2023-05

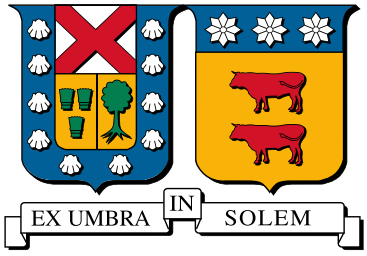
Implementación y Evaluación de Algoritmos de Estimación de Estados Usando Datos Cuantizados

Langhaus Gordon, Osvaldo Augusto José

UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA

DEPARTAMENTO DE ELECTRÓNICA

VALPARAISO - CHILE



"Implementación y Evaluación de Algoritmos de Estimación de Estados Usando Datos Cuantizados"

Osvaldo Augusto José Langhaus Gordon

Memoria para optar al título de Ingeniero Civil Electrónico, Mención en Computadores, Formación Complementaria en Control Automático

PROFESOR GUIA: Dr. Juan Carlos Agüero

PROFESOR CORREFERENTE: Dr. Gonzalo Carvajal

Mayo 2023

Agradecimientos

Dedico esta sección para expresar mis agradecimientos a la Universidad Técnica Federico Santa María, al Centro Avanzado de Ingeniería Eléctrica y Electrónica (AC3E) proyecto Basal ANID FB0008, al proyecto Fondecyt ANID 1211630 del profesor Juan C. Agüero, y a cada profesor y estudiante involucrado.

Implementación y Evaluación de Algoritmos de Estimación de Estados Usando Datos Cuantizados

Osvaldo Augusto José Langhaus Gordon

Memoria para optar al título de Ingeniero Civil Electrónico, Mención en Computadores, Formación Complementaria en Control Automático.

Universidad Técnica Federico Santa María

Profesor Guía: Dr. Juan Carlos Agüero

Mayo 2023

Resumen: Un problema común en el área de la tecnología electrónica es la incapacidad de varios dispositivos para obtener mediciones exactas de sus procesos o magnitudes durante supervisión y control, debido a la naturaleza de los dispositivos de medición. Por ejemplo, el caso de interés es la cuantización introducida a la salida por sensores de bajo costo y baja resolución. En los últimos años se han hecho grandes avances en algoritmos que utilizan ecuaciones matemáticas para solucionar el problema en cuestión, al estimar estados no medibles de un sistema dinámico no lineal en base a conjuntos datos recopilados de su salida que si es medible, incluso cuando esta es cuantizada. Entre estos, la implementación del grupo conformado por Angel L. Cedeño, Ricardo Albornoz, Rodrigo Carvajal, Boris I. Godoy y Juan C. Agüero para el diseño de un filtro Suma de Gaussianas ha mostrado mejores resultados durante sus pruebas que sus predecesores. El problema está en que este se encuentra escrito en MATLAB, lo que dificulta su aplicación en problemas prácticos de estimación de estados. Este proyecto busca describir, implementar, evaluar y comparar diferentes métodos de estimación de estados clásicos para filtraje en Python, junto al filtro Suma de Gaussianas, para demostrar la viabilidad de este método en un entorno de programación que le permita competir con los métodos clásicos y ser usado en aplicaciones prácticas.

Palabras Clave: Python; espacio de estados; estimación de estados; cuantización; filtro de Kalman; filtro de partículas; filtro Suma de Gaussianas

Implementation and Evaluation of State Estimation Algorithms Using Quantized Data

Osvaldo Augusto José Langhaus Gordon

Thesis for the fulfillment of the B.S. in Electronic Engineering, major in Computers, minor in Automatic Control (6 year program).

Universidad Técnica Federico Santa María

Advisor: Juan Carlos Agüero, PhD.

May 2023

Abstract: A common problem in the area of electronic technology is the inability of various devices to obtain accurate measurements of their processes or magnitudes during supervision and control, due to the nature of the measurement devices. For example, the case of interest is the quantization introduced at the output by low-cost and low-resolution sensors. In recent years, great advances have been made in algorithms that use mathematical equations to solve the problem in question, by estimating non-measurable states of a nonlinear dynamical system based on data sets collected from its output that is measurable, even when this is quantized. Among these, the implementation of the group formed by Angel L. Cedeño, Ricardo Albornoz, Rodrigo Carvajal, Boris I. Godoy and Juan C. Agüero for the design of a Gaussian Sum filter has shown better results during its tests than its predecessors. The problem is that it is written in MATLAB, which makes it difficult to apply it to practical state estimation problems. This project seeks to describe, implement, evaluate and compare different classical state estimation methods for filtering in Python, together with the Gaussian Sum filter, to demonstrate the viability of this method in a programming environment that allows it to compete with classical methods and be used in practical applications.

Keywords: Python; state space; state estimation; quantization; Kalman filter; particle filter; Gaussian Sum filter

Índice de contenidos

Índice de figuras

Índice de tablas

Índice de códigos

1. Introducción

1.1. Estado del Arte

En el área de la tecnología electrónica existe un procedimiento matemático que ha sido objeto de extenso estudio desde que se empezaron a implementar sistemas de control y observación de sistemas dinámicos. A este procedimiento se le da el nombre de filtraje y suavizado para la estimación de estados. El filtraje de datos se centra en aplicar operaciones matemáticas para calcular en algún instante de tiempo el valor aproximado de una magnitud presente en un sistema dinámico, a base de la medición en el mismo instante de otra magnitud asociada a la primera y al resultado del instante anterior. Similarmente, el suavizado realiza la misma tarea, pero con mediciones de la magnitud en todo instante de tiempo, en una ventana $[t_1, t_N]$. Tienen inicios en la necesidad de supervisar y manipular la evolución de un proceso cuando este no posee condiciones apropiadas de observabilidad, por ejemplo, el programa espacial de los años 60s necesitaba aproximar la trayectoria de sus naves ante la falta un marco de referencia de posición adecuado o mecanismos sofisticados de localización [15].

Varios científicos y matemáticos del último siglo han combinado sus conocimientos para definir y refinar algoritmos, cuya función es resolver el problema de filtraje cuando el sistema objetivo presenta dinámicas no lineales, y perturbaciones en sus estados y salidas. Estos algoritmos se utilizan hoy en día en diferentes aplicaciones, como diseño de sistemas de control, sistemas de potencia, sistemas de navegación, redes de sensores, entre otras. En particular, en las últimas décadas ha aumentado el uso de sensores y redes de sensores, por ejemplo en aplicaciones industriales para la medición y supervisión automática de señales físicas como el nivel de líquidos, temperatura, potencia eléctrica, etc. Problemas comunes que presentan estos sensores incluyen la pérdida de información de señales medidas con sensores económicos de baja resolución, o por el almacenamiento y transmisión de representaciones reducidas de las señales para minimizar el consumo de recursos en un canal de comunicación. Una de las formas que toma esta pérdida de información debido a sensores se conoce como cuantización de datos. Se trata de una no linealidad, donde un conjunto de datos continuos o pseudo-continuos se reduce a niveles discontinuos en función de uno o más umbrales, generando una pérdida de información que hace estimar los estados del sistema un proceso no trivial ni directo con

métodos convencionales. Un caso común de cuantización es el caso binario, donde hay un único umbral y dos niveles. [14]. Este trabajo se enfoca principalmente en estudiar métodos de estimación de estados capaces de lidiar con cuantización de datos.

Algunos algoritmos clásicos utilizan un enfoque probabilístico, específicamente enfoque bayesiano, para aproximar el estado del sistema a base de salidas aleatorias, con la función objetivo siendo la función de densidad de probabilidad (*Probability Density Function* o PDF) de filtraje bajo condiciones no necesariamente conocidas. El filtro de Kalman estándar es el método más utilizado para realizar esta tarea cuando se trabaja con sistemas lineales, bajo el supuesto de que se encuentran sometidos a ruidos gaussiano. Similarmente, cuando el sistema es no lineal, pero las condiciones del ruido son las mismas, se pueden utilizar otras versiones del algoritmo. El filtro de Kalman extendido [13][18] utiliza una aproximación de Taylor para generar un espacio de estados extendido lineal en torno a una estimación del estado y así aplicar las ecuaciones del filtro de Kalman estándar. El filtro de Kalman en cuadratura [1][31] funciona bajo el mismo principio, pero utilizando puntos de la cuadratura de Gauss (Hermite o Laguerre) para extender el espacio de estados mediante una regresión lineal, y así obtener soluciones cerradas para la PDF de filtraje. Adicionalmente, el filtro de Kalman cuantizado [35], permite incorporar el efecto de la cuantización de las observaciones. Otros métodos que son útiles para el filtraje son basados en métodos de Monte Carlo, tal como el filtro de partículas, también conocido como muestreo secuencial de importancia (*Sequential Importance Sampling* o SIS) [8][17]. Este método estima el estado de un sistema mediante la propagación de partículas aleatorias y pesos a través de las funciones no lineales del sistema, ubicándolos estocásticamente sobre la distribución del estado en algún instante particular. La desventaja de este método es que su exactitud esta directamente relacionada a la cantidad de recursos computacionales asignados algoritmo durante su ejecución, por lo que obtener una aproximación cercana al sistema real puede resultar inviable.

Cada año se realizan esfuerzos para optimizar el proceso de filtraje para sistemas dinámicos no lineales. Recientemente se ha desarrollado e implementado un algoritmo que combina varias estructuras y técnicas matemáticas para estimar de manera probabilística la PDF cuando esta es una combinación lineal de modelos gaussianos, conocido como filtro Suma de Gaussianas [3] [26]. Su atractivo está en que es capaz de conseguir una estimación con una

exactitud comparable a un filtro de partículas de muchas partículas, consumiendo significativamente menos recursos computacionales. Además, sus condiciones de resultado óptimo son menos estrictas que con el filtro de Kalman.

1.2. Definición del Problema y Objetivos

Este proyecto tiene interés en el trabajo del grupo conformado por Angel L. Cedeño, Ricardo Albornoz, Rodrigo Carvajal, Boris I. Godoy y Juan C. Agüero, para implementar y evaluar diferentes algoritmos de estimación de estados que trabajan a partir de datos cuantizados, escrito en las referencias [3] y [5]. Estos artículos elaboran sobre el desarrollo de un diseño que realiza filtraje y suavizado Suma de Gaussianas para lidiar con mediciones sometidas a ruidos gaussianos y cuantización, y posteriormente lo compara con métodos clásicos, catalogando cada método según los resultados obtenidos de un conjunto de experimentos que buscan medir el error de la estimación del filtraje y suavizado en comparación al estado real del sistema, y el tiempo de ejecución del suavizado. En total, los algoritmos implementados son filtro/suavizado de Kalman estándar, extendido y *unscented*, filtro/suavizado de partículas, y filtro/suavizado Suma de Gaussiana.

Este diseño se implementa en el entorno de desarrollo integrado MATLAB, debido a sus notables ventajas en la escritura de código orientado al cálculo matemático, y a su extensa selección de herramientas que facilitan el cómputo de datos. El problema abordado por el proyecto empieza aquí. Este diseño fue creado con el propósito de utilizarse en soluciones a problemas reales de control automático, pero MATLAB no es una plataforma frecuentemente utilizada por instituciones no educativas o de investigación para la ejecución de programas o implementación de aplicaciones sobre sistemas físicos, por diferentes razones. Por ejemplo, MATLAB requiere una licencia, y existen varios entornos y lenguajes de programación gratuitos, cuyo rendimiento es a lo menos comparable con MATLAB y son capaces de entregar resultados semejantes, por lo que resulta más atractivo utilizar alguna de estas alternativas. Esta situación genera dificultades para que el filtro Suma de Gaussianas escrito en MATLAB compita con algoritmos que buscan resolver problemas similares en diferentes entornos y lenguajes de programación, independiente de la posibilidad que este entregue resultados con menor error de estimación, menor tiempo de ejecución y/o menor gasto computacional.

El objetivo principal de este proyecto es proveer un ejemplo de la implementación del filtro Suma de Gaussianas para resolver problemas de estimación de estados con datos cuantizados, escrito en un entorno y/o lenguaje de programación utilizado más ampliamente en la industria de la tecnología electrónica, particularmente en el área del control automático. Este ejemplo tiene el propósito de ser utilizado como marco de referencia para deliberar sobre trabajos futuros asociados al diseño del algoritmo, por lo que es necesario comparar su rendimiento con otros ejemplos de métodos de estimación de estados diferentes y determinar si, bajo las condiciones de prueba dadas, es por lo menos comparable con ellos y con su respectiva implementación en MATLAB.

El trabajo comienza con la selección un entorno y/o lenguaje de programación adecuado, donde se pueda implementar el diseño del filtro Suma de Gaussianas y obtener resultados comparables a su implementación en MATLAB. Se debe tratar de una semántica estandarizada que se pueda usar en plantas existentes, no debe requerir de un entorno cerrado o configuraciones muy particulares para su funcionamiento, y debe ser frecuentemente utilizado en la industria para aplicaciones, como supervisión y control de señales. Posteriormente, se debe implementar el diseño y otros métodos clásicos de estimación de estados con datos cuantizados con la semántica seleccionada, y evaluar el funcionamiento de cada uno. Se ha decidido utilizar Python para escribir el ejemplo basado en el código de MATLAB propuesto. Entonces, los objetivos específicos del proyecto se pueden resumir como:

- Implementar el filtro Suma de Gaussianas y otros métodos de estimación clásicos sobre un modelo de espacio de estados en Python, y comprobar que funcionan apropiadamente mediante la simulación de sus variables estados en el tiempo. Si el modelo con el se trabaja es lineal y sus perturbaciones son despreciables, cada uno de los métodos debe ser capaz de reconstruir muy cercanamente, si no exactamente, el estado del modelo en todo instante t, donde $t = 1, \ldots, N$.

- Comparar el rendimiento de las implementaciones con diseños de código abierto, publicados en repositorios en línea, bajo condiciones idénticas de simulación. Esta tarea se realiza para aumentar el marco de referencia en la comparación final del rendimiento de los algoritmos implementados.

- Evaluar, comparar y catalogar cada implementación seleccionada según criterios de comparación definidos como los promedios de error de estimación, tiempo de ejecución y costo computacional medido a través del uso de memoria máximo. Comparar el resultado final con los experimentos realizados para los diseños en MATLAB y determinar si son consistentes. En el caso de identificar pérdidas en el rendimiento, determinar si estas son constantes entre cada algoritmo o particulares de un diseño.

- Extraer conclusiones apropiadas de los resultados obtenidos sobre la capacidad que posee el ejemplo del filtro Suma de Gaussianas implementado fuera de MATLAB para competir con ejemplos de métodos clásicos en la resolución de problemas de estimación con datos cuantizados.

Para facilitar el trabajo experimental, se opta por centrar el trabajo en el diseño del filtraje, y dejar el suavizado como información complementaria. Además, se decide omitir la implementación de métodos que muestran el rendimiento más deficiente en los experimentos realizados en [5], y que requieren dominar conceptos sobre expansión de modelos matemáticos, como el filtro de Kalman extendido y el filtro de Kalman unscented. En otras palabras, los algoritmos seleccionados para realizar la comparación con el filtro Suma de Gaussianas son el filtro de Kalman estándar, el filtro de Kalman cuantizado y el filtro de partículas MCMC con resampling sistemático.

1.3. Herramientas

El proyecto se enfoca en el modelado y simulación de sistemas dinámicos, por lo que no existe un requerimiento específico de hardware para completar las tareas planteadas. Del lado de software y semántica, es necesario seleccionar un lenguaje de programación que permita cumplir con el objetivo de portabilidad del proyecto, y una plataforma adecuada para compilación y ejecución de programas, que permita la evaluación y obtención de datos asociados a cada algoritmo de interés. Se propone utilizar las siguientes herramientas para el trabajo experimental:

- **Python:** Python es un lenguaje de programación de alto nivel y propósito general, utilizado frecuentemente en el desarrollo de aplicaciones. Es ampliamente utilizado por empresas, instituciones y plataformas web, debido a su distribución completamente abierta y gratuita, facilidad de uso, y su extensa selección de bibliotecas con herramientas útiles para el manejo de datos. Su filosofía se centra en la productividad, legibilidad de código y calidad de software, por lo que es fácil de aprender y utilizar. Posee escritura dinámica durante su ejecución, no es necesario asignar recursos de memoria manualmente. Su biblioteca estándar posee funciones para cumplir prácticamente cualquier tarea, y el proceso de importar bibliotecas externas es sencillo, mediante el uso de su herramienta de administración de paquetes (*pip*). Además, posee rutinas de manejo de excepciones que permiten no pausar la ejecución del código si se encuentra un error esperado. Una de sus desventajas está asociada a su escritura dinámica, la cual provoca que algoritmos escritos en Python tengan un tiempo de ejecución generalmente mayor que en otros entornos y lenguajes como MATLAB, C y C++. Esto también resulta en un uso de memoria ineficiente. Las pruebas experimentales determinaran si estas desventajas resultan perjudiciales para la aplicación de los métodos de estimación, pero hasta el momento Python se presenta como uno de los lenguajes más apropiados para este trabajo.

- **Jupyter Notebook:** Se trata de una aplicación de navegador web que permite establecer de manera sencilla un ambiente de desarrollo para código, administración de datos y aplicaciones. Es una buena herramienta para programadores con poca experiencia pues su interfaz organizada permite escribir, compilar y ejecutar programas desde cualquier

navegador, tanto desde el mismo espacio de trabajo como de archivos locales, y soporta salidas con diversos formatos, incluyendo imágenes, video, HTML, entre otros. Para el proyecto significa un medio para organizar y graficar fácilmente los datos que interesa recopilar para examinar el rendimiento de los algoritmos implementados.

Figura 1.1: Ejemplo de espacio de trabajo con interfaz de Jupyter Notebook.

- **MATLAB:** La plataforma MATLAB aún posee utilidad en este proyecto, a pesar del objetivo principal que es emplear un medio alternativo para implementar los algoritmos. Lo más importante es que MATLAB posee un modo de depuración, con el cual se puede obtener una referencia de los datos durante la ejecución de dichos algoritmos, y comparar las estructuras y operaciones matemáticas individuales aplicadas en Python. Esto se realiza con el fin de asegurar tempranamente una implementación bien encaminada, y no depender del resultado final para comprobar su funcionalidad. Además, MATLAB contiene los ejemplos que se utilizan como referencia para implementar los algoritmos de estimación elegidos en Python, y en la comparación final de datos. Cada ejemplo fue proporcionados por Angel L. Cedeño, previo al inicio del trabajo experimental.

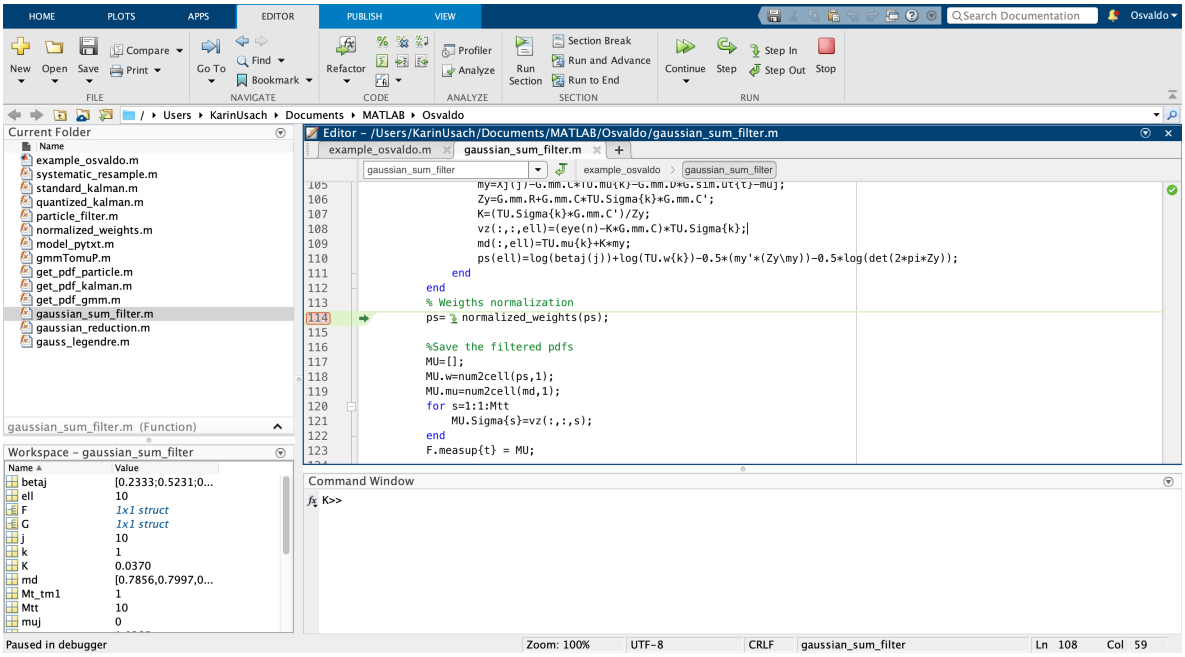


Figura 1.2: Ejemplo de espacio de trabajo de MATLAB en modo de depuración.

