FACULTAD
DE INGENIERIA
Universidad de Buenos Aires

TA154 - Robótica Móvil

Práctica 1 - Transformaciones, locomoción y sensado

1C2025

Dominguez Lautaro Tadeo 109645 ltdominguez@fi.uba.ar

Índice

1. Muestreo de distribuciones de probabilidad

A continuación se presentan tres funciones que permiten generar muestras de una distribución normal $\mathcal{N}(\mu, \sigma^2)$ utilizando únicamente variables aleatorias uniformes en el intervalo $[0, 1]$. Cada función implementa un método distinto: suma de 12 uniformes, muestreo con rechazo y transformación de Box-Muller. En todos los casos, las muestras generadas se ajustan a los parámetros de media y varianza indicados por el usuario.

Para la primera función se utiliza la suma de variables uniformes independientes para aproximar una distribución normal, basado en el Teorema Central del Límite. Al sumar suficientes variables $U(0, 1)$, la distribución de su suma tiende a ser normal.

```
function x = normal_suma_uniformes(mu, sigma2)
    % Genera una muestra de una distribucion normal N(mu, sigma2) utilizando el Teorema Central
        del Limite con 12 uniformes

    N=1000; %Numero de muestras

    n = 12;  % cantidad de variables uniformes a sumar
    u = rand(n, N);  % 12 variables U(0,1)

    % Aplico Teorema Central del Limite
    z = sum(u - 0.5)/(sqrt(n)*sqrt(1/12));

    % Transformar a N(mu, sigma2)
    x = mu + sqrt(sigma2) * z;
end
```

Listing 1: Implementacion con suma de Uniformes

El próximo método es el de rechazo, que genera candidatos a partir de una distribución uniforme y acepta las muestras con una probabilidad proporcional a la relación entre la función de densidad objetivo y una función propuesta. En este caso, se utiliza una uniforme truncada, como se muestra a continuación.

```
function x = normal_rechazo(mu, sigma2)
    % Genera una distribucion normal N(mu, sigma2) usando el metodo de rechazo
    N=1000;

    a = mu - 3 * sqrt(sigma2);  % Establecer los limites a, b, basados en mu y sigma
    b = mu + 3 * sqrt(sigma2);

    % Funcion de densidad de la distribucion normal N(mu, sigma2)
    f = @(x) (1 / (sqrt(2 * pi * sigma2))) * exp(-0.5 * ((x - mu).^2) / sigma2);

    % Funcion de densidad de uniforme U(0,1)
    g = 1 / (b-a);

    x = zeros(1, N);   % Vector para las muestras

    % Generar todas las muestras de la distribucion uniforme U(0,1)
    i=1;
    while i<= N
            % Generar todas las muestras de la distribucion uniforme U(0,1)
            u = rand;
            y = a + (b - a) * rand();
            if u <= f(y)
                x(i) = y;  % Aceptar la muestra
                i= i+1;
            end
    end
end
```

Listing 2: Implementacion con Método de Rechazo

Para la siguiente función se utilizó el método de **Box-Muller**, que transforma dos variables aleatorias uniformes independientes $U_1, U_2 \sim U(0, 1)$ en dos variables normales estándar $Z_1, Z_2 \sim \mathcal{N}(0, 1)$, a través de la siguiente transformación:

$$Z_1 = \sqrt{-2 \ln U_1} \cos(2\pi U_2), \quad Z_2 = \sqrt{-2 \ln U_1} \sin(2\pi U_2. \tag{1}$$

```
function x = normal_box_muller(mu, sigma2)
    % Genera una muestra de una distribucion normal N(mu, sigma2) usando Box-Muller
    N= 1000;     %Numero de muestras

    U1 = rand(1,N);  % muestras U(0,1)
    U2 = rand(1,N);  % otras muestras U(0,1)

    % Box-Muller para obtener una N(0,1)
    Z = sqrt(-2*log(U1)).*cos(2*pi*U2);

    % Escalado a N(mu, sigma2)
    x = mu + sqrt(sigma2) * Z;
end
```

