) * desvios)

   heading = arctan2(
      array([c_massa_atual[1] - c_massa_ant[1]]),
      array([c_massa_ant[0] - c_massa_atual[0]]))

   if heading > 0:
      heading = -(heading - pi)
   else:
      heading = -(pi + heading)

   # Incremente as posições
   self.__partics[:, 0] += cos(heading) * dist_com_ruido
   self.__partics[:, 1] += sin(heading) * dist_com_ruido
   return None
```

3.3 ATUALIZAÇÃO

Nesta etapa, os pesos das partículas são calculados e normalizados com base na distância de cada partícula em relação ao centro de massa atual do objeto. Após ter essa distância armazenada, utiliza-se uma função de densidade de probabilidade para atribuir peso a cada partícula.

Todas as partículas encontram-se com peso igual a 1. Para um desvio theta (neste caso, igual a 20, podendo ser qualquer outro valor), o peso da partícula será multiplicado pela chance dela possuir distância zero até o centro de massa do objeto, isto é, quanto mais próxima do objeto, maior será o peso da partícula.

```
def __atualizacao(self, c_massa_atual: array):

    self.__pesos.fill(1)

    diff_x = (self.__partics[:, 0] - c_massa_atual[0]) ** 2
    diff_y = (self.__partics[:, 1] - c_massa_atual[1]) ** 2
    dist = (diff_x + diff_y) ** 0.5

    # Aplique uma função normal de densidade de probabilidade
    # para calcular a proximidade de cada partícula
    self.__pesos *= scipy.stats.norm(dist, 20).pdf(0)

    # Divida cada peso pela soma de todos (Normalização)
    self.__pesos /= sum(self.__pesos)

    return None
```

3.4 REAMOSTRAGEM

Como já descoberto em testes e documentado em artigos [2] a cada iteração o algorítimo pode sofrer com a degeneração causada pelo aumento constante de peso de algumas partículas.

Imagine que 100 partículas são sorteadas randomicamente e que somente três delas se aproximaram significativamente do centro de massa do objeto. As outras 97 partículas ficaram dispersas pelo espaço, transmitindo a sensação de pouca convergência do algorítimo, mais do que isso, estão consumindo recursos como memória e processamento, sem contribuir de forma efetiva para a acurácia da aplicação, com a chance de inclusive, estar degradando o desempenho das poucas partículas que obtiveram êxito.

Para minimizar o problema da degradação, vários modelos do algorítimo implementam a etapa de reamostragem, que é a atualização de partículas do conjunto ou ou substituição por outras mais aptas.

A reamostragem implementada faz uso da substituição de partículas ruins por partículas de maior peso, e o faz com o sorteamento de partículas do próprio grupo, levando em consideração o peso de cada partícula. Ou seja, partículas de pesos mais altos possuem chance maior de compor o novo grupo.

```
def __reamostragem(self):

    prob = 1 / sum(self.__pesos)
    indices = arange(self.n)
    indices = choice(indices, self.n, p=prob * self.__pesos)

    # Re-escolha as partículas de acordo com o índice sorteado
    self.__partics[:] = self.__partics[indices]

    # Atualize o peso para os pesos das partículas sorteadas e normalize-os
    self.__pesos[:] = self.__pesos[indices]
    self.__pesos /= sum(self.__pesos)
    return None
```

Após aplicar a reamostragem, que é a etapa final do filtro, o centro de massa atual substitui o antigo; a aplicação solicita ao grupo de partículas suas coordenadas e as plotam.

4 EXEMPLO DE USO

Para executar a aplicação, deve-se executar os seguintes comandos para instalar as dependências (NumPy, SciPy, OpenCV) utilizadas:

Comando Linux:

```
$ pip3 install opencv-python numpy scipy
```

O comando para Windows é o mesmo, apenas remove-se o ‘3’ depois do ‘pip’:

Comando Windows:

```
$ pip install opencv-python numpy scipy
```

O algoritmo foi estruturado sob uma aplicação de linha de comando (CLI), portanto, os parâmetros e taxas envolvidas são livres para entrada do usuário. Abaixo é possível conferir um exemplo de execução com 20 partículas, velocidade mínima de 3 e máxima de 6 e com vídeo no caminho ‘../assets/basketball.mp4’:

```
python3 main.py -n 20 -v1 3 -v2 6 -p ../assets/basketball.mp4
```

Em caso de dúvidas sobre cada parâmetro, basta chamar a aplicação passando o parâmetro '-h':