2. Conceptos y Definiciones

2.1. Espacio de Estados

Los espacios de estados, también conocidos como representación en variables de estado, son una herramienta para representar las magnitudes físicas involucradas en un proceso o sistema de procesos que se pueden escribir con notación numérica en función del tiempo o la frecuencia (velocidad de rotación de un motor, caudal del líquido en una tubería, volumen o altura del contenido dentro de un estanque, etc.). Se utilizan para modelar y analizar matemáticamente sistemas lineales y no lineales que poseen una o más entradas y salidas (*Multiple Input Multiple Output* o MIMO). Se le da el nombre *estado* a la forma compacta de la información que describe numéricamente la actividad presente o pasada del sistema, con las cuales es posible estimar su comportamiento en el futuro. Formalmente, se puede describir al estado de un sistema como el conjunto más reducido de información numérica tal que su conocimiento en algún instante $t = t_0$, junto con un conjunto de información de igual naturaleza correspondiente a la entrada, determina el comportamiento del sistema para todo $t > t_0$ [34]. Luego, al asociar esta información a una variable algebraica con la cual se pueden realizar cálculos, se obtiene una *variable de estado*. Las variables de estado se escriben normalmente con x y se necesitan tantas como sea el orden del sistema que se busca modelar. Esto da lugar a un *vector de estados* de n elementos cuando el sistema tiene orden n.

El proyecto utiliza el concepto de espacios de estado según su utilidad en área de la ingeniería de control, donde el modelo matemático describe el comportamiento dinámico del sistema en el dominio del tiempo a través de variables de entrada, salida y estado, relacionadas entre sí por una ecuación diferencial matricial de primer orden. En su forma general, para tiempo continuo y con características lineal y variante en el tiempo, el modelo matemático se escribe de la forma en (2.1) y (2.2). En la ecuación (2.1), $x(t)$ es un vector de estados que agrupa en un subconjunto el mínimo de variables de estado linealmente independientes necesarias para la definición del espacio de estados dinámico en un mismo instante de tiempo t, expresado en la dependencia funcional de t sobre x. De manera idéntica, $u(t)$ es un vector de entradas al sistema en ese instante, $A(t)$ es una matriz de estados que asocia cada variable de estado con la razón de cambio de cada elemento de $x(t)$ que da lugar al estado del instante

siguiente, representado por $\dot{x}(t)$, y $B(t)$ es una matriz de entrada que asocia $u(t)$ con $\dot{x}(t)$. En la ecuación (2.2), las variables restantes son el vector de salidas $y(t)$, la matriz de salidas $C(t)$ y la matriz de transmisión directa $D(t)$. Estas últimas dos se encargan de relacionar respectivamente los estados del sistema y las entradas con la salida en un mismo instante. Para un sistema con de orden n, que necesita n variables de estados, junto con p entradas y q salidas, las matrices tienen las siguientes dimensiones: $A(t) \in \mathbb{R}^{n\times n}$, $B(t) \in \mathbb{R}^{n\times p}$, $C(t) \in \mathbb{R}^{q\times n}$ y $D(t) \in \mathbb{R}^{q\times p}$. Para sistemas invariantes en el tiempo, estas matrices son constantes [29].

$$\dot{x}(t) = A(t)x(t) + B(t)u(t) \tag{2.1}$$

$$y(t) = C(t)x(t) + D(t)u(t) \tag{2.2}$$

El enfoque dado a este trabajo es la aplicación de espacios de estado con modelado digital (dominio del tiempo discreto) y análisis para un sistema lineal invariante en el tiempo, donde la salida presenta una no linealidad expresada por la cuantización de la medición de $y(t)$. Según esta descripción, la representación matemática general del espacio de estados que se utiliza está dada por (2.3) y (2.4). Aquí se pierde la dependencia del tiempo en las matrices A, B, C y D, y se expresan los vectores de estado, entrada y salida en el dominio del tiempo discreto por la estructura donde t se ubica a guion bajo con la variable algebraica que representa el vector. Se utiliza esta representación lo que resta del documento.

$$x_{t+1} = Ax_t + Bu_t \tag{2.3}$$

$$y_t = Cx_t + Du_t \tag{2.4}$$

2.2. Estimación de Estados

Una dificultad que existe en el campo de la ingeniería de control es la incapacidad de medir directamente algunas magnitudes físicas involucradas en un sistema dinámico debido a su configuración, o debido a la presencia de perturbaciones (ruido), que convierten la obtención de mediciones con bajo nivel de error en una tarea costosa. Los métodos de estimación de estados existen para resolver matemáticamente esta dificultad.

La tarea de estimar los estados de un sistema permite encontrar de manera probabilística un conjunto de valores que reflejen el comportamiento de sus magnitudes físicas, asociadas a las variables de estado del sistema, a lo largo del tiempo con el mínimo error posible. Su aplicación permite no solo identificar anormalidades en el funcionamiento de procesos no visibles bajo condiciones normales de operación, sino que, además, pueden predecir su comportamiento a largo plazo y determinar la viabilidad del proceso después de ejecutar un número elevado de iteraciones. Conocidos también como métodos y algoritmos de filtraje o suavizado, son usados en numerosas aplicaciones en áreas de diseño de sistemas de control, identificación de parámetros, diagnóstico de fallas, sistemas de potencia, análisis de datos financieros y económicos, redes de sensores, pronóstico, sistemas cyber-físicos, sistemas de navegación, transporte y gestión del tráfico en carreteras, entre otras [30]. El ejercicio que presentan estos métodos es que, en general, resolver el problema de filtraje para sistemas no lineales puede llegar a ser computacionalmente inviable, ya que las funciones de densidad de probabilidad (PDFs) que describen el estado están descritas matemáticamente por estructuras no gaussianas. Es por esta razón que en la literatura se pueden encontrar diferentes alternativas sub-óptimas para lidiar con sistemas no lineales, por ejemplo algunas versiones del filtro de Kalman, filtro de partículas y filtro Suma de Gaussianas.

El proyecto busca evaluar estos métodos de estimación de estados y determinar aquellos que son adecuados para su aplicación sobre un sistema donde una salida real se encuentra cuantizada, con un modelo para su espacio de estados descrito por las expresiones (2.5), (2.6) y (2.7). Adicional a la representación general en (2.3) y (2.4), este espacio de estados considera la presencia de ruido w_t en el estado y v_t en la salida, y un operador $\mathfrak{q}\{\cdot\}$ para informar que la salida medida y_t es la cuantización de la salida real z_t. Los vectores w_t y v_t son ruidos blancos gaussianos de media cero y matriz de covarianza Q y R, respectivamente.

$$x_{t+1} = Ax_t + Bu_t + w_t \tag{2.5}$$

$$z_t = Cx_t + Du_t + v_t \tag{2.6}$$

$$y_t = \mathfrak{q}\{z_t\} \tag{2.7}$$

2.3. Procesos de Markov

Un proceso de Markov es un modelo estocástico que describe una secuencia de eventos, donde la probabilidad de que el estado de cada evento en un instante cualquiera tome valores determinados depende estrictamente del estado adquirido por el mismo conjunto de eventos en el instante anterior [25]. A grandes rasgos, se le da este nombra a aquellos procesos que cumplen con la propiedad de Markov: Ausencia de memoria (*memorylessness*). Esta propiedad dicta que debe ser posible estimar estados futuros basándose exclusivamente en el estado actual, tal que esta estimación tenga la misma validez que aquella que se pueda encontrar conociendo la historia completa del proceso. Al trabajar con un espacio de estados el dominio del tiempo discreto, un proceso de Markov adquiere el nombre de *cadena de Markov*.

Este documento no explora las complejidades de los procesos de Markov, pero es pertinente incluirlo dentro de los conceptos y definiciones, pues describen los procesos y sistemas que interesa estudiar en las próximas secciones. Se busca, dentro de los objetivos del proyecto, realizar el cálculo estadístico del estado de sistemas mediante la medición de datos cuantizados en el instante $t + 1$, solo con la información disponible en el instante t. En particular, los procesos de Markov son la base para varios métodos de simulación estocástica de Monte Carlo, que se emplean para la estimación de estados mediante filtro de partículas.

2.4. Función de Densidad de Probabilidad (PDF)

Los principales métodos de estimación de estados que se estudian aquí no buscan estimar el espacio de estados del sistema de forma determinística, debido a la inviabilidad de dicho enfoque por la presencia de no linealidades en las mediciones de datos. Esto provoca pérdida de información entre diferentes iteraciones que, según su configuración, deberían entregar el mismo valor. En su lugar, se consideran a los conjuntos de información que constituyen las variables de estado del sistema modelado como un proceso estocástico, donde se opera para estimar con una alta probabilidad de coincidir con los valores entregados por el sistema real. La probabilidad asociada a estos valores se conoce como la función de densidad de probabilidad del estado, y se la denota en este documento como PDF.

Formalmente, una PDF permite especificar la probabilidad con la que una variable aleatoria puede caer dentro de un rango de valores particular, en lugar de un único valor. Como se define en (2.8), la probabilidad está dada por la integral de la PDF denotada por $p(x_t)$ sobre el rango especificado $[a, b]$, donde se busca calcular el área bajo la curva de $p(x_t)$, para todo valor de x_t tal que $a \leq x_t \leq b$.

$$P[a \leq x_t \leq b] \quad = \quad \int_a^b p(x_t) d(x_t) \tag{2.8}$$

Una forma muy conocida que puede tomar la PDF de una variable de estado se conoce como distribución normal o gaussiana. En este caso, la curva que representa la PDF en función de la variable aleatoria se organiza simétricamente alrededor de un valor medio μ donde se ubica el máximo global de $p(x_t)$, con una dispersión fija, separada en relación a μ por múltiplos enteros de su desviación estándar σ (evidente en la figura 2.3). Debido a su forma característica, la PDF de una distribución gaussiana se conoce como una campana de Gauss (*Bell-Curve*). Se la denota frecuentemente con la estructura escrita en (2.9), donde $p(x_t)$ se encuentra condicionado por los valores de μ y P, con P siendo la matriz de covarianza de la distribución, tal que $P = \sigma^2$. Se puede calcular siempre utilizando la expresión (2.10). Pro-

blemas de estimación de estados donde las variables aleatorias poseen distribución gaussiana son sencillos de resolver, gracias a que la probabilidad para determinados valores de media y varianza se encuentran tabulados en diferentes configuraciones según las necesidades del problema a resolver, por ejemplo en la tabla Z, que muestra la probabilidad acumulada para distribución normal estándar.

$$p(x_t|\mu, P) = \mathcal{N}(x_t; \mu, P) \tag{2.9}$$

$$p(x_t|\mu, P) = \frac{1}{P\sqrt{2\pi}} e^{0.5\left(\frac{x_t-\mu}{P}\right)^2} \tag{2.10}$$

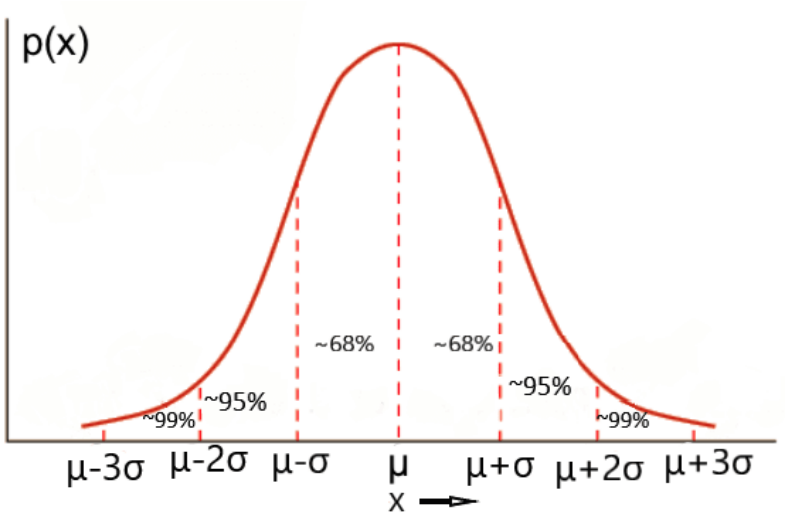


Figura 2.3: Gráfico genérico para una PDF con distribución gaussiana (*Bell-Curve*).

Fuente: Blog Think Simple por usuario ashwinsharmap

La dificultad de los casos estudiados en este documento se presenta al momento de realizar el cálculo de la PDF cuando esta no puede ser representada por un modelo gaussiano. Bajo estas condiciones es necesario emplear algún método que permita encontrar la probabilidad más alta posible y minimizar el error entre el estado estimado y el estado real entregado por el sistema. En particular, el filtro de partículas y el filtro Suma de Gaussianas tienen cualidades

apropiadas para resolver problemas de esta naturaleza, con el segundo funcionando bajo la afirmación que toda distribución de una variable aleatoria se puede escribir como una suma de distribuciones gaussianas.

Por último, al trabajar con variables aleatorias en el dominio del tiempo discreto, se debe calcular la función de masa de probabilidad (*Probability Mass Function* o PMF) en lugar de la PDF. La diferencia entre los dos conceptos está en que la PMF indica la probabilidad de una variable discreta para caer en un valor exacto, por lo que no utiliza integración numérica y está mejor ajustada para un contexto con información digital. Para no perder generalidad, se utiliza el concepto PDF para detallar el objetivo y el modelo matemático de los algoritmos de estimación de estados estudiados que utilizan cálculo estadístico, pero, para espacios de estado en tiempo discreto con datos cuantizados, el procedimiento real es más cercano al cálculo de una sumatoria de múltiples PMFs consecutivas, que forman el mismo rango de posibles valores que puede tomar la PDF en el caso de trabajar con datos continuos.

2.5. No Linealidad por Cuantización de la Salida

Una no linealidad cuya comprensión es esencial para el cumplimiento de los objetivos planteados es la cuantización de los datos debido al método de medición o procesamiento utilizado. Una cuantización en el contexto de la electrónica e ingeniería de control se define como la pérdida de información causada por la incapacidad de un instrumento o dispositivo para procesar entradas continuas (o pseudo-continuas), entregando valores ajustados en la salida según umbrales uniformemente separados. Al recibir un dato en forma de una señal física (energía, posición, etc.), el dispositivo reemplaza el valor entrante proveniente del medio por una aproximación equivalente al umbral directamente superior o inferior (dependiente de la configuración del dispositivo) a dicho valor. Todo dispositivo electrónico presenta cuantización en su salida debido a la naturaleza que posee la información digital, pero en algunos casos, la separación entre cada umbral puede ser lo suficientemente pequeña para obtener una aproximación cercana a un conjunto de datos continuo. Sin embargo, este caso está reservado para dispositivos de alta calidad y costo, por lo que es un ejercicio frecuente encontrar maneras de realizar procesamiento de señales con umbrales grandes. En el contexto de un instrumento de medición digital (sensor), la cuantización se asocia con la exactitud de dicho instrumento.

Ejemplos prácticos de cuantizadores poseen una cantidad finita de niveles de cuantización (*Finite-Level Quantizer* o FLQ), pero teóricamente también es posible trabajar con niveles infinitos (*Ininite-Level Quantizer* o ILQ).

La no linealidad $\mathfrak{q}\{\cdot\} : \mathbb{R} \to \mathcal{V}$, presente en (2.7), representa un ILQ que define la transformación del conjunto de intervalos $\{\mathcal{J}_i \subset \mathbb{R} : i \in \mathcal{I}\}$ al conjunto $\mathcal{V} = \{\nu_i \in \mathbb{R} : i \in \mathcal{I}\}$, separados infinitamente según los índices $\mathcal{I} = \{\ldots, 1, 2, \ldots, L, \ldots\}$. Se puede escribir generalmente con la forma (2.11). La salida y_t del espacio de estados es el resultado de operar sobre cualquier valor de z_t dentro del intervalo $\mathcal{J}_i = \{z_t : q_{i-1} \leq z_t q_i\}$, para obtener su respectivo nivel de cuantización ν_i del conjunto $\mathcal{V}$, donde $i \in \mathcal{I}$ son los índices que dividen el conjunto $\mathbb{R}$. Esto se traduce en un contexto práctico como una lectura del nivel ν_i en todo instante t donde la salida z_t adquiere un valor en entre los umbrales q_{i-1} y q_i. Además, el paso de cuantización lo determina la diferencia entre cada nivel, que es equivalente a la diferencia entre los valores en la frontera q_i y q_{i-1} de cada nivel.

$$\mathfrak{q}\{z_t\} = \begin{cases} \vdots & \vdots & \vdots \\ \nu_1 & \text{si} & q_0 \leq z_t < q_1 \\ \nu_2 & \text{si} & q_1 \leq z_t < q_2 \\ \vdots & \vdots & \vdots \\ \nu_L & \text{si} & q_{L-1} \leq z_t < q_L \\ \vdots & \vdots & \vdots \end{cases} \tag{2.11}$$

Los ejemplos de MATLAB estudiados trabajan con un cuantizador $\mathfrak{q}\{\cdot\}$ definido por la expresión (2.12), que calcula y_t a partir de z_t en términos de la función *round* de MATLAB y un paso de cuantización Δ_q [3]. La función *round(x)* entrega un redondeo al entero más cercano del argumento x (en casos donde la parte fraccionaria sea exactamente 0.5, la función redondea alejándose de cero). Por lo tanto, la salida y_t resultante es el múltiplo entero de Δ_q más cercano a z_t. Con base en esta definición, se puede escribir este cuantizador con la forma en (2.11), donde $\nu_i = \Delta_q i$ y los umbrales de cada intervalo $\mathcal{J}_i$ se calculan como $q_{i-1} = \nu_i - 0.5\Delta_q$ y $q_i = \nu_i + 0.5\Delta_q$ para $i \in \mathcal{I}$.

$$y_t = \Delta_q \text{round}\left\{\frac{z_t}{\Delta_q}\right\} \tag{2.12}$$

El enfoque del proyecto se da a casos con un paso de cuantización Δ_q alto, pero no para casos binarios (el umbral es único y los datos se separan en dos niveles ν_0 y ν_1), pues existe una metodología específica para estimar estados bajo dichas condiciones [4]. Además, varios de los algoritmos que se estudian están diseñados con un enfoque para conjuntos de datos continuos, por lo que es necesario modificar pasos de la secuencia si se desea obtener resultados fiables. En particular, el filtro de Kalman posee una versión diseñada para su funcionamiento con información cuantizada, mientras que en el filtro de partículas requiere una modificación más profunda para que funcione correctamente ante este tipo de no linealidad.

3. Filtro de Kalman (KF)

3.1. Antecedentes Generales

El filtro de Kalman es uno de los algoritmos de filtraje más conocidos y de mayor uso en la literatura, tal que se puede referir a este método como la forma más fundamental para obtener la estimación de los estados, entre los métodos que se estudian en este documento. Posee una implementación simple, que no requiere conceptos matemáticos de alto nivel para su implementación. A pesar de esto, es importante saber manejarlo, pues los resultados obtenidos de su ejecución son de gran utilidad para confirmar la implementación adecuada de otros métodos más complejos.

La teoría detrás del filtro de Kalman fue publicada por primera vez en el año 1960 por Rudolf Emil Kalman, ingeniero eléctrico y matemático húngaro-americano acreditado por el MIT y, en ese entonces, matemático en el Instituto de Investigación para Estudios Avanzados en Baltimore, Maryland, Estados Unidos. El algoritmo descrito corresponde a un conjunto de ecuaciones matemáticas que ofrecen una solución recursiva eficiente del método de mínimos cuadrados en términos de tiempo de ejecución y requerimiento computacional para sistemas lineales con ruido gaussiano, pues no requiere almacenar un número grande de variables para realizar el cálculo del próximo estado. Así, el algoritmo puede estimar el estado de un sistema dinámico en cualquier instante de tiempo. La motivación principal es ofrecer un punto de vista diferente para la solución del problema de filtraje lineal de datos discretos dada por uno de sus predecesores, el Filtro Wiener, con el fin de esquivar algunas de las dificultades que reducen significativamente la utilidad práctica de este método, tal como la falta de transparencia en la matemática del proceso diferencial, la pobre respuesta computacional para la determinación de una respuesta a impulso óptima, entre otras limitaciones de similar naturaleza [20].

Desde la publicación original, debido en gran parte a los avances en la informática digital, el filtro de Kalman ha sido objeto de una amplia investigación y aplicación, especialmente en el área de la navegación autónoma y asistida. Es útil como complemento de sistemas de localización por GPS o sensores, pues no requiere información en tiempo real del entorno y puede estimar estados que no son medibles bajo condiciones normales de operación. Se han formulado versiones diferentes del algoritmo, capaces de cubrir las debilidades que posee el

método original y permitir su aplicación en sistemas que no cumplan con las condiciones de solución óptima, a cambio de reducción de su eficiencia en términos de la exactitud de la estimación y tiempo de ejecución [3]. Versiones destacables son el filtro de Kalman extendido (EKF), filtro de Kalman *unscented* (UKF) y filtro de Kalman cuantizado (QKF), de las cuales solo se implementan las versiones estándar y cuantizadas del algoritmo en este trabajo, pues las otras están diseñadas para aplicaciones que no son relevantes para el problema principal a resolver o poseen varias similitudes con otros métodos que si se explican en profundidad, por lo que los resultados obtenidos terminan siendo redundantes.

3.2. Modelo Matemático

El Filtro de Kalman Estándar se basa en un simple cálculo matricial recursivo en tiempo discreto de las n variables de estado de un sistema dinámico, definidas matemáticamente en un instante t, donde $t = 1, \ldots, N$, por la matriz x_t de orden $n \times 1$, según datos de una matriz de entrada u_t y de salida y_t medibles, que pueden también ser de orden $n \times 1$ si trata de un sistema de MIMO. El procedimiento se separa en dos etapas, la corrección (Measurement Update) y la predicción (Time Update).