Listing 3: Implementacion con Box-Muller

Por último, se agregó una nueva función que implementa el método de la transformación inversa. Este método genera muestras normales aplicando la función inversa de la función de distribución acumulada (CDF) de la normal estándar a una variable uniforme.

```
function x = normal_inv(mu, sigma2)
    % Genera una muestra de una distribucion normal N(mu, sigma2) utilizando el metodo de la
        transformada inversa
    N= 100;     %Numero de muestras

    U = rand(1, N);  % Generamos una muestra uniforme U(0,1)

    % Calcular la inversa de la CDF de la normal estandar usando norminv
    Z = norminv(U, 0, 1);  % norminv(U, 0, 1) da una N(0,1)

    % Transformamos a N(mu, sigma^2)
    x = mu + sqrt(sigma2) * Z;
end
```

Listing 4: Implementacion con Método de la transformación inversa

Se graficó el histograma de las muestras generadas por cada una de las funciones utilizando $\mu = 1$ y $\sigma^2 = 4$. A cada histograma se le superpuso la curva de la densidad teórica de la distribución normal correspondiente, con el objetivo de verificar visualmente el ajuste de los métodos implementados.

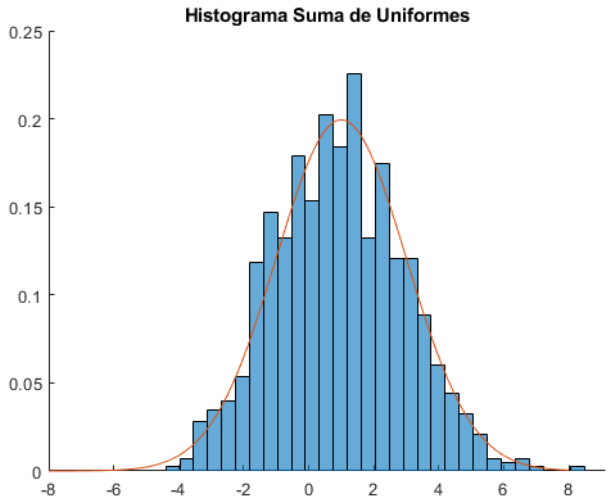

Figura 1: Histograma con suma de Uniformes

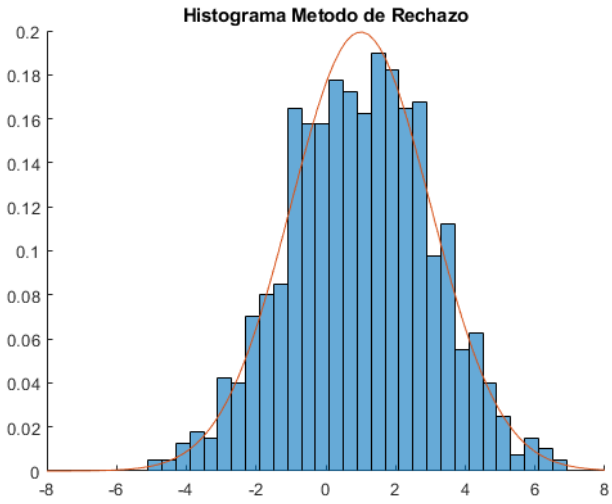

Figura 2: Histograma con Método de Rechazo

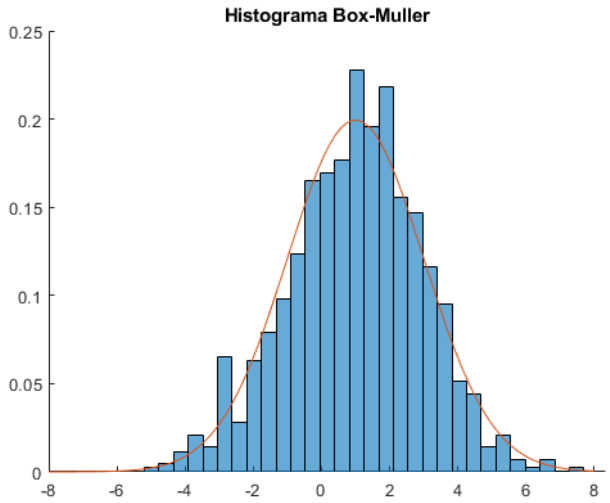

Figura 3: Histograma con Box-Muller

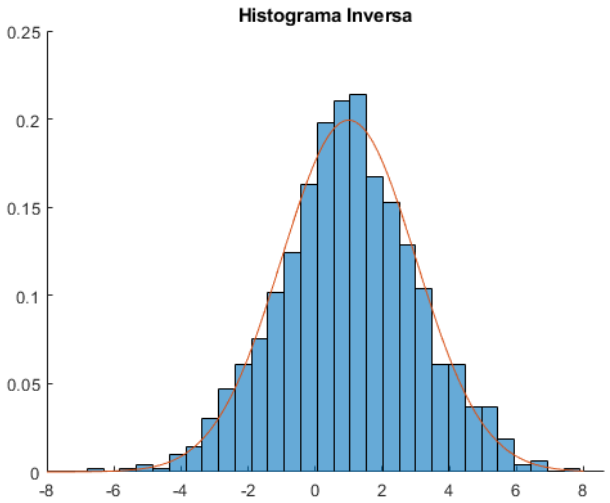

Figura 4: Histograma con Método de la transformación inversa