```
$ python3 main.py -h
```

Observação: Para sistemas Windows, a chamada é a mesma, porém em vez de utilizar

“python3” para chamar a linguagem, usa-se apenas “python”.

Explicação sobre parâmetros de entrada:

- -n: Número inteiro de partículas;
- -v1: Número real para velocidade mínima;
- -v2: Número real para velocidade máxima;
- -p: Caminho do vídeo a ser processado;

Parâmetros opcionais:

- -c: Valor lógico (0 ou 1), define se os centroides das partículas e do objeto serão exibidos;
- -b: Valor lógico (0 ou 1), define se a box do ponto médio das partículas e a box do objeto serão exibidas;
- -d: Valor real (entre 0.1 e 1) que definem a dimensão (altura e largura) do vídeo;
- -f: Valor inteiro que indica o fator de atraso na exibição do vídeo. O valor padrão é 1, quanto maior mais lenta será a exibição, ideal para acompanhar o movimento das partículas;

5 RESULTADOS E OBSERVAÇÕES

Assim como outros algorítimos desenvolvidos durante a disciplina de Inteligência Artificial, o filtro de partículas tem como fator-chave sua natureza estocástica. Graças a essa natureza é possível extrair algumas informações importantes sobre as relações entre o número de partículas, sua velocidade e a precisão do algoritmo.

5.1 RELAÇÃO: QUANTIDADE DE PARTÍCULAS E PRECISÃO

Tendo em vista que as N partículas são geradas em posições randômicas, há a possibilidade de se ter uma alta diversidade e possível proximidade de alguma partícula do centro de massa buscado. Portanto, quanto maior o número de partículas, maior a possibilidade de, pelo menos, uma partícula se aproximar.

5.2 RELAÇÃO: VELOCIDADE E PRECISÃO

Assim como a posição das partículas, optamos por gerar a velocidade também

de forma randômica, porém, dentro de um intervalo mínimo e máximo definido por quem faz a chamada da aplicação.

Notou-se que o aumento da velocidade implica em um desordenamento maior entre a posição das partículas, até mesmo das mais agrupadas e próximas ao centro de massa buscado. Ou seja, o aumento de velocidade traz como fator positivo a variância significativa de posições de partículas mal localizadas, contudo, seu efeito colateral é a perturbação das posições de partículas já bem localizadas. Ou seja, uma velocidade alta gera uma rápida convergência, porém não mantém estabilidade sólida ao atingir o centro de massa.

5.3 REAMOSTRAGEM

De forma análoga ao mecanismo de elitismo no algoritmo genético, a reamostragem é uma etapa crucial não só para evitar a degradação do algorítimo como também para garantir a seleção e possível povoamento do conjunto com os melhores indivíduos de acordo com um valor, que neste caso, é o peso.

Tendo em vista que a reamostragem ocorre a cada iteração, a convergência do algorítimo só depende de ao menos uma partícula estar melhor localizada do que o restante, dessa forma, esse processo tende a replicar essa partícula bem-sucedida. Se por ventura, a replicação ocorrer somente com partículas de baixo peso, na próxima iteração há novamente a possibilidade obter-se as partículas de maiores pesos (maior probabilidade de ser escolhida).

A única distinção entre reamostragem e o elitismo é que o primeiro método faz uso da escolha de objetos por probabilidade baseada no peso, isto é, há a possibilidade das melhores partículas não serem escolhidas, enquanto que o segundo método sempre obtivera o melhor indivíduo, sem dar margem para eventos estocásticos.

5.4 DESENVOLVIMENTO COM FOCO NO DESEMPENHO

Embora o projeto tenha sido desenvolvido na linguagem de programação Python, uma linguagem interpretada e de alto nível de abstração, foram utilizadas bibliotecas e uma modelagem de dados que permitissem altíssimo desempenho se comparado com um código puramente em Python e plenamente orientado a objetos.

A modelagem das partículas e de seus atributos em arrays de tipos primitivos (inteiro, real) permitiu operações aritméticas básicas e limpas. Neste caso, modelar o grupo de partículas como uma classe e a partícula em si como outra, não só aumentaria

a quantidade de código mas também seu tempo processamento, tendo em vista as validações para objetos, instanciação via construtor e outros custos que englobam o encapsulamento de dado via Orientação a Objetos.

Aliado à modelagem, a biblioteca NumPy, que como o nome indica, tem seu foco voltado para aplicações científicas ou com quantidade rica de operações matemáticas, possibilitou operações envolvendo múltiplos arrays (soma, multiplicação, subtração e divisão de N arrays, como se fossem simples inteiros ou números reais) em linha única de código, o que proporcionou simplicidade e clareza ao código, além de alto desempenho, uma vez que mais da metade do código NumPy é escrito na linguagem C [3] e é otimizada para operações e tipos específicos de estruturas de dados (como o array).

6 REFERÊNCIAS E OUTROS MATERIAIS BASE

[1] “Métodos Sequencias de Monte Carlo Bayesianos: Aspectos Computacionais, Inferenciais e Aplicações”. Página 26. Acesso em 05/07/2019. Disponível em: http://www.est.ufmg.br/portal/arquivos/doutorado/teses/tese_felipe_carvalho_alvares_da_silva.pdf

[2] “Filtro de partículas hibridizado com métodos da computação natural para detecção e rastreamento”. Página 16. Acesso em 05/07/2019. Disponível em: http://repositorio.ufes.br/bitstream/10/4241/1/tese_4175_.pdf

[3] Projeto e código-fonte Numpy. Acesso em 04/07/2019. Disponível em: https://github.com/numpy/numpy

[4] Detecção de objetos por cores, via OpenCV. “Object detection using HSV Color space – OpenCV 3.4 with python 3”. Acesso em 20/06/2019. Disponível em: https://www.youtube.com/watch?v=SJCu1d4xakQ

[5] Código base para construção do filtro. “Parcticle Filter Explained With Python Code From Scratch”. Acesso em 03/07/2019. Disponível em: http://ros-developer.com/2019/04/10/parcticle-filter-explained-with-python-code-from-scratch/