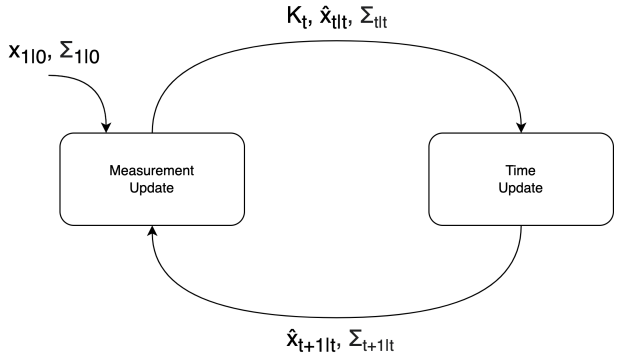


Figura 3.4: Diagrama del algoritmo recursivo del Filtro de Kalman.

Al ejecutar el algoritmo, la solución obtenida es óptima y de mínima varianza [10], solo en los casos donde el modelo matemático del sistema cumple de manera estricta las condiciones mencionadas brevemente en la sección de antecedentes generales: El sistema debe ser lineal y estar sometido a ruido con distribución gaussiana. En términos estadísticos, el método busca

aproximar la PDF de los estados en el momento t, tal que la probabilidad condicional de x_t dado $y_{1:t}$, sea una distribución gaussiana de x_t con media $\hat{x}_{t|t}$ y covarianza $\Sigma_{t|t}$, debiendo cumplirse lo mismo para el siguiente instante, $t+1$. La solución cerrada de las ecuaciones de filtraje en el dominio del tiempo discreto están dadas en (3.13) para el estado actual, y en (3.14) para el próximo estado, donde $\hat{x}_{t|t}$ es el valor del estado estimado por el algoritmo en el instante t dado el estado real en este instante y $\Sigma_{t|t}$ es la covarianza asociada al error de la estimación dada, tal que $\Sigma_{t|t} = cov(x_t - \hat{x}_{t|t})$.

$$p\left(x_t|y_{1:t}\right) \sim \mathcal{N}\left(x_t; \hat{x}_{t|t}, \Sigma_{t|t}\right) \tag{3.13}$$

$$p\left(x_{t+1}|y_{1:t}\right) \sim \mathcal{N}\left(x_{t+1}; \hat{x}_{t+1|t}, \Sigma_{t+1|t}\right) \tag{3.14}$$

Toda instancia del Filtro de Kalman sobre un único sistema empieza con un valor inicial del estado, $x_{1|0} = \mu_1$, y de la covarianza $\Sigma_{1|0} = P_1$. Luego, las dos etapas recursivas del algoritmo y sus respectivas variables de filtraje son:

- **Etapa de Corrección:** En la etapa de corrección el algoritmo comienza con el cálculo de una ganancia K_t en el instante t, conocida simplemente como Ganancia de Kalman (*Kalman Gain*). Previo al filtro de Kalman, lo usual es asignar un valor arbitrario a la ganancia y posteriormente corregirla manualmente en medio de la ejecución en caso de obtener valores que no relacionan apropiadamente el modelo del sistema con las mediciones. El filtro de Kalman optimiza este procedimiento mediante la inclusión de la estimación del instante anterior en la fórmula de la ganancia, permitiendo realizar una corrección activa durante la ejecución y obtener mejor rendimiento. Con una ganancia de Kalman alta, el filtro asigna un peso mayor a las mediciones recientes, y se adapta a estas de manera más receptiva, mientras que una ganancia baja hace que el filtro se ajuste a las predicciones del modelo. El filtro de Kalman calcula la ganancia K_t en el instante t con la ecuación (3.15), correspondiente al producto matricial entre la covarianza $\Sigma_{t|t-1}$ (covarianza en el instante t condicionada por su valor en $t-1$), la

matriz de salida C, y una expresión dada por estas dos magnitudes, más la covarianza del ruido de medición R. Luego, el algoritmo utiliza el valor resultante para calcular la media del estado estimada $\hat{x}_{t|t}$ y la covarianza $\Sigma_{t|t}$ en el instante t con las ecuaciones (3.16) y (3.17) respectivamente. La corrección del estado se expresa en (3.16) por la relación proporcional de la diferencia entre la medición de la salida y_t y el valor del modelo descrito por su espacio de estado, utilizando la medición de la entrada u_t y el estado $\hat{x}_{t|t-1}$, donde la constante de proporcionalidad es igual a K_t. La covarianza se corrige en (3.17) con la diferencia entre la matriz identidad I y el factor K_tC. El objetivo de cada iteración de la etapa de corrección es minimizar la covarianza que se espera obtener en el instante $t+1$ [36].

$$K_t = \Sigma_{t|t-1}C^T\left(R + C\Sigma_{t|t-1}C^T\right)^{-1} \tag{3.15}$$

$$\hat{x}_{t|t} = \hat{x}_{t|t-1} + K_t\left(y_t - C\hat{x}_{t|t-1} - Du_t\right) \tag{3.16}$$

$$\Sigma_{t|t} = \left(I - K_tC\right)\Sigma_{t|t-1} \tag{3.17}$$

- **Etapa de Predicción:** En la etapa de predicción el algoritmo utiliza los valores de $\hat{x}_{t|t}$ y $\Sigma_{t|t}$ estimados en la etapa de corrección para determinar el valor que tomarán estas mismas variables en el instante $t+1$, condicionadas por el valor que toman en t, en otras palabras, sus valores antes de la corrección en $t+1$. Para el próximo estado $\hat{x}_{t+1|t}$ basta con utilizar las funciones lineales del espacio de estados conocidas del sistema modelado para realizar el cálculo según el estado y a la entrada u_t, como la ecuación escrita en (3.18). El cálculo de la próxima covarianza $\Sigma_{t+1|t}$ es igual de simple, utiliza la matriz de estados A y la covarianza del ruido del estado Q para operar la ecuación (3.19).

$$\hat{x}_{t+1|t} = A\hat{x}_{t|t} + Bu_t \tag{3.18}$$

$$\Sigma_{t+1|t} = Q + A\Sigma_{t|t}A^T \tag{3.19}$$

3.3. Implementación de KF

El algoritmo 1 contiene el pseudo-código que representa los pasos del filtro de Kalman estándar. Su estructura se basa en el pseudo-código para la versión extendida del método, sacado de la referencia [5].

Algorithm 1 Filtro de Kalman Estándar

1: **Input:** PDF del estado inicial $p(x_1)$, con $x_{1|0} = \mu_1$, y $\Sigma_{1|0} = P_1$.
2: **for** $t = 1 \rightarrow N$ **do**
3: Calcular K_t según (3.15).
4: Calcular $\hat{x}_{t|t}$ según (3.16).
5: Calcular $\Sigma_{t|t}$ según (3.17).
6: Calcular $\hat{x}_{t+1|t}$ según (3.18).
7: Calcular $\Sigma_{t+1|t}$ según (3.19).
8: **end for**
9: **Output:** $\hat{x}_{t|t}$ y $\Sigma_{t|t}$ para $t = 1, ..., N$.

3.3.1. Ejemplo de Implementación de KF utilizando MATLAB

El filtro de Kalman estándar posee una implementación sencilla dentro del ambiente de programación MATLAB. Gracias a que se trata de una plataforma centrada en el cálculo numérico, las expresiones (3.15), (3.16), (3.17), (3.18) y (3.19) se pueden escribir de forma directa, sin la necesidad de aplicar estructuras particulares para operar sobre arreglos multi-dimensional, como el caso de matrices de orden mayor a 1.

El ejemplo proporcionado por Angel L. Cedeño se estructura de la siguiente manera. El método completo se encuentra en un archivo de función, que contiene la declaración *[F] = standard_ kalman(G)*. Esta función recibe como argumento de entrada una estructura G, que contiene los parámetros del modelo del espacio de estados y vectores con la simulación del estado xt y salida yt, y entrega como salida una estructura F con vectores de la estimación del estado y covarianza en todo instante t, donde $t = 1, \ldots, N$. El primer elemento de estos vectores se iguala a las variables xin y Pin, almacenadas en G. El resto es un bucle *for* de

N interacciones (N se almacena en G) escrito en el segmento de código (3.1), donde cada iteración opera la ganancia, la etapa de corrección y la etapa de predicción en este instante. Los resultados son almacenados en los vectores correspondientes, xtt y Stt para el estado y covarianza corregido, y $xtmt$ y $Stmt$ para la predicción en $t+1$.

Código 3.1 : Cómputo de estados con KF (MATLAB)

```
for t=1:1:G.sim.N
    % Kalman Gain
    F.K{t}=(F.Stmt{t}*G.mm.C')/(G.mm.R+G.mm.C*F.Stmt{t}*G.mm.C');
    % Measurement Update
    F.xtt{t}=F.xtmt{t}+F.K{t}*(G.sim.yt{t}-G.mm.C*F.xtmt{t}-G.mm.D*G.sim.ut{t});
    F.Stt{t}=(eye(length(G.mm.C))-F.K{t}*G.mm.C)*F.Stmt{t};
    % Time Update
    F.xtmt{t+1}=G.mm.A*F.xtt{t}+G.mm.B*G.sim.ut{t};
    F.Stmt{t+1}=G.mm.Q+G.mm.A*F.Stt{t}*G.mm.A';
end
```

El código completo para la implementación del filtro de Kalman estándar en MATLAB se puede encontrar en el anexo 9.43.

3.3.2. Ejemplo de Implementación de KF escrito en Python

La secuencia de pasos para implementar el filtro de Kalman en Python son similares al caso descrito para MATLAB. Sin embargo, como el lenguaje está orientado a la legibilidad de código, diseñadores optan por crear funciones separadas para operaciones que se repiten varias veces dentro de un programa. Esta filosofía no es diferente para la mayoría de las implementaciones publicadas de este método. Además, como Python no posee una sintaxis nativa para realizar cálculos matemáticos multidimensionales, es necesario emplear bibliotecas de uso libre que contengan herramientas que compensen esta debilidad.

Entre las implementaciones estudiadas durante la etapa de investigación del proyecto, el ejemplo escogido para evaluar y comparar el rendimiento del filtro de Kalman estándar en Python proviene del repositorio *state_estimation* de Github, publicado por el usuario *cggos* o Gavin Gao. Aquí, Gavin Gao ofrece diferentes métodos para solucionar el problema de estimación implementados en diferentes plataformas y lenguajes de programación (MATLAB,

Python, C++, etc.). En esta sección interesa estudiar su diseño del filtro de Kalman, el cual emplea para resolver problemas de localización en coordenadas cartesianas (x,y), a base de una matriz de mediciones aleatorias.

Código 3.2 : Cómputo de estados con KF (Diseño de Gavin Gao)

```
# Number of iterations in Kalman Filter
N_iter = 100

# Applying the Kalman Filter
for i in arange(0, N_iter):
    (X, P) = kf_predict(X, P, A, Q, B, U)
    (X, P, K, IM, IS, LH) = kf_update(X, P, Y, H, R)
    Y = array([[X[0,0] + abs(0.1*randn(1)[0])], [X[1,0] + abs(0.1*randn(1)[0])]])
    x1.append(X[0])
    x2.append(X[1])
    y1.append(Y[0])
    y2.append(Y[1])
```

El código utiliza dos funciones para realizar la etapa de corrección y predicción en cada iteración, desde 1 hasta N_{iter} con el bucle *for* del segmento de código (3.2), y una función para calcular la PDF de las variables objetivo. La primera función que se ejecuta es *kf_ predict(X, P, A, Q, B, U)*, que se puede ver en (3.3). Recibe un estado X, las covarianzas P y Q, las matrices A y B, y la entrada U para operar según las fórmulas (3.18) y (3.19), y entregar un estado X y covarianza P del siguiente instante. El filtro de Kalman no es estricto sobre el orden de la corrección y predicción, sin embargo el cómputo de los datos debe ser consistente a lo largo de la ejecución.

Código 3.3 : Predicción de estados en KF (Diseño de Gavin Gao)

```
def kf_predict(X, P, A, Q, B, U):
    X = dot(A, X) + dot(B, U)
    P = dot(A, dot(P, A.T)) + Q
    return (X, P)
```

La otra función de interés, *kf_ update(X, P, Y, H, R)*, se puede ver en (3.4) y funciona de forma similar. Se opera con las fórmulas (3.15), (3.16) y (3.17), donde las matrices H y R asocian la salida Y con X y U, y son equivalentes a C y D en (2.6). Los objetivos del proyecto no incluyen mostrar el gráfico de las PDFs de las variables objetivo, por lo que la función *gauss_ pdf(Y, IM, IS)* se puede omitir.

Código 3.4 : Corrección de estados en KF (Diseño de Gavin Gao)

```
def kf_update(X, P, Y, H, R):
    IM = dot(H, X)
    IS = R + dot(H, dot(P, H.T))
    K = dot(P, dot(H.T, inv(IS)))
    X = X + dot(K, (Y-IM))
    P = P - dot(K, dot(IS, K.T))
    LH = gauss_pdf(Y, IM, IS)
    return (X, P, K, IM, IS, LH)
```

En su forma actual, este código es apropiado para realizar los experimentos necesarios. La única modificación que se debe realizar es al vector de mediciones Y, para incluir el efecto del cuantizador $\mathfrak{q}\{\cdot\}$ presente en (2.7).

El código completo del diseño de Gavin Gao para el filtro de Kalman estándar se puede encontrar en el anexo 9.52.

3.4. Filtro de Kalman Cuantizado (QKF)

El filtro de Kalman cuantizado es una versión modificada del algoritmo estándar previamente descrito, diseñado para incluir el efecto del cuantizador $\mathfrak{q}\{\cdot\}$ en el espacio de estados que modela la salida y_t descrita por (2.7), según su estructura en (2.11). Para conseguir una respuesta adecuada ante mediciones con este tipo de no linealidad, se realiza una modificación en la ecuación (3.16), que entrega el valor corregido de la media estimada de la PDF del estado en el instante t, a base de la diferencia entre la salida medida y la salida modelada por el espacio de estados con mediciones de la entrada. En la literatura se describen dos formas para considerar el efecto de la cuantización [16][22]. La primera forma está dada por la ecuación (3.20), donde se aplica el cuantizador $\mathfrak{q}\{\cdot\}$ a la salida modelada para obtener un conjunto

de datos modelados con los mismos niveles de cuantización que la salida. La segunda forma se expresa en la ecuación (3.21), ahora aplicando $\mathfrak{q}\{\cdot\}$ tanto a la salida modelada como a la salida medida. La razón por la cual ambas formas resultan en la misma corrección del estado es que el conjunto de datos y_t ya se encuentra cuantizado, por lo tanto, $\mathfrak{q}\{y_t\} = y_t$. Luego, si se definen los niveles de cuantización para un paso Δ_q según la expresión (2.12), tal que $\nu_i = \Delta_q i$, se puede afirmar que $y_t - \mathfrak{q}\left\{C\hat{x}_{t|t-1} - Du_t\right\} = \mathfrak{q}\left\{y_t - \left(C\hat{x}_{t|t-1} - Du_t\right)\right\}$.

$$\text{Forma 1:} \quad \hat{x}_{t|t} = \hat{x}_{t|t-1} + K_t\left(y_t - \mathfrak{q}\left\{C\hat{x}_{t|t-1} - Du_t\right\}\right) \tag{3.20}$$

$$\text{Forma 2:} \quad \hat{x}_{t|t} = \hat{x}_{t|t-1} + K_t\mathfrak{q}\left\{y_t - \left(C\hat{x}_{k|k-1} - Du_t\right)\right\} \tag{3.21}$$

3.5. Implementación de QKF

3.5.1. Ejemplo de Implementación de QKF utilizando MATLAB

La implementación del filtro de Kalman cuantizado en MATLAB corresponde a una leve modificación del algoritmo utilizado para la versión estándar, donde se cambia la operación dada por (3.16) en la etapa de corrección para ajustarse a la forma 2, dada por (3.21), y se escribe en el código como (3.5).

Código 3.5 : Cómputo de estados con QKF (MATLAB)

```
1       F.xtt{t}=F.xtmt{t}+F.K{t}*(G.sim.yt{t}-Q(G,G.mm.C*F.xtmt{t}+G.mm.D*G.sim.ut{t}));
```

Para no perder la legibilidad del código, se implementa una función auxiliar *q=Q(G,zt)*, escrita en (3.6), que recibe la estructura G y una muestra zt, y entrega el valor cuantizado q de la muestra según $\mathfrak{q}\{\cdot\}$, Estos ejemplos cuantizan la salida del espacio de estados modelado utilizando la definición del cuantizador dada por (2.12), por lo que la misma expresión debe ser usada en la etapa de corrección del algoritmo implementado. El paso de cuantización lo define la variable *Delta* almacenada en G.

Código 3.6 : Cuantización de datos mediante función *round* (MATLAB)

```
function q=Q(G,zt)
    q=G.mm.Delta*round(zt/G.mm.Delta);
end
```

El anexo 9.44 muestra la modificación realizada al algoritmo en 9.43 para implementar el filtro de Kalman cuantizado.

3.5.2. Ejemplo de Implementación de QKF escrito en Python

Análogo al caso de MATLAB, la implementación en Python del filtro de Kalman cuantizado consiste en modificar la función *kf_ update()* del código de Gavin Gao ecrito en (3.4) para incluir el efecto de la cuantización según (3.21), con un cuantizador definido por (2.12). Se emplea una función *np.around(X)* para redondear la expresión y obtener un valor cuantizado dado por el paso de cuantización *Delta*. El segmento de código modificado se puede observar en (3.7). Debido a que no se implementan funciones nuevas, esta versión del código no está incluida en los anexos.

Código 3.7 : Computo del estado con QKF (Diseño de Gavin Gao)

```
    X = X + dot(K, Delta*np.around(dot(Y-IM,1/Delta)))
```

4. Filtro de Partículas (PF)

4.1. Antecedentes Generales

Dejando detrás los fundamentos del filtro de Kalman, ahora es necesario cubrir más de los requerimientos que debe tener la solución del problema planteado para la estimación de estados. Para resolver el problema de la salida cuantizada y ruidos de distribución no gaussiana se requieren métodos más adaptativos, sin estar condicionados por características del sistema que deban ser conocidas a priori. Esta sección del documento estudia las cualidades de los métodos secuenciales probabilísticos con enfoque bayesiano para la estimación de estados, conocidos como filtros de partículas, y sus cualidades que les permiten aproximar la PDF de las variables de estado pertenecientes a sistemas dinámicos, sin conocimiento previo de su naturaleza.

Los filtros de partículas, también conocidos como métodos secuenciales de Monte Carlo, son el nombre otorgado a un vasto conjunto de algoritmos que buscan estimar el estado de un sistema dinámico utilizando solo el cálculo probabilístico de procesos completamente estocásticos. El nombre *filtro de partículas* fue usado por primera vez en 1960 por el investigador francés, Pierre Del Moral, en el área de la mecánica de fluidos [7], mientras que "Secuencia de Monte Carlo" proviene de los profesores de estadística de las Universidades de Harvard y Rutgers, Jun Liu y Rong Chen, en el año 1998 [23], quienes tomaron inspiración en el diseño del filtro de Monte Carlo propuesto por Genshiro Kitagawa, matemático japones de la Universidad de Tokyo, en 1993 [25]. En el mismo año que Kitawaga, el físico americano, Jay Gordon, publica su trabajo sobre algoritmos de estadística bayesiana [17]. Todos estos trabajos formarían la base de lo que es estudia en este documento sobre el filtro de partículas.

Estos métodos son una propuesta para solucionar el problema de filtraje con la motivación de calcular la probabilidad a posteriori de los estados de un proceso de Markov [25] bajo condiciones de observabilidad parcial y ruidos aleatorios cuya distribución no se puede determinar fácilmente. Una de sus mayores ventajas es que el modelo del espacio de estados con el que se desea trabajar puede ser no lineal, y el estado inicial y la distribución del ruido no presentan una limitación en las aplicaciones del algoritmo. La metodología de los filtros de partículas se encuentra bien documentada, en particular los procedimientos disponibles

para el muestreo de las partículas sin la necesidad de recurrir a suposiciones en la mayoría de los casos, aunque se incluye en el documento el estudio de un caso especifico donde realizar una suposición resulta inevitable. La mayor desventaja de estos filtros es que, si bien poseen un alto nivel de adaptabilidad, su rendimiento al ser aplicado en sistemas de orden elevado disminuye drásticamente, pues requieren una gran cantidad de recursos computacionales para almacenar sus variables de estado aproximadas durante el cálculo recursivo. A pesar de esto, resultan un buen método cuando el enfoque de la aproximación es disminuir el error de la PDF para cualquier sistema, con excepción de sistemas lineales con distribución gaussiana, donde el Filtro de Kalman siempre resulta óptimo.