A su vez, se calcularon los tiempos de ejecución de las diferentes funciones implementadas para la generación de distribuciones gaussianas y la función predeterminada de MATLAB `normrnd`. Los tiempos de ejecución obtenidos para cada uno de los métodos son los siguientes:

- Tiempo del método Suma de Uniformes: 5.7590e-04 s
- Tiempo del método Box-Muller: 5.3820e-04 s
- Tiempo del método de Rechazo: 0.0016 s
- Tiempo del método Inversa: 5.5290e-04 s
- Tiempo de la función predeterminada `normrnd`: 3.5150e-04 s

En conclusión, los métodos personalizados para la generación de distribuciones gaussianas, como la Suma de Uniformes, Box-Muller y la Inversa, son efectivos y presentan tiempos de ejecución similares, con una ligera diferencia entre ellos. Sin embargo, el método de Rechazo es considerablemente más lento y menos eficiente en comparación con los otros métodos. Igualmente, la función predeterminada de MATLAB, `normrnd`, es la opción más rápida y eficiente en términos de tiempo de ejecución, lo que la convierte en la opción preferida si se busca optimizar el rendimiento al generar distribuciones gaussianas.

2. Modelo de movimiento basado en odometría

A continuación, se implementa un modelo de movimiento probabilístico basado en odometría, el cual estima la nueva posición del robot a partir de su pose actual x_t, una lectura de odometría u_t, y parámetros de ruido *alpha* . Este modelo considera errores en las rotaciones y traslación del movimiento y utiliza el método de muestreo mediante la suma de variables uniformes (Teorema Central del Límite) para simular ruido normalmente distribuido.