4.2. Modelo Matemático

Los filtros de partículas buscan aproximar directamente los estados del sistema en todo instante t mediante la propagación de muestras (o *partículas*) y pesos a lo largo de las funciones no lineales del sistema, representados en el modelo por x_t y κ_t respectivamente, sin necesidad de conocer su distribución estadística. Esto lo diferencia enormemente del Filtro de Kalman, pues su rango efectivo de aplicaciones no se encuentra limitado por la naturaleza del sistema si se desea encontrar una solución cerrada óptima. La expresión (4.22) describe la aproximación de la PDF de fitraje generada por este método, correspondiente a la suma de M partículas muestreadas y ponderadas por los pesos, donde $x_t^{(i)}$ es la i-ésima partícula de la suma y $\omega_t^{(i)}$ es el i-ésimo peso. Cada componente de esta suma se encuentra condicionado por una función delta de Dirac $\delta(\cdot)$, distinta de cero estrictamente cuando la i-ésima partícula coincide con el valor del estado x_t.

$$p\left(x_t|y_{1:t}\right) \approx \sum_{i=1}^{M} \omega_t^{(i)} \delta\left(x_t - x_t^{(i)}\right) \tag{4.22}$$

Debido a que la distribución del sistema no es conocido, tomar muestras de la PDF del sistema resulta difícil. Para resolver esta dificultad, el algoritmo basa su modelo matemático en

la expresión (4.23), que propone calcular los pesos de manera recursiva utilizando distribución de importancia para generar la partículas. Según este principio, se tiene que la relación entre el i-ésimo peso ω_t en el instante t es proporcional al producto entre el peso calculado en el instante $t-1$ y otro factor, compuesto el producto de la PDF de la salida y la PDF de transición de estados, normalizadas por la distribución de importancia $q\left(x_t|x_{t-1}^{(i)}, y_t\right)$.

$$\omega_t^{(i)} \propto \omega_{t-1}^{(i)} \frac{p\left(y_t|x_t^{(i)}\right) p\left(x_t^{(i)}|x_{t-1}^{(i)}\right)}{q\left(x_t|x_{t-1}^{(i)}, y_t\right)} \tag{4.23}$$

La ventaja de utilizar distribución de importancia es que esta es óptima y minimiza la varianza de los pesos ω_t cuando se cumple (4.24), donde la distribución de importancia es igual a la PDF de transición de estados. [8][30].

$$q\left(x_t|x_{t-1}^{(i)}, y_t\right) = p\left(x_t|x_{t-1}^{(i)}\right) \tag{4.24}$$

El problema que ocurre en esta instancia es que la distribución de importancia óptima es igual de difícil de medir que la PDF de los estados, a menos que se cumplan condiciones específicas como el caso del filtro de Kalman, donde la distribución del estado y la salida es siempre gaussiana. La solución proviene del algoritmo *filtro Bootstrap*, el cual propone trabajar con base en la suposición que se cumple esta igualdad en todo instante t, tal que los cálculos de los pesos estén dados por (4.26) [8][17][24]. Esto es válido porque en cálculo estadístico siempre se puede llegar a una aproximación de la PDF con un error reducido si se cuenta con una muestra lo suficientemente grande. Entonces, aunque la suposición sea incorrecta a corto plazo, el ajuste de los pesos permite estimar la PDF del sistema al tomar solo las partículas donde si se cumple, entregando una estimación aceptable si dicho número de partículas es alto.

$$\omega_t^{(i)} \propto \omega_{t-1}^{(i)} p\left(y_t | x_t^{(i)}\right) \tag{4.25}$$

$$\omega_t^{(i)} = \frac{p(y_t|x_t^{(i)})}{\sum_{j=1}^{M} p(y_t|x_t^{(j)})} \tag{4.26}$$

Esta elección produce un filtro de partículas con una implementación sencilla. Sin embargo, la efectividad del algoritmo ahora está condicionada por el número de partículas necesario para obtener una estimación con un nivel de exactitud aceptable. Un sistema de alto orden y/o un número de particular elevado requiere un alto costo computacional asociado a la cantidad de datos que deben ser almacenados durante cada iteración del cálculo recursivo, lo cual puede resultar en una limitación de las posibles aplicaciones del filtro.

Otro problema inevitable que poseen estos métodos se conoce como "fenómeno de degeneración" (*degeneracy phenomenon*) [8], que describe la cualidad que poseen los pesos de aumentar su varianza en el tiempo. Esto hace que, después de un cierto número de iteraciones, las partículas tengan un peso despreciable, y por lo tanto el algoritmo le asigna recursos computacionales a la actualización de partículas cuya contribución a la estimación final es cercana a cero [27], desperdiciando dichos recursos. Se propone una solución al fenómeno de degeneración, donde las partículas con pesos pequeños son eliminadas y las partículas con pesos grandes son replicadas en cada iteración, generando un nuevo conjunto de partículas con una distribución equitativa de los pesos [17]. El documento se refiere a este procedimiento como *resampling*.

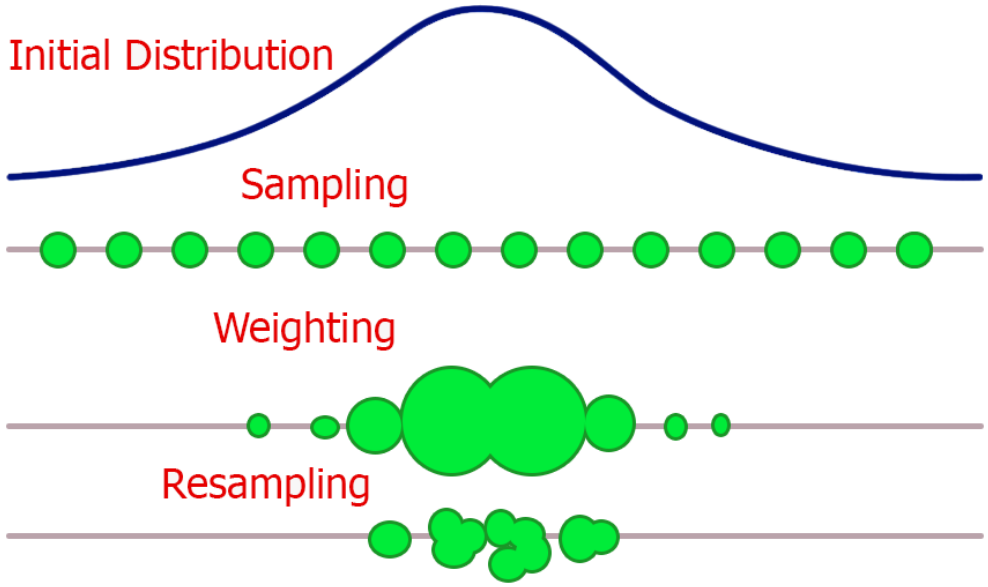


Figura 4.5: Representación gráfica del proceso de muestreo de partículas con pesos para filtro de partículas.

Fuente: Martin Dimitrov, 2021, lancaster.ac.uk

Existen diferentes técnicas para realizar resampling, por lo que la implementación del algoritmo puede elegir la que mejor se ajuste al sistema con el que se trabaja. Sin embargo, el método elegido puede sufrir de un problema que puede necesitar ser abordado conocido como empobrecimiento de las partículas (*particle impoverishment*), donde ocurre una pérdida en la diversidad del conjunto muestreado debido a la repetición de partículas pesadas [37], perjudicando la capacidad del algoritmo para aproximar estadísticamente el estado del sistema con las mediciones disponibles. La literatura sugiere algunos métodos que pueden ser incluidos en resampling para reducir el problema de empobrecimiento en una distribución de probabilidad [33], por ejemplo, la técnica de Monte Carlo basado en cadenas de Markov (*Markov Chain Monte Carlo* o MCMC) para agregar variabilidad a las nuevas muestras [9]. La elección tiende a depender de la complejidad del problema de estimación.

4.3. Solución ante el Problema de Salida Cuantizada

A fin de no perder exactitud en la estimación al implementar el filtro de partículas cuando la salida es un conjunto de datos cuantizados, es necesario incluir un paso adicional para considerar la no linealidad $\mathfrak{q}()$ de (2.7) en el cálculo recursivo del algoritmo. Como la salida y_t es una variable aleatoria discreta y discontinua debido a la cuantización, el modelo probabilístico de la expresión $p\left(y_t|x_t^{(i)}\right)$ es una PMF. Entonces, la definición de la PDF no es válida para calcular los pesos $\kappa_t^{(i)}$ según (4.25). La solución ante este problema se plantea en [3] para distribución de filtraje recursivo con enfoque bayesiano, dada la regla de PDF generalizada (*Generalized Probability Density function*) [30], que es válida para distribuciones discretas y absolutamente continuas.

$$p\left(y_t|x_t^{(i)}\right) = \int_{a_t}^{b_t} \mathcal{N}\left(v_t^{(i)}; 0, R\right) dv_t \tag{4.27}$$

$$p\left(y_t|x_t^{(i)}\right) = \mathcal{N}\left(b_t^{(i)}; 0, R\right) - \mathcal{N}\left(a_t^{(i)}; 0, R\right) \tag{4.28}$$

Las expresiones (4.27) y (4.28) describen una ecuación de integración que resuelve el problema del cálculo recursivo de los pesos $\omega_t^{(i)}$. Está basada en el principio que la PDF de $y_t|x_t$ se puede aproximar por una función de distribución acumulativa (*Cumulative Distribution Function* o CDF) de una distribución gaussiana multivariable con media cero y matriz de covarianza R, integrada entre los límites resultantes de las funciones a_t y b_t. La tabla 4.1 define la dependencia de las funciones según el nivel ν_i del conjunto $\mathcal{V}$ donde se ubica la salida y_t en el instante t. Los límites se reemplazan por la expresión correspondiente asociada al valor obtenido en la salida, en función del estado x_t y la entrada u_t en el mismo instante con sus respectivas matrices de asociación escritas en (2.6), y los umbrales del i-ésimo intervalo $\mathcal{J}_i$ para un cuantizador definido por (2.12). Además, se hace distinción en la definición de a_t y b_t si el cuantizador es FLQ o ILQ [5].

En la mayoria de los casos, y en particular en los casos que son de interés para este proyecto, la PDF de la variable $y_t|x_t$ en la ecuación (4.27) posee una distribución no gaussiana y, por lo tanto, genera PDFs de corrección y predicción que también son no gaussianas. Esto causa una pérdida significativa de exactitud en la estimación al utilizar la aproximación de la PDF en algoritmos diseñados bajo la suposición que dichos valores son gaussianos, por ejemplo el filtro de Kalman y sus versiones. Sin embargo, el filtro de partículas y el filtro Suma de gaussianas no trabajan bajo esta suposición, por lo que la aproximación es válida.

	y_t	a_t	b_t
FLQ	ν_1	$-\infty$	$q_1 - Cx_t - Du_t$
	ν_i, $i = 2, \ldots, L-1$	$q_{i-1} - Cx_t - Du_t$	$q_i - Cx_t - Du_t$
	ν_L	$q_L - Cx_t - Du_t$	∞
ILQ	ν_i, $i = \ldots, 1, \ldots, L, \ldots$	$q_{i-1} - Cx_t - Du_t$	$q_i - Cx_t - Du_t$

Tabla 4.1: Límites de integración para cálculo de pesos con datos cuantizados.

4.4. Implementación de PF

El algoritmo 2 contiene el pseudo-código que representa los pasos del filtro de partículas para estimar con una salida cuantizada, según los dicta el modelo matemático.

Algorithm 2 Filtro de Partículas

1: **Input:** PDF del estado inicial $p(x_1)$, con N_{par} partículas.
2: Tomar muestras de $x_t^{(i)} \sim p(x_1)$, con $i = 1, ..., N_{par}$.
3: **for** $t = 1 \rightarrow N$ **do**
4: Tomar muestras de $x_t^{(i)} \sim p(x_t|x_{t-1}^{(i)})$, según (4.24), correspondiente a distribución de importancia con Filtro Bootstrap, con $i = 1, ..., N_{par}$.
5: Calcular los pesos $\omega_t^{(i)}$ usando $p\left(y_t|x_t^{(i)}\right)$ dado en (4.28) y normalizar según (4.26), con $i = 1, ..., N_{par}$.
6: Realizar resampling de la partículas para generar $x_l^{(j)}$, con $j = 1, ..., N_{par}$. Se cumple que: $P(x_t^{(j)} = \tilde{x}_t^{(i)}) = \omega_t^{(i)}$, con $i = 1, ..., N_{par}$.
7: **end for**
8: **Output:** Conjunto de partículas $\left(x_t^{(j)}\right)_{j=1}^{N_{par}} \sim p(x_t|y_{1:t})$.

4.4.1. Ejemplo de Implementación de PF utilizando MATLAB

El filtro de partículas posee una implementación sencilla dentro de MATLAB, sin embargo, en un sentido diferente a las versiones del filtro de Kalman previamente explicadas. Mientras que la simplicidad del filtro de Kalman proviene de la interpretación determinista que le da este proyecto, el filtro de partículas se vuelve sencillo, pues varios de sus pasos pueden ser reducidos por aproximaciones debido a la naturaleza cuantizada de los datos con los que se está trabajando. A pesar de esto, el ejemplo utiliza algunas herramientas disponibles en MATLAB que requieren explicación.

El ejemplo que se utiliza como referencia fue proporcionado por Angel L. Cedeño. Comienza con la declaración de una función *[PF] = particle_filter(G)*, la cual recibe como entrada la misma estructura G que el filtro de Kalman y entrega como salida una estructura PF con la información de la estimación en cada instante t, donde $t = 1, \ldots, N$. Antes de comenzar con la secuencia, se inicializan las partículas con las que se realiza la estimación utilizando el código escrito en (4.8). La técnica para tomar muestras según distribución de importancia consiste en repetir el estado inicial xin para $1 \times Npar$ partículas mediante la función *repmat(xin,1,Npar)*, y luego sumar ruido a estas partículas con distribución gaussiana y covarianza dada por la descomposición de Cholesky de la covarianza inicial Pin.

Código 4.8 : Cómputo de partículas iniciales para PF (MATLAB)

```
1    xpar = repmat(G.mm.Icon.xin,1,G.sim.Npar)+chol(G.mm.Icon.Pin)*randn(n,G.sim.Npar);
```

La descomposición de Cholesky es un método empleado en simulaciones de Monte Carlo para que un conjunto de valores aleatorios multidimensionales tenga la misma covarianza asignada que el modelo del sistema. La función *chol(Pin)* sigue la fórmula (4.29), entrega una matriz triangular inferior L, tal que el producto entre L y su transpuesta es Pin, siempre que Pin sea definida positiva.

$$P_{in} = LL^T \tag{4.29}$$

El ejemplo procede a realizar la estimación en un bucle *for* de N iteraciones. Primero calcula los pesos según la expresión (4.28) con el segmento de código escrito en (4.9), donde los límites de integración a_t y b_t (escritos como lim_a y lim_b en el código) se definen según la tabla 4.1 para un cuantizador dado por (2.12). Aquí, x_t son todas las partículas x_{par} y u_t es la entrada del modelo en el instante t. Los umbrales q_{i-1} y q_i del intervalo $\mathcal{J}_i$ con paso de cuantización $Delta$ están dados por los arreglos a y b, inicializados al comienzo del algoritmo con $a = y_t - 0.5Delta$ y $b = y_t + 0.5Delta$. La distribución gaussiana de (4.28) se realiza con la función *mvncdf(X,0,R)*, que retorna un vector con la CDF de la distribución gaussiana multivariable de media cero y covarianza R, evaluada en cada fila de la matriz X. Los pesos resultantes se normalizan con la suma de sus elementos, de acuerdo a la fórmula (4.26).

Código 4.9 : Cómputo de pesos en PF (MATLAB)

```
lim_a = repmat(a(t),1,G.sim.Npar) - G.mm.C*xpar-G.mm.D*G.sim.ut{t};
lim_b = repmat(b(t),1,G.sim.Npar) - G.mm.C*xpar-G.mm.D*G.sim.ut{t};
w = transpose(mvncdf(lim_b',0,G.mm.R) - mvncdf(lim_a',0,G.mm.R));
w = w/sum(w);
```

El siguiente paso es el resampling con el método sistemático de las muestras obtenidas para las partículas $xpar$ y los pesos w [33]. Esta técnica consiste en dividir la población de partículas en M sub-poblaciones, tal que el peso acumulado de la n-ésima sub-población es menor a $u^{(n)}$, donde $n = 1, \ldots, M-1$ y $M = Npar$. Los marcadores $u^{(n)}$ se calculan con las expresiones (4.30) y (4.31), con $u^{(0)}$ sacado de la distribución uniforme en el intervalo $(0, 1/M]$ y los u^n subsecuentes calculados de forma determinística.

$$u^{(0)} \sim U\left(0, \frac{1}{M}\right] \tag{4.30}$$

$$u^{(i)} = u^{(0)} + \frac{n}{M}, \qquad n = 1, \ldots, M-1 \tag{4.31}$$

La función *xpar=systematic_resample(w,par)* realiza este proceso registrando el último índice del vector de suma acumulada (calculado con la función *cumsum(w)*) de los pesos w recibidos como entrada, donde se cumple que el valor acumulado es menor al $u^{(n)}$ actual. Una vez que el vector *index* contiene todos los índices donde ocurrió la división, la función extrae los valores de las partículas en la entrada *par* ubicados en los índices registrados y los copia en un nuevo vector de partículas *xpar*, el cual entrega como salida. Este proceso le otorga variabilidad a las muestras en *xpar*, pues poseen pesos equidistantes según la relación $1/M$.

La secuencia finaliza reiniciando el valor de los pesos w y normalizándolos por $1/Npar$. Se realiza el producto punto entre los pesos normalizados y las partículas después del resampling, con la suma de los elementos del resultando siendo el estado estimado en el instante t. El segmento de código para realizar estos pasos se pueden ver en (4.10).

Código 4.10 : Cómputo del estado con PF (MATLAB)

```
1    w=(1/G.sim.Npar)*ones(1,G.sim.Npar);
2    PF.xtt{t} = sum(w.*xpar,2);
```

Antes de comenzar con la siguiente iteración, el algoritmo ejecuta la línea de código (4.11) para inicializar nuevamente las partículas con los parámetros del modelo y valores de entrada. Se reemplazan la entrada *xin* con el estado en el instante $t+1$ según la expresión (2.5), donde x_t corresponde es reemplazado por las partículas después del resampling en el instante t.

Código 4.11 : Inicialización de las partículas para intante $t+1$ (MATLAB)

```
1    xpar = G.mm.A*xpar + G.mm.B*G.sim.ut{t} + chol(G.mm.Q)*randn(n,G.sim.Npar);
```

El código completo para la implementación del filtro de partículas en MATLAB y la función de resampling sistemático se pueden encontrar en los anexos 9.45 y 9.46.

4.4.2. Ejemplo de Implementación de PF escrito en Python

La implementación en Python del filtro de partículas no varía significativamente con respecto a la implementación en MATLAB descrita en la sección anterior, el cálculo matemático es el mismo, pero las estructuras utilizadas y la organización del código pueden variar dependiendo de las preferencias del programador. Una dificultad presente en esta parte del trabajo es que se desconoce si existen implementaciones libremente publicadas que se encuentren optimizadas ante condiciones de cuantización en la medición, por lo que la elección debe tomar en cuenta la necesidad realizar modificaciones.

El ejemplo seleccionado proviene del mismo usuario que diseñó la implementación del filtro Kalman explicada en su respectiva sección, Gavin Gao, como parte de su repositorio *state_ estimation*. Se elige este diseño, pues contiene código considerablemente más fácil de leer y examinar, en comparación a varios ejemplos del mismo método encontrados en línea. En su código, el programador decide contener el proceso completo de definición de partículas, cómputo de pesos, resampling y cómputo del estado estimado de en un único archivo *robotics_ localization.py*. Una desventaja es que el código define cada operación matemática como un sistema de orden 4, por lo que es necesario modificar cada una de ellas si se desea trabajar con un modelo de menor orden, o trabajar con el sistema de orden 4 y definiendo un elevado número de coeficientes como 0.

El diseño de Gavin Gao utiliza el filtro de partículas para estimar la posición de un cuerpo según observaciones aleatorias realizadas en tiempo real mientras se ejecuta el proceso de estimación. El bucle *for* principal, escrito en (4.12), realiza dos tareas en cada iteración, desde 1 a N. Primero calcula un estado $xTrue$, una salida z y un estado para realizar *Dead Reckoning*. Este último describe un procedimiento para calcular posición en base a estimaciones realizadas en todo instante t pasado. Esto no es relevante para el proyecto, por lo que no se dará atención a esta variable de aquí en adelante. La salida z obtenida, junto con la entrada ud, se utilizan para calcular los estados $xEst$ y covarianza $PEst$ estimados para cualquier instante. Los arreglos px y pw son de largo NP y se utilizan para almacenar las partículas y los pesos en cada iteración del algoritmo, y $hxEst$ contiene todos los estados $xEst$ estimados una vez que finaliza el proceso.

Código 4.12 : Cómputo de estado con PF (Diseño de Gavin Gao)

```
    for i in range(N):
        u = calc_input()
        xTrue, z, xDR, ud = observation(xTrue, xDR, u, RFID)
        xEst, PEst, px, pw = pf_localization(px, pw, z, ud)
        hxEst = np.hstack((hxEst, xEst))
        hxDR = np.hstack((hxDR, xDR))
        hxTrue = np.hstack((hxTrue, xTrue))
```

En el diseño, dos funciones son de interés para el proyecto, *pf_localization(px, pw, z, u)* y *resampling(px, pw)*. La función *pf_localization(px, pw, z, u)* recibe la salida observada z y una entrada u, y entrega el estado estimado en un instante. Las diferencias que se pueden identificar en esta parte con respecto a la implementación de MATLAB son, en primer lugar, las partículas son inicializadas con valor 0 y se les suma ruido gaussiano a cada partícula por separado, dentro de un bucle *for* que va de 1 a NP. Los pesos se inicializan en 1, y se calculan bajo la suposición que las observaciones son gaussianas, con media igual al diferencial de posición dz dado por z y las partículas X, y covarianza Q, donde Q es un parámetro del modelo. Este cálculo se realiza en el segmento de código escrito en (4.13)

Código 4.13 : Cómputo de pesos para PF (Diseño de Gavin Gao)

```
    dx = x[0, 0] - z[i, 1]
    dy = x[1, 0] - z[i, 2]
    prez = math.sqrt(dx ** 2 + dy ** 2)
    dz = prez - z[i, 0]
    w = w * gauss_likelihood(dz, math.sqrt(Q[0, 0]))
```

Los pesos calculados se normalizan al igual que en MATLAB. En este momento, el algoritmo procede a calcular el estado $xEst$ antes de realizar resampling, con el producto punto entre las partículas y los pesos. Esto sugiere que el proceso puede ser realizado antes o después de calcular el estado, sin embargo debe ser consistente durante toda la ejecución del algoritmo.

El resampling de las partículas se realiza con la técnica estratificada, la cual es similar al resampling sistemático, con la excepción que los marcadores $u^{(n)}$ se calculan para $n = 0, \ldots, M-1$, donde $M = NP$, aleatoriamente con distribución uniforme en el intervalo $\left(\frac{n-1}{M}, \frac{n}{M}\right]$, según la expresión (4.32).

$$u^{(n)} \sim U\left(\frac{n-1}{M}, \frac{n}{M}\right], \qquad n = 1, \ldots, M-1 \tag{4.32}$$

Al término del proceso, se almacenan las nuevas partículas en el arreglo px y se vuelven a inicializar los pesos normalizados, según se muestra en (4.14). Se trata de un paso que este diseño realiza dentro de la función asignada para el resampling, mientras que el ejemplo de implementación en MATLAB explicado en la sección anterior lo realiza fuera de esta función.

Código 4.14 : Resampling de partículas para PF (Diseño de Gavin Gao)

```
    Neff = 1.0 / (pw.dot(pw.T))[0, 0]  # Effective particle number
    if Neff < NTh:
        wcum = np.cumsum(pw)
        base = np.cumsum(pw * 0.0 + 1 / NP) - 1 / NP
        resampleid = base + np.random.rand(base.shape[0]) / NP
        inds = []
        ind = 0
        for ip in range(NP):
            while resampleid[ip] > wcum[ind]:
                ind += 1
            inds.append(ind)

        px = px[:, inds]
        pw = np.zeros((1, NP)) + 1.0 / NP  # init weight
```

Estas dos funciones son suficientes para comprender este ejemplo del filtro de partículas en Python. Lo siguiente es identificar las modificaciones que se deben realizar para operar adecuadamente con datos cuantizados. Los objetivos del proyecto no requieren que el algoritmo realice observaciones en tiempo real, por lo que este procedimiento puede ser omitido y reemplazado por llamar a los elementos de un arreglo y_t modelado previamente al inicio de la secuencia. Además, en la función *pf_localization(px, pw, z, u)* se debe incluir la fórmula (4.28) para calcular los pesos según los límites a_t y b_t, dados por los umbrales del intervalo $\mathcal{J}_i$. El bucle *for* implementado para cada partícula puede ser omitido, pues los NP pesos pueden ser calculados en un comando aplicando la función *multivariate_normal(w, mean, cov)* de la biblioteca *scipy* de Python. Se entra en más detalle sobre estas modificaciones en la sección de trabajo experimental del documento.

El código completo del diseño de Gavin Gao para localización mediante filtro de partículas se encuentra en el anexo 9.53.

5. Filtro Suma de Gaussianas (GSF)

5.1. Antecedentes Generales

El filtro de Kalman permite estimar el estado de forma simple y eficiente, pero esto se logra estrictamente cuando el modelo cumple con dos características: linealidad y PDF con distribución gaussiana. Dejando de lado el caso con datos no lineales cuantizados, las perturbaciones presentes en la mayoría de ejemplos prácticos son no gaussianas, por lo que el filtro de Kalman no es generalmente aplicable para obtener estimaciones con un grado adecuado de exactitud. El filtro de partículas remedia esta situación, ofreciendo un método de estimación que no está sujeto a condiciones estrictas de aplicación. Sin embargo, al ser un método que se basa en cálculo de probabilidades, trabajar con procesos de alto orden y complejidad, manteniendo un alto nivel de exactitud, requiere la generación de un número elevado de partículas y un uso extenuante de tiempo de procesamiento y recursos computacionales como memoria. Los objetivos de este proyecto no es exclusivamente la implementación de estos algoritmos, sino que, además, la evaluación de estos mismos, pues la motivación original es determinar su viabilidad en un entorno de programación que le otorgue utilidad situaciones prácticas. El problema planteado es buscar la forma de obtener la PDF de filtraje eficientemente cuando el modelo del espacio de estados posee una distribución con la forma de suma de gaussianas debido al efecto de sus perturbaciones. La solución que se propone es el filtro Suma de Gaussianas para filtraje y suavizado [3]. Este documento explica en detalle el proceso de filtraje con este método, dejando el suavizado como información complementaria.

El método de estimación de estados con filtro Suma de Gaussianas tiene sus inicios en la necesidad de estimar las variables de estado de un sistema dinámico lineal, cuando este se encuentra sometido a ruidos de estado y ruidos de medición que se pueden escribir como la suma de múltiples distribuciones gaussianas, y que conllevan a la PDF del estado a adquirir la característica de una suma de gaussianas. Una de las primeras publicaciones que utiliza este nombre proviene del diario Automatica de la Federación Internacional de Control Automático (IFAC) en el año 1971 [32]. Los autores Harold W. Sorenson y Daniel L. Alspach escriben en su artículo sobre un método para realizar estimaciones recursivas según el modelo bayesiano usando aproximaciones con sumas de gaussianas, con el propósito de determinar una PDF *a*

posteriori y covarianza mínima para sistemas lineales con ruido no gaussiano de una manera relativamente directa. Desde entonces se han realizado numerosos avances para mejorar el rendimiento de este método y adaptarlo para diferentes aplicaciones y condiciones de trabajo.

Entre las aplicaciones para extender el uso de este método más allá del trabajo con sistemas lineales, se pueden resaltar dos: la estimación de estados con medición de datos cuantizados [3], particularmente datos binarios [4], y su implementación en sistemas, cuyas entradas y salidas dependen de funciones no lineales, tal como lo es ejemplo del modelo Hammerstein-Wiener [2]. En la primera y segunda referencia se desarrolla un algoritmo de doble filtrado para condiciones de salida no integrable cuantizada, aproximando la PMF condicional de la salida usando una regla de cuadratura de Gauss y entregando una estructura de suma de gaussianas. En la tercera, el algoritmo estima el modelo no lineal Hammerstein-Wiener con base en una aproximación de la función de probabilidad de la salida dada el estado usando la regla de cuadratura Gauss-Legendre, entregando también una suma de gaussianas. Estos dos ejemplos sirven para entender la expansión de este modelo para ajustarse a la problemática del proyecto, que es la operación del filtro cuando la medición del sistema es de datos cuantizados, configuración de datos que es no lineal y no integrable.

5.2. Modelo Matemático

El filtro Suma de Gaussianas [3] combina distintas estructuras gaussianas para estimación de estados, que trabajan en tándem para determinar la PDF del sistema. El primer concepto que se utiliza es el de modelo de mezclas gaussianas (*Gaussian Mixture Model* o GMM). Un GMM se define como un modelo probabilístico donde se asume que una distribución aleatoria de datos sin una forma característica es una combinación lineal de un número finito de distribuciones gaussianas multivariables y ponderadas, resultando en la superposición de campanas de Gauss con representación gráfica como la presente en la figura 5.6. El ejemplo muestra un GMM básico donde el modelo está formado por la suma de tres PDFs gaussianas con diferente valor medio μ y ponderación (o densidad) idéntica.

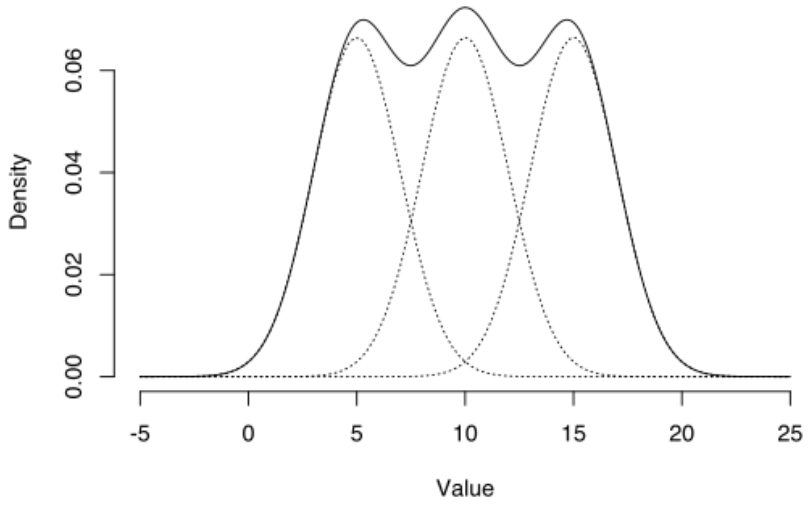


Figura 5.6: Representación gráfica de un GMM de tres componentes.

Fuente: Luis Benites, 2022, statologos.com

Al utilizar GMM se puede resolver el problema de aproximación de la PDF dada por el ruido cuando este es no gaussiano, mediante una representación matemática de la densidad con la forma expuesta en (5.33). En esta expresión, ζ es una variable aleatoria cualquiera que representa una combinación de PDFs gaussianas, ϕ_i es el i-ésimo peso de la mezcla, λ_i es la i-ésima media de la distribución gaussiana y Γ_i es la i-ésima matriz de covarianza. Los valores de los κ pesos ϕ se asignan tal que cumplan con la condición ($\phi_i > 0$, $\sum_{i=1}^{\kappa} \phi_i = 1$). El filtro Suma de Gaussianas busca entregar una PDF del estado representada por $p\left(x_t|y_{1:t}\right)$ con forma de GMM.

$$p(\zeta) \quad = \quad \sum_{i=1}^{\kappa} \phi_i \mathcal{N}\left(\zeta; \lambda_i, \Gamma_i\right) \tag{5.33}$$

Las ecuaciones para el proceso de filtraje con esta estructura son conocidas y están dadas por (5.34) y (5.35). Ahora, $p\left(x_t|y_{1:t}\right)$ y $p\left(x_{t+1}|y_{1:t}\right)$ representan las PDFs de la etapa de corrección y predicción del algoritmo, respectivamente.

$$p\left(x_t|y_{1:t}\right) = \frac{p\left(y_t|x_t\right)p\left(x_t|y_{1:t-1}\right)}{p\left(y_t|y_{1:t-1}\right)} \tag{5.34}$$

$$p\left(x_{t+1}|y_{1:t}\right) = \int p\left(x_{t+1}|x_t\right)p\left(x_t|y_{1:t}\right)dx_t \tag{5.35}$$

Al conocer la función de probabilidad de las mediciones $p\left(y_t|x_t\right)$ y el modelo de la PDF de transición de estados $p\left(x_{t+1}|x_t\right)$, entonces las ecuaciones de filtraje se pueden resolver con relativa facilidad. Sin embargo, el efecto de la cuantización de la salida y su distribución no gaussiana hace muy difícil encontrar la función de probabilidad $p\left(y_t|x_t\right)$. El filtro de partículas es una solución ante este problema, pero no cubre la necesidad de reducir la complejidad computacional para modelos de orden alto.

Recordando la definición de la PDF, la PMF resulta ser el modelo probabilístico más apropiado para trabajar con conjuntos de datos cuantizados, debido a la naturaleza discreta de estos datos. Definiendo una PMF $p\left(y_t|x_t\right)$ con forma equivalente a (4.27) dada por (5.36), y límites de integración a_t y b_t definidos por la tabla 4.1, se puede utilizar la regla de cuadratura Gauss-Legendre [6] para aproximar un GMM con la forma (5.37). La constante K es definida por el usuario e indica la cantidad de puntos de la regla de cuadratura Gauss–Legendre necesarias para minimizar el error entre la predicción y el modelo, R es la matriz de covarianza de la salida y $\{\zeta_t^\tau, \eta_t^\tau, \mu_t^\tau\}$ es un conjunto asociado a los intervalos $\mathcal{J}_i$ que separa los niveles de un ILQ ubicado en la salida con la forma descrita en (2.11). El conjunto se define en la tabla 5.2, donde ω_τ y ψ_τ son pesos y puntos obtenidos por la regla de cuadratura [3] [6].

$$p\left(y_t|x_t\right) = \int_{a_t}^{b_t} \mathcal{N}\left(v_t; 0, R\right)dv_t \tag{5.36}$$

$$p\left(y_t|x_t\right) \approx \sum_{\tau=1}^{K} \zeta_t^\tau \mathcal{N}\left(\eta_t^\tau; Cx_t + Du_t + \mu_t^\tau, R\right) \tag{5.37}$$

	$y_t = \nu_1$	$y_t = \nu_i$ **FLQ:** $i = 2, \ldots, L-1$ **ILQ:** $i = \ldots, 1, \ldots, L, \ldots$	$y_t = \nu_L$
ζ_t^τ	$2\omega_\tau/(1+\psi_\tau)^2$	$\omega_\tau (q_i - q_{i-1})/2$	$2\omega_\tau/(1+\psi_\tau)^2$
η_t^τ	$-(1-\psi_\tau)/(1+\psi_\tau)$	$\psi_\tau (q_i - q_{i-1})/2$	$(1-\psi_\tau)/(1+\psi_\tau)$
μ_t^τ	$-q_1$	$-(q_i + q_{i-1})/2$	$-q_{L-1}$

Tabla 5.2: Tabla de variables para la aproximación GMM de $p(y_t|x_t)$ según regla de cuadratura Gauss-Legendre.

La estructura de filtraje bayesiana general entrega una PDF de la corrección del estado en (5.34) equivalente a la multiplicación de dos GMM, y $p(y_t|y_{1:t-1})$ es una constante de normalización. El resultado es $p(x_t|y_{1:t})$ con forma GMM de más componentes que los factores $p(y_t|x_t)$ y $p(x_t|y_{t-1})$. Esto conlleva a que la PDF de la predicción del estado $p(x_{t+1}|y_{1:t})$ en (5.35) también es una distribución GMM. El algoritmo para filtraje con sumas de gaussianas se deriva de esta afirmación [3].

La secuencia comienza con la inicialización en $t = 1$, donde $p(x_1) \sim \mathcal{N}(x_1; \mu_1, P_1)$ es la distribución del estado inicial con media μ_1 y covarianza P_1, previa a la primera iteración de la etapa de corrección y predicción. Las PDFs de interés son $p(x_t|y_{1:t})$ para la corrección del estado actual x_t, condicionado por una medición actual y previa de la salida aproximada, y $p(x_{t+1}|y_{1:t})$ para la predicción del estado futuro x_{t+1}. Las ecuaciones (5.38) y (5.39) se utilizan para el cálculo de ambas PDFs en los instantes $t = 1, \ldots, N$. En la corrección, se tiene una suma de $M_{t|t}$ componentes donde $M_{t|t} = KM_{t|t-1}$, K es la misma constante que en (5.37) y $M_{t|t-1}$ es el número de componentes gaussianas utilizadas para la predicción en el instante $t-1$. En la predicción, $M_{t+1|t} = M_{t|t}$. Con estas expresiones queda definido de manera general las bases para encontrar la solución de la estimación del estado.

$$p(x_t|y_{1:t}) = \sum_{k=1}^{M_{t|t}} \gamma_{t|t}^k \mathcal{N}\left(x_t; \hat{x}_{t|t}^k, \Sigma_{t|t}^k\right) \tag{5.38}$$

$$p(x_{t+1}|y_{1:t}) = \sum_{k=1}^{M_{t+1|t}} \gamma_{t+1|t}^k \mathcal{N}\left(x_{t+1}; \hat{x}_{t+1|t}^k, \Sigma_{t+1|t}^k\right) \tag{5.39}$$

En cada iteración de la ecuación (5.38) se deben computar $M_{t|t}$ veces las matrices de pesos $\gamma_{t|t}^k$, medias $\hat{x}_{t|t}^k$ y covarianzas $\Sigma_{t|t}^k$ del paso de corrección. El índice superior k lo define una 2-tupla (τ, l), tal que $k = (l-1)\,K + \tau$, donde $\tau = 1, \ldots, K$ y $l = 1, \ldots, M_{t|t-1}$. Las fórmulas para calcular cada matriz se escriben en (5.40), (5.41) y (5.42). En las expresiones asociadas, I es una matriz identidad, C, D y u_t se definen en el espacio de estados en (2.6).

$$\gamma_{t|t}^k = \frac{\overline{\gamma}_{t|t}^k}{\sum_{s=1}^{M_{t|t}} \overline{\gamma}_{t|t}^s} \tag{5.40}$$

$$\hat{x}_{t|t}^k = \hat{x}_{t|t-1}^l + K_t^l \left(\eta_t^\tau - \kappa_t^{l\tau}\right) \tag{5.41}$$

$$\Sigma_{t|t}^k = \left(I - K_t^l C\right) \Sigma_{t|t-1}^l \tag{5.42}$$

La Tabla 5.2 contiene las definiciones de los coeficientes ζ_t^τ, η_t^τ y μ_t^τ. Las fórmulas (5.43) y (5.44) describen las variables utilizadas en (5.40), (5.41) y (5.42).

$$\overline{\gamma}_{t|t}^k = \zeta_t^\tau \gamma_{t|t-1}^l \mathcal{N}_{\eta_t^\tau}\left(\kappa_t^{l\tau}, V_t^l\right) \tag{5.43}$$

$$K_t^l = \Sigma_{t|t-1}^l C^T \left(V_t^l\right)^{-1} \tag{5.44}$$

$$\kappa_t^{l\tau} = C\hat{x}_{t|t-1}^l + Du_t + \mu_t^\tau \tag{5.45}$$

$$V_t^l = R + C\Sigma_{t|t-1}^l C^T \tag{5.46}$$

El procedimiento de la ecuación (5.39) es similar, los pesos $\gamma^k_{t+1|t}$, medias $\hat{x}^k_{t+1|t}$ y covarianzas $\Sigma^k_{t+1|t}$ del paso de predicción se deben calcular con las fórmulas (5.47), (5.48) y (5.49) en base a los resultados obtenidos en (5.40), (5.41) y (5.42). las matrices A, B y Q, y la entrada u_t son las mismas en (2.5).

$$\gamma^k_{t+1|t} = \gamma^k_{t|t} \tag{5.47}$$

$$\hat{x}^k_{t+1|t} = A\hat{x}^k_{t|t} + Bu_t \tag{5.48}$$

$$\Sigma^k_{t+1|t} = Q + A\Sigma^k_{t|t}A^T \tag{5.49}$$

Durante la ejecución del filtro Suma de Gaussianas se computan las PDFs del estado x_t en cada instante t, donde $t = 1, \ldots, N$. En (5.38) se nota que se en cada instante se repite la etapa de corrección y se calcula una sumatoria de $M_{t|t} = KM_{t|t-t}$ componentes. Como K es una constante definida por el usuario antes de comenzar la ejecución y $M_{t|t-1}$ proviene del instante anterior, la cantidad de información que se debe almacenar y administrar crece exponencialmente con cada iteración y, después de unas cuantas, se produce un consumo excesivo de recursos computacionales. El diseño del filtraje y suavizado propone utilizar un método para reducir el número de componentes en cada iteración de estos algoritmos, de tal forma que un GMM $\{\phi_i, \nu_i, \Gamma_i\}_{i=1}^J$ se pueda transformar en un GMM $\{\phi_i, \nu_i, \Gamma_i\}_{i=1}^S$, donde $1 \leq S \leq J$. Estos métodos se conocen como reducción de Gauss o reducción de suma de gaussianas (*Gaussian Sum Reduction*) y existen abundantes maneras de llevar a cabo este procedimiento [1]. Este proyecto utiliza el método Kullback–Leibler para realizar la reducción [28], inspirado por diseño de Kitawaga [26]. Se emplea el algoritmo (5.50) para combinar dos componentes que poseen disimilitud mínima en un solo componente que mantenga el momento de primer y segundo orden de ambos. La función $\mathcal{M}(.,.)$ es descrita por las ecuaciones (5.51), (5.52) y (5.53).

$$(\phi_{ij}, \nu_{ij}, \Gamma_{ij}) = \mathcal{M}\left\{(\phi_i, \nu_i, \Gamma_i)\,(\phi_j, \nu_j, \Gamma_j)\right\} \tag{5.50}$$

$$\phi_{ij} = \phi_i + \phi_j \tag{5.51}$$

$$\nu_{ij} = \overline{\phi}_i \nu_i + \overline{\phi}_j \nu_j \tag{5.52}$$

$$\Gamma_{ij} = \overline{\phi}_i \Gamma_i + \overline{\phi}_j \Gamma_j + \overline{\phi}_i \overline{\phi}_j \,(\nu_i - \nu_j)\,(\nu_i - \nu_j)^T \tag{5.53}$$

$$\overline{\phi}_i = \frac{\phi_i}{\phi_i + \phi_j} \tag{5.54}$$

$$\overline{\phi}_j = \frac{\phi_j}{\phi_i + \phi_j} \tag{5.55}$$

Medir la disimilitud de dos componentes i y j, donde $i \neq j$, se realiza con la expresión (5.56), que cumple con $\mathcal{D}(i,j) = \mathcal{D}(j,i)$ y $\mathcal{D}(i,i) = 0$. Entonces, al realizar la combinación de todo par de componentes donde su medida de disimilitud sea mínima, se obtiene una reducción en la cantidad de componentes en la operación de J a $0.5J(J-1)$. Esto resulta en la omisión de componentes redundantes en el modelo y en un aumento de la eficiencia del algoritmo, sin pérdidas significativas de información. Al implementar cada una de las operaciones expuestas en esta sección, se puede realizar el proceso de filtraje y obtener la estimación del estado por medio del filtro Suma de Gaussianas.