Se generan 5000 muestras del nuevo estado del robot, y se grafican las posiciones resultantes (x_{t+1}, y_{t+1}), como se solicita en la consigna. A continuación la implementación del modelo.

```
% Parametros dados
x_t = [2.0; 4.0; pi/2];               % Estado inicial del robot (x, y, orientacion)
u = [pi/4; 0.0; 1.0];                 % Comando de movimiento (rotacion1, rotacion2, traslacion)
alpha = [0.1; 0.1; 0.01; 0.01];       % Parametros del modelo de ruido
N = 5000;                             % Numero de muestras

% Generar muestras del modelo de movimiento
[x_tmas, y_tmas, tita_tmas] = modelo_movimiento(x_t, u, alpha);

% Graficar las posiciones (x, y) resultantes de las muestras
figure;
plot(x_tmas, y_tmas, '.');          % Grafico de dispersion de las posiciones
xlabel('x_{t+1}');
```

```
ylabel('y_{t+1}');
title('Distribucion de muestras del modelo de movimiento');
axis equal;
grid on;

% Funcion que implementa el modelo de odometria con ruido
function [xtmas, ytmas, titatmas] = modelo_movimiento(x_t, u, alpha)

    % Calculo de las varianzas de los errores
    sigma2_1 = alpha(1)*abs(u(1)) + alpha(2)*u(3);              % Varianza de la primera
        rotacion
    sigma2_2 = alpha(3)*u(3) + alpha(4)*(u(1)+u(2));             % Varianza de la traslacion
    sigma2_3 = alpha(1)*abs(u(2)) + alpha(2)*u(3);              % Varianza de la segunda
        rotacion

    % Generacion de movimientos ruidosos usando distribuciones normales
    delta_rot1 = u(1) + normal_suma_uniformes(0, sigma2_1);     % Rotacion 1 ruidosa
    delta_tran = u(3) + normal_suma_uniformes(0, sigma2_2);     % Traslacion ruidosa
    delta_rot2 = u(2) + normal_suma_uniformes(0, sigma2_3);     % Rotacion 2 ruidosa

    % Calculo de las nuevas posiciones (x, y, orientacion)
    c1 = cos(x_t(3) + delta_rot1);
    c2 = sin(x_t(3) + delta_rot1);

    xtmas = x_t(1) + c1 .* delta_tran;                          % Nueva coordenada x
    ytmas = x_t(2) + c2 .* delta_tran;                          % Nueva coordenada y
    titatmas = x_t(3) + delta_rot1 + delta_rot2;                % Nueva orientacion

end
```

Listing 5: Modelo de movimiento basado en odometría

La Figura 5 muestra la dispersión de las 5000 posiciones finales (x_{t+1}, y_{t+1}) obtenidas a partir del modelo. Se puede observar cómo el ruido en los comandos de odometría afecta la distribución de las muestras, generando una nube de puntos alrededor del movimiento esperado.