$$\mathcal{D}(i,j) = \frac{1}{2}\left[\phi_{ij} \log\left(\det\{\Gamma_{ij}\}\right) - \phi_i \log\left(\det\{\Gamma_i\}\right) - \phi_j \log\left(\det\{\Gamma_j\}\right)\right] \tag{5.56}$$

El diseño estudiado también explica el procedimiento par solucionar el problema de suavizado con aproximación suma de gaussianas. La diferencia principal que tiene este procedimiento con el filtraje está en que el suavizado cuenta con las mediciones que se realizan en todo instante t, donde $t = 1, \ldots, N$, entonces toda iteración utiliza todas las muestras, mientras que el filtraje es causal y utiliza las mediciones a medida que avanza el tiempo. Igual que con el proceso de filtraje, el suavizado empieza con la fórmula para la estructura bayesiana en (5.57).

$$p\left(x_t|y_{1:N}\right) = p\left(x_t|y_{1:t}\right) \int \frac{p\left(x_{t+1}|y_{1:N}\right) p\left(x_{t+1}|x_t\right)}{p\left(x_{t+1}|y_{1:t}\right)} dx_{t+1} \tag{5.57}$$

El problema que surge de esta estructura es el mismo, cuando la PDF de filtraje se escribe como una suma de gaussiana este modelo no es aplicable, pues existe una división para una PDF suma de gaussianas. Esto hace muy complejo el cálculo de la PDF deseada. Como solución se recurre nuevamente al diseño de Kitawaga [26], el cual provee una solución con base en dos filtros llamada fórmula Two-Filter. Como su nombre lo indica, este diseño implementa dos procedimientos, el suavizador y el filtro en reversa (*Backwards Filter*), para formar de manera recursiva una PDF que si se puede aproximar como un GMM. El filtro en reversa se describe por la recusión (5.58), donde (5.58) es la predicción del estado y (5.59) es la corrección. Notar que ahora se realiza la predicción antes de la corrección. Además, se puede observar claramente la distinción con el filtraje, pues las funciones $p\left(y_{t+1:N}|x_t\right)$ y $p\left(y_{t:N}|x_t\right)$ están condicionadas por el estado x_t, en lugar de la salida y_t.

$$p\left(y_{t+1:N}|x_t\right) = \int p\left(y_{t+1:N}|x_{t+1}\right) p\left(x_{t+1}|x_t\right) dx_{t+1} \tag{5.58}$$

$$p\left(y_{t:N}|x_t\right) = p\left(y_t|x_t\right) p\left(y_{t+1:N}|x_t\right) \tag{5.59}$$

Por otro lado, la PDF de suavizado se puede describir con la expresión (5.60). Aquí, $p\left(x_t|y_{1:t-1}\right)$ se obtiene de (5.39), $p\left(y_{t:N}|y_{1:t-1}\right)$ es una constante de normalización y $p\left(y_{t:N}|x_t\right)$ se obtiene de (5.59).

$$p\left(x_t|y_{1:N}\right) = \frac{p\left(x_t|y_{1:t-1}\right)p\left(y_{t:N}|x_t\right)}{p\left(y_{t:N}|y_{1:t-1}\right)} \tag{5.60}$$

Se llega a una aproximación de la PDF de suavizado en tiempo $t = N$ de manera proporcional, dada por (5.61). Recordando la explicación del proceso de filtraje, se pueden identificar similitudes entre esta expresión y (5.38). En este caso, $S_{t|N}$ se asocia a $M_{t|t}$ según el procedimiento de reducción gaussiana, y $\overline{\epsilon}^{\tau}_{t|N}$ se calculan con $\gamma^{\tau}_{t|t}$. Es importante notar las similitudes entre la solución de ambos procesos para facilitar la implementación del filtro en reversa y suavizado, bajo el pretexto que se desee expandir los objetivos del proyecto. Más información de sobre estos procesos y la definición de sus variables se pueden encontrar en las referencias [3] y [26].

$$p\left(x_t|y_{t:N}\right) \propto \sum_{\tau=1}^{S_{t|N}} \overline{\epsilon}^{\tau}_{t|N}\mathcal{N}_{x_t}\left(\hat{x}^{\tau}_{t|N}, \Sigma^{\tau}_{t|N}\right) \tag{5.61}$$

5.3. Implementación de GSF

El algoritmo 3 muestra el pseudo-código que representa los pasos del filtro Suma de Gaussianas para salida cuantizada. Su estructura proviene de la referencia [3].

Algorithm 3 Filtro de Suma de Gaussianas

Input: PDF del estado inicial $p(x_1)$, modelo GMM donde $M_{1|0} = 1$, $\gamma_{1|0} = 1$, $\hat{x}_{1|0} = \mu_1$ y $\Sigma_{1|0} = P_1$. Puntos de cuadratura Gauss-Legendre $\left\{\omega_\tau, \psi|tau\right\}_{\tau=1}^{K}$.
for $t = 1 \rightarrow N$ **do**
    Calcular y almacenar ζ_t^τ, η_t^τ y μ_t^τ según la tabla 5.2.
    Corrección:
    Declarar $M_{t|t} = KM_{t|t-1}$.
    for $l = 1 \rightarrow M_{t|t-1}$ **do**
        for $\tau = 1 \rightarrow K$ **do**
            Computar índice $k = (l-1)K + \tau$.
            Computar y almacenar $\gamma_{t|t}^k$, $\hat{x}_{t|t}^k$ y $\Sigma_{t|t}^k$, según (5.40), (5.41) y (5.42).
        end for
    end for
    Predicción:
    Declarar $M_{t+1|t} = M_{t|t}$.
    for $k = 1 \rightarrow M_{t+1|t}$ **do**
        Computar y almacenar $\gamma_{t+1|t}^k$, $\hat{x}_{t+1|t}^k$ y $\Sigma_{t+1|t}^k$, según (5.47), (5.48) y (5.49).
    end for
    Ejecutar algoritmo Kullback–Leibler de reducción Gaussiana para obtener el GMM reducido de $p\left(x_t|y_{1:t}\right)$, dado por las expresiones (5.50) a (5.56).
end for
Output: Las PDFs de filtraje $p\left(x_t|y_{1:t}\right)$, las predicciones $p\left(x_{t+1}|y_{1:t}\right)$, y el conjunto $\{\zeta_t^\tau, \eta_t^\tau, \mu_t^\tau\}$, con $t = 1, \ldots, N$.

5.3.1. Ejemplo de Implementación de GSF utilizando MATLAB

La implementación del filtro Suma de Gaussiana en MATLAB cumple el rol de referencia para su implementación en Python. Al ser un método relativamente reciente en comparación con otros métodos clásicos, su concepto se encuentra mayoritariamente contenido en la literatura y no es una tarea sencilla encontrar un ejemplo de su implementación en Python libremente publicado con la cual se pueda comparar su rendimiento. Es por esta razón que es importante comprender las estructuras utilizadas en algoritmo de MATLAB, pues su migración a Python es el único medio disponible con el cual se puede obtener una implementación apropiada.

El diseño que se analiza en esta sección proviene del trabajo experimental asociado al documento "A Two-Filter Approach for State Estimation Utilizing Quantized Output Data" [3], donde se desarrolla un enfoque de doble filtraje para estimar el estado de un sistema dinámico con salida cuantizada. Este ejemplo en particular lo proporciona Angel L. Cedeño, como apoyo para el trabajo experimental de este proyecto. Los archivos contienen un código básico para ejecutar exclusivamente el proceso de filtraje utilizando filtro Suma de Gaussianas.

El código comienza con la inicialización del arreglo de estados *xtmt* y covarianzas *Stmt*, y con la estructura *TU* para almacenar parámetros iniciales de la PDF de predicción en el instante $t = 1$. En esta parte se identifica la primera función auxiliar del programa, *[x,P]=gmmTomuP(Data)*. Su propósito es recibir un conjunto de pesos w, medias μ y varianzas Σ contenida en la estructura *Data*, y entregar un estado x y covarianza P modelado según un GMM con esos parámetros, con el segmento de código que se observa en (5.15). Si los conjuntos son de largo $tam = 1$, la función entrega $x = \mu$ y $P = \Sigma$. De no cumplirse la condición, la función itera sobre los pesos, calculando un estado y covarianza con fórmulas matriciales asociadas a la distribución gaussiana escrita en (5.33).

Código 5.15 : Cómputo de estado y convarianza inicial con GMM (MATLAB)

```
    for i=1:1:tam
        x=x+alphai{i}*mui{i};
    end
    for i=1:1:tam
        P=P+alphai{i}*(Pi{i}+(mui{i}-x)*(mui{i}-x)');
    end
```

Otra función en la inicialización es *[yx] = model_pytxt(G)*, que estima la PMF de la salida cuantizada condicionada por el estado. Se utiliza la regla de cuadratura Gauss-Legendre de K puntos para aproximar $p(y_t|x_t)$, expresada por arreglos que contienen el conjunto $\{\zeta_t^\tau, \eta_t^\tau, \mu_t^\tau\}$. Los N elementos de cada arreglo se calculan iterativamente según la tabla 5.2, como se ve en (5.16). Esta implementación utiliza una función $Q = gauss\_legendre(n,varargin)$ para obtener los n pesos y puntos según la Gauss-Legendre, sin embargo, como este es un método estandarizado que entrega siempre la misma salida al recibir la misma entrada, no es requerido analizarlo.

Código 5.16 : Cómputo de aproximación con regla Gauss-Legendre (MATLAB)

```
for t=1:1:G.sim.N
    at=G.sim.yt{t}-0.5*G.mm.Delta;
    bt=G.sim.yt{t}+0.5*G.mm.Delta;
    yx.muj{t}=-(at+bt)/2;
    yx.Xj{t}=((bt-at)/2)*xj;
    yx.betaj{t}=wj*(bt-at)/2;
end
```

El siguiente paso es la etapa de corrección del algoritmo. Se escribe un bucle *for* doble, el primero de 1 a Mt_tm1, y el segundo de 1 a K, donde Mt_tm1 indica la cantidad de componentes gaussianas para la estimación y K la cantidad de puntos de la regla de cuadratura Gauss-Legendre. Aquí se computan los pesos no normalizados $\overline{\gamma}_{t|t}^{k}$, las medias $\hat{x}_{t|t}^{k}$ y las covarianzas $\Sigma_{t|t}^{k}$ con las expresiones (5.41) a (5.46). Los valores calculados se almacenan en los arreglos ps, md y vz, respectivamente, según el código en (5.17). La variable vz es un vector de matrices con dimensión $n \times n \times Mtt$, y md es una matriz $n \times Mtt$ y ps es una matriz $1 \times Mtt$, donde n es la cantidad de filas que posee la matriz A del modelo del espacio de estados y $Mtt = K * Mt\_tm1$.

Código 5.17 : Definición de matrices para la corrección de estados (MATLAB)

```
vz=zeros(n,n,Mtt);
md=zeros(n,Mtt);
ps=zeros(1,Mtt);
```

Una vez completado el cómputo de las variables, *[sm,lse] = normalized_weights(x)* se encarga de calcular los pesos normalizados $\gamma_{t|t}^{k}$ con la expresión (5.40) según los valores almacenados en ps. La función de MATLAB *[xmax,k] = max(x)* se utiliza para encontrar el mayor peso almacenado en x y en índice donde se ubica. Luego, se itera sobre cada elemento de x con el código (5.18), mediante un bucle *for* donde se crean dos vectores. El primer vector es e, que contiene la exponencial de la diferencia entre cada elemento de x y $xmax$, y el segundo vector es s, que contiene la suma acumulada de e, excepto la cantidad en el índice k donde se

encuentra $xmax$. La función entrega el vector e, normalizado por $s + 1$.

Código 5.18 : Normalización de pesos mediante iteración (MATLAB)

```
    for i = 1:n
        e(i) = exp(x(i)-xmax);
        if i ~= k
            s = s + e(i);
        end
    end
```

La etapa de predicción es muy similar a la corrección. Ahora se utiliza un solo bucle *for*, de 1 a Mtt, para computar $\gamma^k_{t|t}$, $\hat{x}^k_{t|t}$ y $\Sigma^k_{t|t}$ según las expresiones (5.47), (5.48) y (5.49). Los arreglos donde se almacenan los resultados son S, m y w, y tienen las mismas dimensiones que vz, md y ps, respectivamente. Estos se definen con el segmento de código en (5.19).

Código 5.19 : Definición de matrices para la predicción de estados (MATLAB)

```
S=zeros(n,n,Mtt);
m=zeros(n,Mtt);
w=zeros(1,Mtt);
```

La última función compleja que debe analizarse es el algoritmo de reducción gaussiana por el método de Kullback–Leibler. Este proceso se implementa mediante la función auxiliar *[pso,med,var]=gaussian_reduction(LI,LS,LD,ps,md,vz)*, la cual se llama solo si se cumple la condición $Mtt \geq K$. Para mantener el orden del código, la implementación separa el procedimiento en tres sub-funciones, las cuales se combinan para reducir la cantidad total de componentes gaussianos almacenados en ps, md y vz, entre los límites inferior LI y superior LS definidos por el usuario. La sub-función *[wij,muij,Pij] = merged(psi,psj,mdi,mdj,vzi,vzj)* computa ϕ_{ij}, ν_{ij} y Γ_{ij} con las expresiones (5.51) a (5.55), con el código escrito en (5.20).

Código 5.20 : Cómputo de coeficientes $\phi_{ij}/\nu_{ij}/\Gamma_{ij}$ (MATLAB)

```
wij=psi+psj;
wiIij = (psi)/(psi+psj);
wjIij = (psj)/(psi+psj);
muij = wiIij*mdi+wjIij*mdj;
Pij = wiIij*vzi + wjIij*vzj + wiIij*wjIij*(mdi-mdj)*(mdi-mdj)';
```

Por otro lado, las sub-funciones $B = boundS(ps,md,vz,B)$ y $B = bound(ps,md,vz,iast,B)$ evalúan la disimilitud escrita en (5.56), bajo diferentes rangos dentro de los arreglos. Para esta tarea se utilizan operaciones equivalentes donde los argumentos son diferentes. Un ejemplo de la forma que toman estas operaciones en el código se puede observar en (5.21).

Código 5.21 : Ejemplo de código para calcular dismilitud (MATLAB)

```
B(i,j)=0.5*(wij*log(det(Pij))-ps(i)*log(det(vz(:,:,i)))-ps(j)*log(det(vz(:,:,j))));
```

El cuerpo principal de la función evalúa la disimilitud entre cada elemento de ps, md y vz en un bucle *while*, mientras se cumpla $k > LS$, donde k es un número entero que se inicializa con la cantidad de elementos en ps. Este proceso toma la forma del código (5.22). En cada iteración se extraen los índices (i, j) donde se ubican los elementos que poseen la menor disimilitud entre sí, sin ser el mismo, mediante una función de MATLAB *[iaste,jaste]=find(B==min(min(B)))*. Luego, se reemplazan los elementos en i de cada arreglo por ϕ_{ij}, ν_{ij} y Γ_{ij}, respectivamente, los elementos en j se eliminan, y se reduce k en uno. Cuando deja de cumplirse $k > LS$, implica que el GMM posee menos componentes gaussianos que el máximo deseado, resultado en un modelo con menor número de componentes redundantes y dando fin al algoritmo. La función retorna los arreglos reducidos pso, med y var.

Código 5.22 : Reducción de Gauss mediante cálculo de dismilitud (MATLAB)

```
[iaste,jaste]=find(B==min(min(B)));  iast=iaste(1); jast=jaste(1);
[w,mu,P] = merged(ps(iast),ps(jast),md(:,iast),md(:,jast),vz(:,:,iast),vz(:,:,jast));
ps(iast)=w;    md(:,iast)=mu;    vz(:,:,iast)=P;
ps(jast)=[];   md(:,jast)=[];    vz(:,:,jast)=[];
B(jast,:)=[];     B(:,jast)=[];
B = bound(ps,md,vz,iast,B);
k=k-1;
```

Los arreglos reducidos se almacenan en la estructura TU, para ser utilizados como retroalimentación para el bucle. Al resultado de la etapa de corrección se les da forma GMM con *[x,P]=gmmTomuP(Data)*, y se almacenan como un estado xtt y covarianza Ptt estimados. Se hace lo mismo con el resultado de la predicción, esta vez almacenados en $xtmt$ y $Ptmt$. Con esto concluye la explicación de la implementación en MATLAB del filtro Suma de Gaussianas.

Cada códigos utilizado en este ejemplo para la implementación del filtro Suma de Gaussianas en MATLAB se pueden encontrar en los anexos de este documento, entre los anexos 9.47 y 9.51.

6. Plan de Trabajo

6.1. Tareas a Realizar

Hasta el momento se ha realizado un estudio y compresión detallado del funcionamiento e implementación de cada algoritmo de filtraje de interés para este proyecto, y su aplicación para la resolución del problema de estimación de estados con datos cuantizados. El trabajo que se debe realizar con esta información es implementar cada algoritmo en el lenguaje de programación Python, evaluar su funcionamiento y comparar su rendimiento bajo condiciones normales de ejecución.

Se opta enfocar el trabajo de este proyecto en determinar tres cualidades de los algoritmos implementados. Primero, el vector de estados estimados con cada método debe tener error mínimo y cercano a cero en relación con el vector de estados del modelo en el tiempo, en casos donde la cuantización y ruido del modelo son lo suficientemente bajos para considerarlos como despreciables. Como ejemplo para estos casos, el modelo puede tener parámetros como paso de cuantización $\Delta_q \leq 0.01$, covarianza de ruido de estado $P_{w_t} \leq 0.01$ y covarianza de ruido de medición $P_{v_t} \leq 0.0025$, mientras que su estado posee un orden magnitud de 10^1 o mayor. Cumplir con esta condición es un requerimiento mínimo de funcionalidad para afirmar que la implementación de los algoritmos se encuentra bien encaminada. Luego, se debe comparar la implementación en Python publicada de los métodos clásicos (filtro de Kalman estándar, filtro de Kalman cuantizado y filtro de partículas) con su respectiva implementación basada en la reproducción del código de MATLAB, según criterios como exactitud de estimación y tiempo de procesamiento. El propósito de este paso es ampliar el marco de referencia con el cual se realiza la comparación con el ejemplo del filtro Suma de Gaussianas implementado en Python. Finalmente, se deben clasificar las elecciones según los mismos criterios y comparar los resultados con los experimentos de la referencia [5].

El trabajo experimental busca confirmar que el ejemplo implementado en Python para el filtro Suma de Gaussianas entrega resultados a lo menos comparable con el mismo diseño implementado en MATLAB según se lo describe en [3], y posteriormente observar como se compara con ejemplos de métodos clásicos en Python. En el caso de que se identifiquen pérdidas significativas en el rendimiento de los algoritmos en comparación con su implementación en

MATLAB, otra conclusión válida es determinar si estas pérdidas son nativas de Python y constantes entre los diferentes métodos, o si dependen del comportamiento particular de algún procedimiento aplicado, tal como un comando u operación que no está optimizado y puede ser reemplazado.

6.2. Metodología

El cumplimiento de los objetivos del proyecto requiere escribir cada algoritmo en forma de comandos de Python, comprobar su funcionamiento y medir sus propiedades. La metodología de trabajo consiste en aplicar técnicas y recursos disponibles para resolver estas problemáticas.

Se cuenta con los ejemplos de MATLAB descritos en la sección de definiciones de cada algoritmo estudiado, recibidos previo al comienzo del trabajo experimental. La etapa de implementación se basa en reproducir las estructuras y herramientas nativas de MATLAB presentes en estos ejemplos, reemplazándolos por estructuras equivalentes existentes en el lenguaje Python. Además, MATLAB permite trabajar en modo de depuración para registrar el cambio de sus variables internas. Esta herramienta permite determinar durante las pruebas si las estructuras implementadas en Python entregan la misma salida que en MATLAB al recibir la misma entrada.

Del lado de Python, se plantea utilizar diferentes bibliotecas de funciones disponibles en el repositorio *pip* para facilitar la tarea de escribir rutinas que no estén directamente relacionadas con el tema principal del proyecto, tal como operaciones matemáticas complejas o generación de conjuntos de datos aleatorios con alguna distribución específica. Las principales bibliotecas que se utilizan para la implementación de algoritmos centrados en manejo de datos numéricos, y específicamente en cálculo estadístico, son *numpy* y *scipy*. Numpy contiene una extensa selección de rutinas que permiten la declaración y cómputo rápido de arreglos multidimensionales. En particular se destacan operaciones matriciales como el producto y la transposición, las cuales son altamente complejas utilizando exclusivamente herramientas nativas de Python. Scipy es una biblioteca enfocada a las ciencias, experimentación y optimización, se considera una extensión de *numpy*, pues provee herramientas adicionales para el cómputo de arreglos y estructuras especializadas para datos. Este proyecto utiliza *scipy* por sus funciones enfocadas al cálculo probabilístico.

Por último, la etapa de evaluación requiere determinar propiedades de cada algoritmo asociadas a la ejecución del programa. Python posee herramientas especiales para recopilar y almacenar este tipo de información.

- Para medir el tiempo que se demora la ejecución de cada algoritmo se utiliza la función *time.time()*, que almacena en una variable flotante el instante donde se llama la función. Realizar un llamado inmediatamente antes y después de ejecutar el algoritmo, y después restar los valores obtenidos permite registrar el tiempo de ejecución.

- El error cuadrático medio entre el vector modelado Y y la estimación $\hat{Y}$, ambos de largo N, se puede calcular con la formula 6.62. Para no sacrificar el orden del código, se emplea la función *mean_squared_error(Y_true, Y_pred)* de la biblioteca *sklearn*, que realiza el cálculo apropiado entre el vector de datos real Y_{true} y el vector de estimaciones Y_{pred}. Una prueba rápida permite comprobar que la función y la fórmula implementada directamente entregan la misma salida al recibir la misma entrada.

$$MSE = \frac{1}{N}\sum_{i=1}^{N}\left(Y_i - \hat{Y}_i\right)^2 \tag{6.62}$$

- La memoria máxima utilizada por cada método se puede determinar con la biblioteca *tracemalloc*, encargada de perfilar la memoria asignada a un proceso específico. El rastreo de memoria empieza al llamar la función *tracemalloc.start()*, desde la cual se puede registrar los bytes utilizados actualmente y los bytes máximos desde el primer llamado mediante *tracemalloc.get_traced_memory()*. Para no superponer las mediciones entre los métodos es importante llamar a *tracemalloc.clear_traces()* después de cada ejecución para reiniciar el registro.

6.3. Resultados Esperados

En el documento "On Recursive State Estimation for Linear State-Space Models Having Quatizized Output Data" [5] se realiza un estudio del problema de estimación sobre espacios de estado dinámicos donde la salida medible es un conjunto de datos cuantizados. Se compara el rendimiento de la implementación en MATLAB de diferentes métodos de estimación, entre los cuales se identifican métodos clásicos, como versiones del filtro de Kalman y filtros de partículas basados en MCMC con diferentes formas de resampling, y la implementación experimental del filtro Suma de Gaussianas encontrado en la literatura. Los resultados de este estudio se pueden observar en la tabla 6.3, correspondiente a la clasificación de los métodos de filtraje y suavizado, según su exactitud de estimación (determinado con el error cuadrático medio) y tiempo requerido de ejecución. Más información sobre la notación utilizada en esta tabla y otras conclusiones rescatadas se pueden encontrar en el documento correspondiente.

Este proyecto busca que los resultados de la evaluación para los ejemplos de cada método de estimación implementado en Python sean a lo menos comparable con los resultados escritos en la tabla 6.3. El marco de referencia es angosto, pues se ha optado por analizar solo el proceso de filtraje de los métodos seleccionados, y, por lo tanto, la única comparación factible en el presente es con las mediciones del error de estimación en MATLAB. Un caso favorable es que se mantenga la clasificación del error de cada método con valores similares, tal que el filtro Suma de Gaussianas entregue el menor error de estimación, seguido del filtro de partículas con resampling sistemático y número de partículas en orden descendente, y finalizando con las versiones del filtro de Kalman. En el caso de que esto no se cumpla, habrá que deliberar sobre la causa detrás de las diferencias que los resultados puedan presentar, ya sea por las técnicas empleadas en la implementación en Python o por propiedades nativas del lenguaje de programación.

Posición	Exactitud de filtraje		Exactitud de suavizado		Tiempo de ejecución de suavizado	
	ECM	Algortimo	ECM	Algortimo	Tiempo de ejecución	Algortimo
1	0.6724	GSF	0.5207	GSS	0.0026	KS
2	0.6740	PF-RWM-SYS(1000)	0.5212	PS-RWM-SYS(1000)	0.0031	QKS
3	0.6744	PF-RWM-ML(1000)	0.5220	PS-RWM-ML(1000)	0.0111	UKS
4	0.6754	PF-RWM-SYS(500)	0.5231	PS-RWM-SYS(500)	0.1453	PS-RWM-SYS(100)
5	0.6765	PF-RWM-ML(500)	0.5247	PS-RWM-ML(500)	0.1644	PS-RWM-LS(100)
6	0.6880	PF-RWM-SYS(100)	0.5393	PS-RWM-MT(1000)	0.1718	PS-RWM-ML(100)
7	0.6948	PF-RWM-ML(100)	0.5415	PS-RWM-SYS(100)	0.2077	PS-MH-LS(100)
8	0.7588	PF-RWM-MT(1000)	0.5420	PS-RWM-MT(500)	0.2109	PS-MH-SYS(100)
9	0.7830	PF-RWM-MT(500)	0.5470	PS-RWM-ML(100)	0.2354	PS-MH-ML(100)
10	0.9590	PF-MH-SYS(1000)	0.5689	PS-RWM-MT(100)	0.3931	GSS
11	0.9593	PF-MH-MT(1000)	0.6708	PS-MH-SYS(1000)	0.3984	PS-RWM-MT(100)
12	0.9595	PF-MH-ML(1000)	0.6711	PS-MH-ML(1000)	0.4579	PS-MH-MT(100)
13	0.9608	PF-MH-SYS(500)	0.6737	PS-MH-SYS(500)	0.4676	EKS
14	0.9612	PF-MH-MT(500)	0.6746	PS-MH-ML(500)	0.6048	PS-RWM-SYS(500)
15	0.9612	PF-MH-ML(500)	0.6752	PS-MH-MT(1000)	0.6348	PS-RWM-LS(500)
16	0.9686	PF-MH-SYS(100)	0.6781	PS-MH-MT(500)	0.7469	PS-RWM-ML(500)
17	0.9697	PF-MH-ML(100)	0.6927	PS-MH-SYS(100)	0.9772	PS-MH-LS(500)
18	0.9715	PF-MH-MT(100)	0.6927	PS-MH-ML(100)	1.2054	PS-RWM-SYS(1000)
19	1.0138	KF	0.6974	PS-MH-MT(100)	1.2274	PS-RWM-LS(1000)
20	1.6731	PF-RWM-MT(100)	0.7469	PS-MH-LS(1000)	1.2709	PS-MH-SYS(500)
21	1.8616	QKF	0.7497	PS-MH-LS(500)	1.4192	PS-MH-ML(500)
22	5.0549	UKF	0.7667	PS-MH-LS(100)	1.5197	PS-RWM-ML(1000)
23	7.3381	PF-RWM-LS(1000)	0.9100	KS	1.8362	PS-RWM-MT(500)
24	7.3602	PF-RWM-LS(500)	0.9393	PS-RWM-LS(1000)	2.0277	PS-MH-LS(1000)
25	7.6912	PF-RWM-LS(100)	1.2900	PS-RWM-LS(500)	2.3945	PS-MH-MT(500)
26	8.3846	PF-MH-LS(1000)	1.6693	QKS	3.0254	PS-MH-SYS(1000)
27	8.4079	PF-MH-LS(500)	5.0545	UKS	3.3505	PS-MH-ML(1000)
28	8.6717	PF-MH-LS(100)	6.4904	PS-RWM-LS(100)	3.6364	PS-RWM-MT(1000)
29	47.7827	EKF	33.8842	EKS	5.0651	PS-MH-MT(1000)

Tabla 6.3: Comparación de implementación en MATLAB de algoritmos de filtraje y suavizado para estimación con datos cuantizados [5].

7. Trabajo Experimental

7.1. Modelado de Sistema con Salida Cuantizada

Para corroborar la funcionalidad de los métodos de estimación escogidos para la comparación de rendimiento es necesario generan un conjunto de datos de salida que contenga valores explícitamente cuantizados. Se utiliza Python para modelar los vectores de estado x_t y salida y_t del sistema que se busca estimar, tal que ambos sigan la forma en (2.5), (2.6) y (2.7).

Los vectores de entrada *ut*, ruido de estado *wt* y ruido de medición *vt* se generan de forma aleatoria mediante la función de numpy *random.normal(a, b, N)*, como se observa en (7.23). Esta función genera una matriz de números aleatorios con distribución gaussiana de media a, desviación estándar b, cuya dimensión está dada por N. Si N es un valor entero, se genera un vector de largo N, y si es una tupla $N = (n, m)$, se genera una matriz $n \times m$. A manera de obtener resultados manejables, se configura una semilla para la que la generación de números aleatorios sea la misma en cada ejecución del programa. Este paso se realiza con el comando *random.seed(0)* al inicio del código.

Código 7.23 : Definición de vectores de entrada y ruido (Python)

```
1    # Input of the System
2    ut = np.dot(np.sqrt(2), np.random.normal(0, 1, N))
3    # Model Noise
4    wt = np.dot(np.sqrt(Q), np.random.normal(0, 0.1, (A.shape[0],N)).reshape(A.shape
     [0],N));
5    # Measurements noise
6    vt = np.dot(np.sqrt(R), np.random.normal(0, 0.05, N).reshape(1,N));
```

El siguiente paso es declarar los parámetros o propiedades del sistema para realizar los cálculos necesarios sin tener que escribir el valor en cada instancia. Se opta por crear una clase para almacenar todos los parámetros. Las *clases* de Python son una herramienta para generar estructuras que organizan varios conjuntos de datos en su interior, pues permiten acceder a estos datos llamando únicamente al objeto declarado utilizando la clase. La clase *GaussSystem(object)*, escrita en (7.24), se declara con el valor N de muestras a estimar, las matrices

A, B, C y D, las covarianzas Q y R del ruido, y otros parámetros que se utilizan en el filtro de partículas y el filtro suma de gaussianas. Con esta clase se puede declarar un objeto G, con el cual se accede a cualquier parámetro X dentro de G durante el programa, llamándolo con $G.X$. El objeto es flexible, esto significa que puede adquirir más parámetros que los ingresados durante su declaración con una operación $G.X = X$, donde X es no explícito en la clase *GaussSystem*. Este procedimiento se lleva a cabo numerosas veces durante la definición de variables, para guardar información adicional como el paso de cuantización *Delta*, el estado inicial y la covarianza inicial, calculados según GMM con la función *gmmTomuP(Data)*. Esta función posee una estructura equivalente a la función en MATLAB del mismo nombre, con excepción de algunas particularidades de Python para operar sobre matrices, que se explican en próximas secciones.

Código 7.24 : Definición de clase para un modelo de estados gaussiano (Python)

```
class GaussSystem(object):
    def __init__(self, A = None, B = None, C = None, D = None, Q = None, R = None, N =
     None, Kv = None, minC = None, maxC = None, minG = None, maxG = None):

        if(A is None or C is None):
            raise ValueError("Set proper system dynamics.")

        self.n = A.shape[1]
        self.m = C.shape[1]

        self.A = A
        self.B = 0 if B is None else B
        self.C = C
        self.D = 0 if D is None else D
        self.Q = np.eye(self.n) if Q is None else Q
        self.R = np.eye(self.n) if R is None else R
        self.N = 100 if N is None else N
        self.Kv = 10 if Kv is None else Kv
        self.minC = 2 if minC is None else minC;
        self.maxC = 2 if maxC is None else maxC;
        self.minG = 1 if minG is None else minG;
        self.maxG = 1 if maxG is None else maxG;
```

Para calcular el conjunto de datos del estado xt y salida antes de la cuantización zt se utiliza el código escrito en (7.25). Basta con usar un bucle *for* para todo t, donde $t = 1, \ldots, N$, armando el espacio de estados del modelo en cada iteración. Para realizar la multiplicación matricial entre A, B, C y D, y los vectores de estado y entrada, se debe utilizar la función de numpy *dot(A,B)*, pues la operación producto nativa de python no soporta factores matriciales. La salida cuantizada yt con un paso de cuantización Δ_q se obtiene aplicando la fórmula (2.12) sobre todos los elementos de zt, donde el operador *around()* es equivalente a la función *round* de MATLAB.

Código 7.25 : Cómputo del vector de estados y salida del modelo (Python)

```
# Simulated data
for i in range(N):
    xt[i+1] = np.dot(A, xt[i]) + np.dot(B, ut[i]) + wt[0, i]
    zt[i] = np.dot(C, xt[i]) + np.dot(D, ut[i]) + vt[0, i]

yt = Delta*np.around(np.dot(zt,1/Delta))
```

Se comprueba el modelo mediante gráficos de la simulación de un sistema de primer orden para $t = 1, \ldots, 100$, donde se elige a manera de ejemplo un paso de cuantización $\Delta_q = 7$, matrices positivas de transición $A = [0.9]$, $B = [1.0]$, $C = [2.0]$ y $D = [0.5]$, y el ruido wt y vt es gaussiano con $Q = [1.0]$ y $R = [0.5]$. El modelado de los vectores de estado y salida del sistema se encuentran ilustrados en las figuras 7.7 y 7.8. Se le asigna el nombre *Model State* y *Model Output* a los vectores xt y zt respectivamente, y se identifica que el segundo es una combinación lineal del primero debido a las matrices unidimensionales positivas. Por otro lado, la salida yt recibe el nombre *Measurements*, y se puede identificar claramente un paso de cuantización con tamaño siete que sigue al vector zt.

El código completo para el modelado del sistema se puede encontrar en el anexo 9.54. Este código también contiene los comandos para ejecutar cada algoritmos de estimación, medir sus propiedades de ejecución y graficar los resultados.

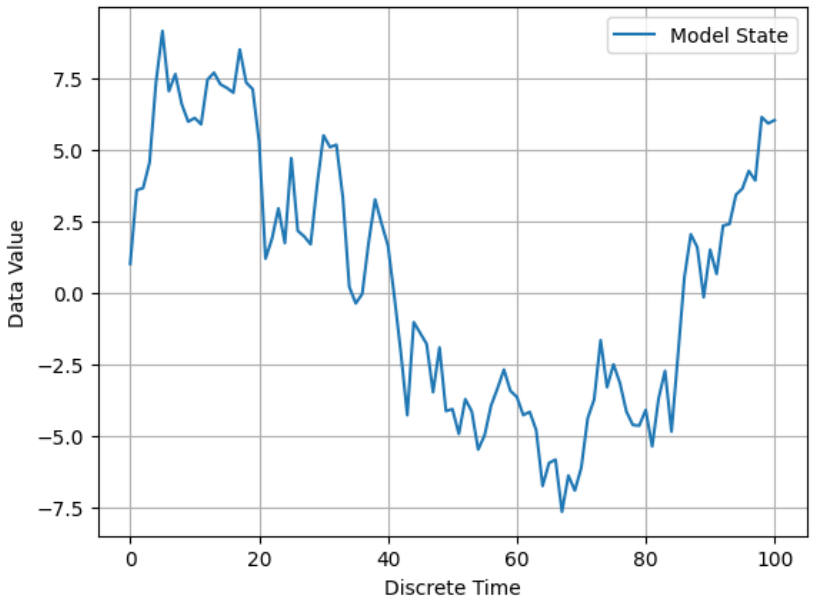


Figura 7.7: Gráfico del vector de estado xt del modelo en función del tiempo discreto.

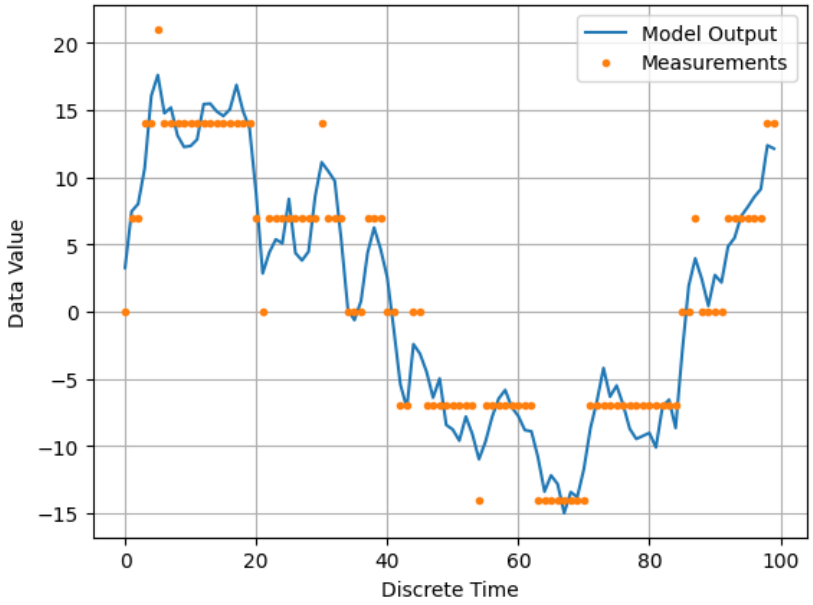


Figura 7.8: Gráfico del vector de salida real zt y salida cuantizada yt del modelo en función del tiempo discreto.

7.2. Implementación del Filtro de Kalman en Python

7.2.1. Implementación del Filtro de Kalman Estándar

El filtro de Kalman estándar posee la implementación más sencilla entre los métodos de interés. Estimar del estado requiere solamente realizar las operaciones para calcular la Ganancia de Kalman según (3.15), la etapa de *corrección* indicadas en (3.16) y (3.17), y la etapa de *predicción* indicada en (3.18) y (3.19), para todo instante t, donde $t = 1, \ldots, N$. Se puede realizar este procedimiento de forma directa en un bucle *for*; sin embargo, el uso de funciones y clases permite ordenar el código. Con esta intención, se decide escribir una función *KFEstimation(G, Xin, Pin)*, mediante el comando *def* de Python, y una clase *KalmanFilter(object)*, que contienen los pasos y la ejecución del algoritmo. Es importante mencionar que toda variable declarada dentro de una clase o función es interna a esta estructura y no puede ser compartida entre ellas, pero existen maneras de remediar esto.

La función *KFEstimation* recibe como argumentos un espacio de estados G definido por la clase *GaussSystem* y valores iniciales del estado *xin* y covarianza *Pin*. La función crea un objeto *KalmanFilter* y un vector de largo N para almacenar los valores estimados del estado, y ejecuta un bucle de N iteraciones de las etapas de corrección y predicción.

La clase *KalmanFilter* define propiedades internas de los estados y covarianzas estimadas en cada instante t, cuyo propósito es almacenarlas sin sobrescribir parámetros del modelo G. Define también dos funciones internas, *MeasurementUpdate(self, t)* para la etapa de corrección y *TimeUpdate(self, t)* para la predicción. El argumento *self* es una palabra reservada de Python que permite hacer referencia a propiedades de la clase sin necesidad de volver a declararlas, mientras que t es simplemente el instante actual. Primero, *MeasurementUpdate* calcula la Ganancia de Kalman con el código escrito en (7.26).

Código 7.26 : Cómputo de la Ganancia de Kalman (Python)

```
1       # Kalman Gain
2       S = self.G.R+np.dot(self.G.C, np.dot(self.Stmt, self.G.C.T))
3       self.K = np.dot(np.dot(self.Stmt,self.G.C.T), np.linalg.inv(S))
```

En (7.26), se separa el factor S de la operación para no utilizar expresiones muy largas que puedan reducir la legibilidad del código. La estructura $X.T$ permite obtener la traspuesta de la matriz X, y la función *linalg.inv(X)* es nativa de numpy y permite obtener la matriz inversa. Además, para utilizar un parámetro M del modelo G, estos se deben escribir de la forma $self.G.M$.

El siguiente paso es la corrección, donde se utilizan operaciones simples para determinar el valor de $\hat{x}_{t|t}$ y $\Sigma_{t|t}$. Estos son almacenados en las variables internas $self.xtt$ y $self.Stt$ de la clase. El código para realizar los cálculos de esta etapa están escritos en (7.27).

Código 7.27 : Etapa de corrección en KF (Python)

```
    # Measurement Update
    y = self.G.yt[t] - np.dot(self.G.C, self.xtmt) - np.dot(self.G.D,self.G.ut[t])
    self.xtt = self.xtmt + np.dot(self.K, y)
    self.Stt = np.dot(np.eye(self.G.m) - np.dot(self.K, self.G.C), self.Stmt)
```

La etapa de predicción se escribe de forma similar, como se puede ver en (7.28). Esta vez se calcula $\hat{x}_{t+1|t}$ y $\Sigma_{t|t}$, y se guardan en $self.xtmt$ y $self.Stmt$ respectivamente, para ser utilizados en la etapa de corrección del instante $t+1$. El estado almacenado en $self.xtt$ se retorna la función *KFEstimation* y se coloca en el vector de estados estimados.

Código 7.28 : Etapa de predicción en KF (Python)

```
    # Time Update
    self.xtmt =  np.dot(self.G.A, self.xtt) + np.dot(self.G.B, self.G.ut[t])
    self.Stmt = self.G.Q +  np.dot(np.dot(self.G.A, self.Stt), self.G.A.T)
    return self.xtt
```

Se simula una ejecución del filtro de Kalman estándar con un paso de cuantización bajo y ruidos de varianza reducida. Se configura un parámetro *Delta*, tal que $Delta = 0.01$, y se generan los vectores wt y vt ruido con desviación estándar de 0.1 para el ruido de estado y 0.05 para el ruido de medición. En la figura 7.9 se muestran los gráficos de estado del modelo y estimación del estado bajo la antedicha configuración. Se puede identificar un vector de estimaciones con error mínimo y cercano a cero en relación con el vector de estados del

modelo, particularmente en la imagen de la derecha, donde ambas líneas ocupan el mismo espacio. Se confirma que el algoritmo cumple con el requerimiento mínimo de funcionalidad para su implementación.

El código completo para la implementación en Python del filtro de Kalman estándar se puede encontrar en el anexo 9.55.