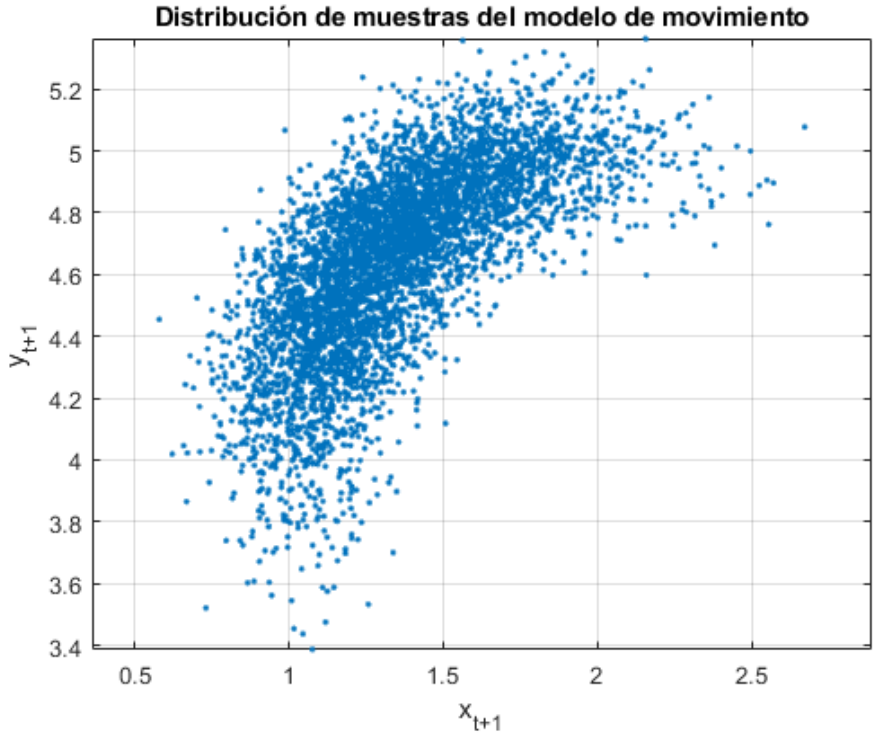

Figura 5: Modelo de movimiento

3. Filtro Discreto

En esta sección se implementa un filtro de Bayes discreto para estimar la posición de un robot que se mueve en un mundo unidimensional acotado de 20 celdas. El robot comienza en la celda 10 (índice 10 en un arreglo de 0 a 19) y su movimiento está sujeto a incertidumbre. El filtro de Bayes discreto consiste en mantener una distribución de probabilidad (*belief*) sobre la posición del robot. Esta distribución se actualiza a medida que el robot ejecuta comandos de movimiento, incorporando la incertidumbre del modelo. A continuación la implementación del Filtro Discreto.

```
function new_bel = aplicar_movimiento(bel, comando)
    N = length(bel);  % Numero total de celdas en el mundo
    switch comando   % Definir el modelo de movimiento segun el comando
        case 'avanzar'
            % Probabilidades asociadas al comando "avanzar": 25% de quedarse, 50% de avanzar 1,
                25% de avanzar 2 celdas
            probs = [0.25, 0.5, 0.25];
            desplazamientos = [0, 1, 2];  % Desplazamientos relativos desde la celda actual
        case 'retroceder'
            % Probabilidades similares pero en direccion opuesta
            probs = [0.25, 0.5, 0.25];
            desplazamientos = [0, -1, -2];
    end
    new_bel = zeros(size(bel));  % Inicializamos el nueva belive en cero
    for i = 1:N  % Para cada celda actual i (donde el robot podria estar)
        % Para cada posible resultado del modelo de movimiento
        for j = 1:length(probs)
            dest = i + desplazamientos(j);  % Calculamos la celda de destino

            if dest >= 1 && dest <= N
                % Si el destino esta dentro del mundo, distribuimos la probabilidad a esa celda
                new_bel(dest) = new_bel(dest) + bel(i) * probs(j);
            else
                % Si el movimiento se sale del mundo, se asume que el robot se queda en su
                    lugar
                new_bel(i) = new_bel(i) + bel(i) * probs(j);
            end
        end
    end
```

En la siguiente figura se muestra el *belive* resultante de avanzar nueve veces seguido de retroceder tres veces consecutivas. También se puede la animación del paso a paso en el siguiente enlace.

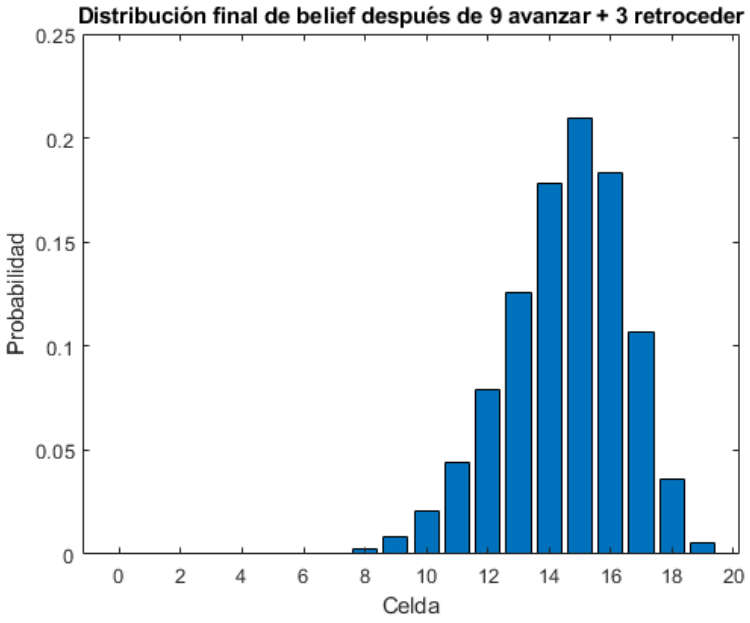

Figura 6: Belief distribución final

4. Filtro de Partículas

A continuación se presentan las funciones clave utilizadas para implementar el filtro de partículas en un escenario de localización de un robot.

El primer paso es el cálculo de pesos mediante el modelo de medición, que compara las observaciones del sensor con las distancias esperadas entre cada partícula y los marcadores del entorno. La función *measurement_model* calcula la probabilidad de cada observación utilizando la distancia entre la partícula y el marcador.

```
function weight = measurement_model(z, x, l)
    % Computes the observation likelihood of all particles.
    %
    % The employed sensor model is range only.
    %
    % z: set of landmark observations. Each observation contains the id of the landmark
       observed in z(i).id and the measured range in z(i).range.
    % x: set of current particles
    % l: map of the environment composed of all landmarks
    sigma = [0.2];
    weight = ones(size(x, 1), 1);

    if size(z, 2) == 0
        return
    end

    for i = 1:size(z, 2)
        landmark_position = [l(z(i).id).x, l(z(i).id).y];
        measurement_range = [z(i).range];
%-------------------------------------------------------------------------
     % Calcular la distancia esperada desde cada particula al marcador
        dx = x(:,1) - landmark_position(1);
        dy = x(:,2) - landmark_position(2);

        expected_range = sqrt(dx.^2 + dy.^2); % Apartir de las distacias entre el robot y el
           landmark en x e y , calculo la distancia hasta el landmark

        % Calcular el likelihood de esa medicion para cada particula
        p = (1 / (sqrt(2 * pi) * sigma)) * exp(-0.5 * ((measurement_range - expected_range).^2)
            / sigma^2);

        % Actualizar el peso de cada particula multiplicando (porque las observaciones son
           independientes)
        weight = weight .* p;
%-------------------------------------------------------------------------

    end

    weight = weight ./ size(z, 2);
end
```

Listing 6: Modelo de medición

El siguiente paso es el re-muestreo de las partículas, la función *resample* implementa el método de muestreo estocástico universal, el cual selecciona partículas con una probabilidad proporcional a su peso.

```
    function new_particles = resample(particles, weights)
    % Returns a new set of particles obtained by performing
    % stochastic universal sampling.
    %
    % particles (M x D): set of M particles to sample from. Each row contains a state
       hypothesis of dimension D.
    % weights (M x 1): weights of the particles. Each row contains a weight.
    new_particles = [];

    M = size(particles, 1);

```

```
%-------------------------------------------------------------------------
    % Distancia entre pasos
    step = 1 / M;

    % Primer paso aleatorio entre 0 y step
    start = rand * step;

    punteros = start + (0:M-1) * step; % M saltos equiespacidos

    weights_cumsum = cumsum(weights); % Acumulo todos los pesos
    i=1;
    for j = 1:M

        while punteros(j)< weights_cumsum(i) && i < M
            i = i + 1; %Salteo la particula por que el puntero no cayo en el tramo asignado a
                esa particula
        end
    new_particles(j, :) = particles(i, :);  % Elegimos esa particula
    end
%-------------------------------------------------------------------------
end
```

Listing 7: Algoritmo de Remuestreo

Finalmente, para obtener una estimación de la posición del robot, se calcula la media ponderada de las partículas. La función *mean_position* devuelve la posición estimada basada en los pesos y las posiciones de las partículas.

```
    function mean_pos = mean_position(particles, weights)
    % Returns a single estimate of filter state based on the particle cloud.
    %
    % particles (M x 3): set of M particles to sample from. Each row contains a state
        hypothesis of dimension 3 (x, y, theta).
    % weights (M x 1): weights of the particles. Each row contains a weight.

    % initialize
    mean_pos = zeros(1,3);

     mean_pos = sum(particles .* weights, 1) / sum(weights);
end
```

Este conjunto de funciones implementa un filtro de partículas que permite estimar la posición de un robot en un entorno dinámico. El resultado de todo este proceso puede verse en el siguiente video.