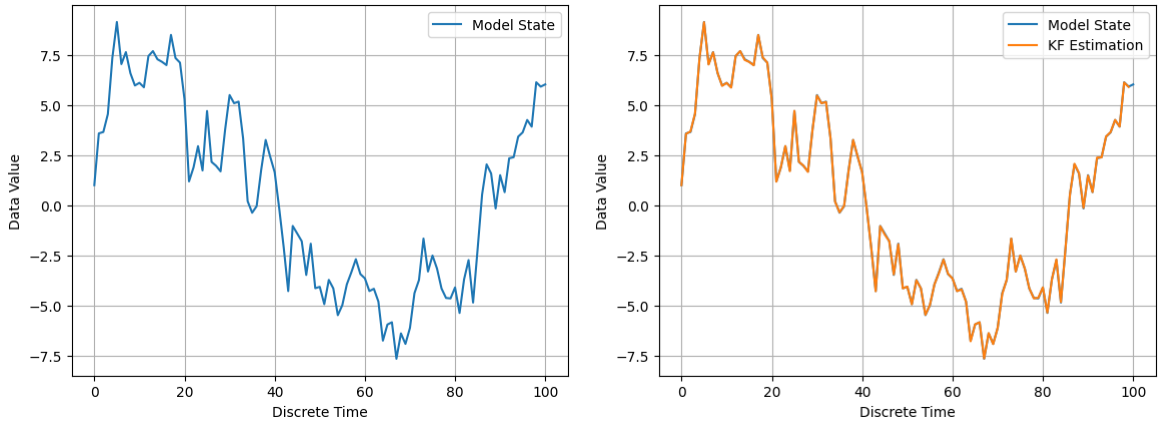


Figura 7.9: Comparación de gráficos de estado del modelo y estimación por filtro de Kalmam estándar con cuantización y ruido bajo.

7.2.2. Filtro de Kalman Cuantizado

La implementación del filtro de Kalman cuantizado es estructuralmente idéntica a la versión estándar. La única modificación requerida se puede observar en el código (7.29), que corresponde a la función *MeasurementUpdate* de la clase, ahora nombrada *QuantizedKalmanFilter*. La operación para calcular el estado corregido se realiza de acuerdo con la expresión (3.20) o (3.21). Se elige la forma 1 para este proyecto, tal que solo la estimación pasada y la entrada están sujetas a un operador de cuantización $\mathfrak{q}\{\cdot\}$ definido por (2.12) dentro del método, mientras que las mediciones, que ya están cuantizadas, no son alteradas.

Código 7.29 : Etapa de corrección en QKF (Python)

```
    # Measurement Update
    y = self.G.yt[t] - self.Quantization(np.dot(self.G.C, self.xtmt) + np.dot(self.G.D
    , self.G.ut[t]))
    self.xtt = self.xtmt + np.dot(self.K, y)
    self.Stt = np.dot(np.eye(self.G.m) - np.dot(self.K, self.G.C), self.Stmt)
```

Se implementa una función dentro de la clase *QuantizedKalmanFilter*, escrita en (7.30), la cual toma como argumento la referencia *self* y un valor *zt*. Su propósito es solamente aplicar la expresión (2.12) al argumento *zt* y retornar una salida *y* cuantizada con el paso de cuantización *Delta* almacenado en *G*. A esta función se le da el nombre *Quantization(self, zt)*.

Código 7.30 : Función para cuantizar datos según un paso *Delta* (Python)

```
    def Quantization(self, zt):
        y = self.G.Delta*np.around(zt/self.G.Delta)
        return y
```

Se realiza la comprobación bajo las mismas configuraciones que con el filtro de Kalman estándar, resultando en los gráficos de la figura 7.10. Nuevamente, el vector de estimaciones posee error mínimo y cercano a cero con el vector de estados del modelo.

El código completo para la implementación en Python del filtro de Kalman cuantizado se puede encontrar en el anexo 9.56.

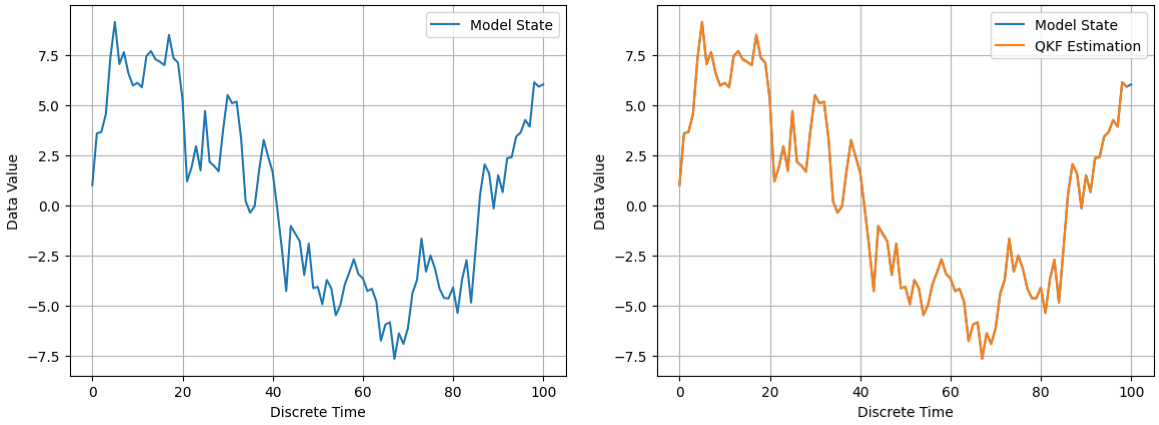


Figura 7.10: Comparación de gráficos de estado del modelo y estimación por filtro de Kalmam cuantizado con cuantización y ruido bajo.

7.3. Implementación del Filtro de Partículas en Python

Un filtro de partículas basado en MCMC posee una implementación breve y directa en términos del número de funciones, comandos y operaciones necesarias para realizar la estimación del estado, sin embargo, cada proceso involucrado implica un grado de complejidad mayor que las operaciones de los filtros de Kalman estándar y cuantizado. Esto se debe a la naturaleza estocástica de la generación y distribución de partículas a lo largo de las funciones no lineales del modelo. La biblioteca *scipy* es de gran ayuda para facilitar la implementación, pues contiene funciones similares a las presentes en MATLAB para realizar varios de los cálculos estadísticos apropiados.

La implementación comienza de forma similar a las anteriores, con una función para la estimación nombrada *PFEstimation(G, Npar, xin, Pin)* y una clase para el algoritmo nombrada *ParticleFilter(object)*. Ambas estructuras son tratadas igual que sus semejantes en el filtro de Kalman. *PFEstimation* recibe el modelo G, los valores iniciales del estado xin y covarianza Pin, y, además, un valor entero $Npar$, que indica la cantidad de partículas que se utilizan en la ejecución del algoritmo. La función instancia el objeto *ParticleFilter*, crea el vector de estimaciones y ejecuta un bucle para el cálculo del estado en $t = 1, \ldots, N$. Al finalizar, retorna el vector de estimaciones.

La clase *ParticleFilter* inicializa primero los parámetros internos del filtro, donde se incluyen vectores auxiliares para almacenar valores durante la ejecución. El paso más importante en esta parte es la inicialización del vector de partículas *xpar*, con el código que se observa en (7.31). El conjunto de partículas iniciales corresponde a una repetición del estado *xin* a lo largo de un vector de *Npar* elementos mediante la función, al cual se le suma un conjunto de *Npar* valores aleatorios con distribución gaussiana de media 0 y desviación estándar 1, ajustados por la factorización de Cholesky de la covarianza P_{in}. Las funciones utilizadas en este paso son *repmat(a,m,n)*, que crea una matriz $m \times n$ de repeticiones de a, y *chol(a)*, que realiza una factorización de Cholesky de a, tal que el resultado retornado es una matriz l, donde $a = l \cdot l'$.

Código 7.31 : Inicialización de partículas para PF (Python)

```
1    self.xpar = (repmat(self.xin, 1, self.Npar) + np.dot(chol(self.Pin), np.random.
      normal(0, 1, (self.G.n,self.Npar))))
```

Los otros parámetros que se inicializan durante la declaración de la clase son a y b, que definen los umbrales q_{i-1} y q_i del intervalo $\mathcal{J}_i$ asociado al nivel de cuantización del conjunto $\mathcal{V}$ donde se ubica cada medición de la salida y_t del modelo.

La siguiente definición en *ParticleFilter* es la función *ParticleUpdate(self, t)*, donde se realiza la estimación completa en el instante t. Teniendo un conjunto de partículas, el primer paso de la secuencia es calcular los pesos de cada elemento según la expresión (4.25), escrita en el código (7.32). Aquí, debido a que se está trabajando con mediciones cuantizadas, se requiere emplear la aproximación de $p\left(y_t|x_t^{(i)}\right)$ con datos cuantizados resumida por (4.27), donde los límites de integración a_t y b_t están dados por la tabla 4.1. La matriz de pesos w se calcula entre los límites correspondientes mediante una distribución normal multivariable dada por la función *multivariate_ normal.cdf(X, mu, Sigma)* de *spipy*, cuyos argumentos son la matriz X, la media *mu* y la covarianza *Sigma*. Los pesos resultantes son normalizados mediante la suma de sus componentes, comprobando que esta suma sea siempre distinta de cero.

Código 7.32 : Cómputo de pesos con PF (Python)

```
    w = (mvn.cdf(lim_b.T, mean = 0., cov = self.G.R) - mvn.cdf(lim_a.T, mean = 0., cov
     = self.G.R)).reshape(1,self.Npar)
    if np.sum(w) == 0:
        raise ValueError("Not proper CDF calculation.")
    w = np.dot(w, 1/np.sum(w))
```

El proceso de resampling se ejecuta de forma idéntica que en la implementación de MATLAB. Para mantener la organización de los códigos, se crea un archivo separado de la función y se le asigna el nombre *systematic_resample(w, par)*. Esta función toma los pesos w y partículas *par*, y opera para entregar un nuevo conjunto de partículas no sometido al fenómeno de degeneración. La única función matemática proveniente de una biblioteca y utilizada en este procedimiento es *cumsum(X)* de *numpy*, la cual es equivalente a la función de MATLAB con el mismo nombre. Este procedimiento se lleva a cabo con el código escrito en (7.33).

Código 7.33 : Función de resampling sistemático para PF (Python)

```
    Q = np.cumsum(w)
    u0 = np.random.uniform(0,1,(1,1))/M

    while n < M:
        u = u0 + n/M
        while Q[m] < u:
            m = m+1
        index[0, n]= m
        n = n+1

    for i in range(M):
        xpar[0, i] = par[0, int(index[0,i])]

    return xpar
```

Ahora que las partículas se encuentran ponderadas por los pesos mediante el proceso de resampling, el código en (7.34) reinicia los valores de w mediante una matriz de unos dada por la función *ones(X)*, la cual es normalizada por un factor $1/Npar$. La estimación del estado que se retorna a la función *PFEstimation* es la suma de los elementos resultares al realizar el producto punto entre los pesos equivalentes normalizados y el conjunto ponderado de partículas *xpar*.

Código 7.34 : Cómputo de pesos normalizados y estado estimado con PF (Python)

```
1       w = np.dot(1/self.Npar, np.ones((1,self.Npar)))
2       self.xtt = np.sum(w*self.xpar, axis = 1)
```

Finalmente, el vector *xpar* se utiliza para operar sobre las funciones del modelo, resultando en un nuevo conjunto inicializado apropiadamente para el instante $t+1$. Esto se puede ver en el código (7.35). El conjunto resultante se almacena en la variable *xpar* del objeto *ParticleFilter*.

Código 7.35 : Cómputo partículas para el instante $t+1$ (Python)

```
1       self.xpar = (np.dot(self.G.A, self.xpar) + np.dot(self.G.B, self.G.ut[t]) + np.dot
        (chol(self.G.Q), np.random.normal(0, 1, (self.G.n, self.Npar))))
```

Se confirma que esta implementación del filtro de partículas cumple requerimiento mínimo de funcionamiento con un procedimiento equivalente al aplicado con los métodos anteriores, resultando en los gráficos de la figura 7.11. Para este caso no importa el número de partículas elegido, pues la medición carece de no linealidades. Los estados estimados en el tiempo poseen error mínimo y cercano a cero en relación con el vector de estados del modelo.

Los códigos completos para la implementación en Python del filtro de partículas y resampling se pueden encontrar en los anexos 9.57 y 9.58.

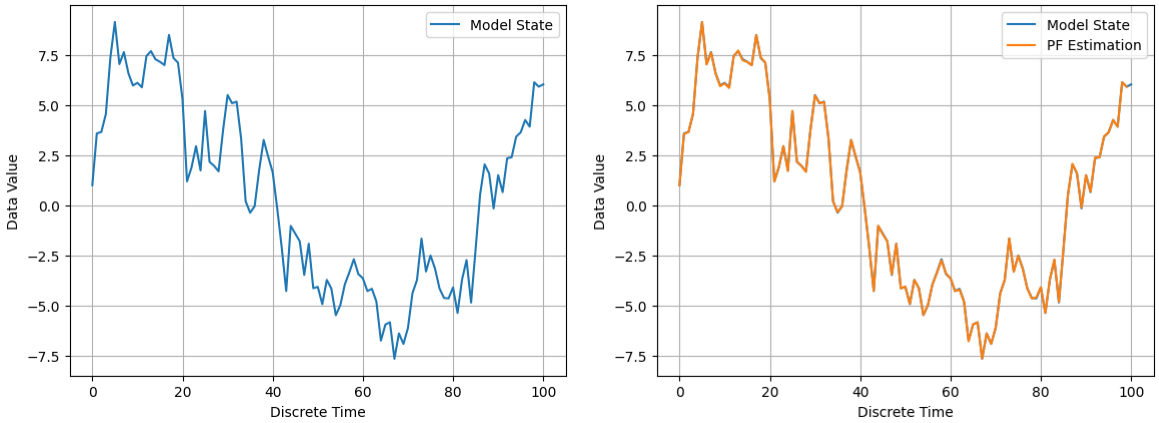


Figura 7.11: Comparación de gráficos de estado del modelo y estimación por filtro de partículas basado en MCMC con cuantización y ruido bajo.

7.4. Implementación del Filtro Suma de Gaussianas en Python

El filtro Suma de Gaussianas posee la implementación más compleja entre los métodos estudiados en el proyecto. A diferencia de las versiones del filtro de Kalman y el filtro de partículas, no se cuenta con una implementación equivalente en Python extraída de fuentes publicadas para usar como ejemplo. Se depende completamente de la reproducción del ejemplo implementado en MATLAB, que en sí se basa en un diseño expresado en la literatura. Debido a esto, varias operaciones realizadas son análogas y no se explican en detalle en esta sección. Se recomienda regresar a la sección de modelo matemático e implementación en MATLAB del filtro Suma de Gaussianas para encontrar más información.

Se crea una función *GSFEstimation(G, xin, Pin)*, encargada de ejecutar el algoritmo para N iteraciones, y una clase *GaussianSumFilter(object)*, donde se declaran los parámetros y las funciones de la ejecución. La función *GSFEstimation* posee características idénticas con respecto a los otros métodos, tanto en los argumentos que recibe, como en estructura. Aquí se inicializa el objeto *GaussianSumFilter* y se ejecutan sus funciones internas en bucle para llenar un vector de estimaciones en instantes $t = 1, \ldots, N$.

La clase *GaussianSumFilter* comienza con la inicialización de sus variables internas. En esta parte del código, se implementa una subclase nombrada *TimeMeasUp*, que se puede observar en (7.36). Se utiliza para definir estructuras que contengan parámetros de la PDF de filtraje como los pesos w_{ell}, medias x_{ell} y covarianzas S_{ell} en las diferentes etapas del algoritmo y en cada instante t. En la implementación de MATLAB, estas estructuras se separan en *measup* para la corrección y *timeup* para la predicción, sin embargo, como su formato es el mismo y solo cambian los valores que almacenan, en la implementación de Python se puede utilizar una clase única con múltiples instancias.

Código 7.36 : Definición de clase para almacenar parámetros de la PDF (Python)

```
1      class TimeMeasUp:
2          def __init__(self, G = None):
3              self.w = [] if G is None else G.w
4              self.mu = [] if G is None else G.mu
5              self.Sigma = [] if G is None else G.Sigma
```

Otra inicialización importante es $p(y_t|x_t)$. Se escribe la función *model_ pytxt(G)*, que toma el modelo G y entrega una aproximación de la PDF calculada por la regla de cuadratura Gauss-Legendre con Kv puntos, donde Kv es un parámetro almacenado en G. Las variables se definen con el código (7.37). La función *GaussLegrende(X)* proviene de la sub-biblioteca *polynomial* de numpy, y entrega los puntos y pesos apropiados dado el valor entero X ingresado como argumento. Estos valores retornan como matrices $1 \times X$, lo cual es conflictivo con las operaciones matriciales que se deben realizar durante este procedimiento. Debido a esto, se requiere cambiar los vectores xj y wj a matrices $X \times 1$ mediante la función *reshape(m,n)*. La PDF se crea con una clase *Model*, que le otorga propiedades Xj, muj y $betaj$.

Código 7.37 : Cómputo de pesos y puntos Gauss-Legendre de la PDF (Python)

```
1       xj, wj = GaussLegrende(G.Kv)
2       xj = np.array(xj).reshape(G.Kv, 1)
3       wj = np.array(wj).reshape(G.Kv, 1)
4       yx = Model()
```

Teniendo todas las variables necesarias, se declaran las funciones internas de la clase *GaussianSumFilter*. Estas son *MeasurementUpdate(self, t)* y *TimeUpdate((self, t)*, cuyos propósitos son los mismos que sus equivalentes en las versiones del filtro de Kalman (anexos 9.55 y 9.56). La corrección empieza computando $p(y_t|x_t)$ con los valores almacenados en y_t, seguido por la declaración los arreglos vz, md y ps con el código (7.38). Estos arreglos son de tres, dos y una dimensión respectivamente.

Código 7.38 : Definición de matrices para corrección de estados con GSF (Python)

```
1       ell = 0
2       self.Mtt = self.G.Kv*self.Mt_tm1
3       vz = np.zeros((self.Mtt, self.n, self.n))
4       md = np.zeros((self.n, self.Mtt))
5       ps = np.zeros((1, self.Mtt))
```

Python utiliza un formato diferente a MATLAB para definir y operar con arreglos de múltiples dimensiones. Mientras que MATLAB utiliza $M(:,:,a)$ para referirse al contenido de la posición a de la tercera dimensión de M, Python utiliza $M[a]$. Para modificar los valores de la columna a de la matriz M, MATLAB y Python utilizan formatos semejantes $M(:,a)$ y $M[:,a]$. Por último, MATLAB puede referirse al elemento en el índice a de un arreglo unidimensional M con $M(a)$, mientras que Python debe usar $M[0,a]$ si se desea mantener la consistencia con los otros formato para definir matrices (arreglo de arreglos incluso en casos donde la matriz es $1 \times n$).

Los valores de los arreglos *vz*, *md* y *ps* se calculan con operaciones equivalentes a MATLAB. El arreglo *ps* contiene los pesos de la ecuación (5.38), los cuales deben ser normalizados según (5.40), mediante la función *normalized_ weights(x)* escrita en (7.39). Los únicos pasos que difieren con la versión de esta función en MATLAB son la aplicación de las funciones *max(x)* y *argmax(x)* de numpy, que entregan el elemento de máximo valor en el arreglo x y el índice donde se posiciona, respectivamente (en casos donde el arreglo es de dimensión mayor a 1, se debe especificar el eje en los argumentos de *arg* y *argmax*). Además, se omite escribir comandos que no son utilizados en MATLAB, tal como la confirmación que x es un vector y el cálculo de *lse*.

Código 7.39 : Función de normalización de pesos para GSF (Python)

```
    n = len(x[0])
    e = np.zeros((1, n))
    a = np.max(x)
    k = np.argmax(x)
    s = 0

    for i in range(n):
        e[0,i] = np.exp(x[0,i] - a)
        if i != k:
            s = s + e[0,i]

    return e/(1 + s)
```

El resto de las operaciones de la etapa de corrección son equivalentes a MATLAB.

Las propiedades de la PDF de filtraje resultante se almacenan en la variable MU con el código (7.40), para posteriormente ser utilizadas al calcular el estado y covarianza estimados en el instante t con la función *gmmTomuP(Data)*.

Código 7.40 : Registro de la PDF del estado despues de la corrección (Python)

```
    # Save the filtered pdfs
    self.MU.w = ps
    self.MU.mu = md
    self.MU.Sigma = vz
    self.measup.append(self.MU)
```

Se procede a la etapa de predicción. En esta parte de la implementación no hay diferencias notables con respecto al código de MATLAB, solo hace falta aplicar el formato de Python explicado en la etapa de corrección para encontrar las propiedades de la PDF para el instante $t+1$. Los elementos resultantes se almacenan en la variable interna TM. Estos pasos se realizan con el segmento de código (7.41).

Código 7.41 : Etapa de predicción y registro del estado con GSF (Python)

```
    # TIME UPDATE STEP
    S = np.zeros((self.Mtt, self.n, self.n))
    m = np.zeros((self.n, self.Mtt))
    w = np.zeros((1, self.Mtt))

    for k in range(self.Mtt):
        w[0, k] = self.MU.w[0, k]
        m[:, k] = np.dot(self.G.A, self.MU.mu[:, k]) + np.dot(self.G.B, self.G.ut[t])
        S[k] = np.dot(self.G.A, np.dot(self.MU.Sigma[k], self.G.A.T)) + self.G.Q

    # Save the predicted pdf
    self.TM.w = w
    self.TM.mu = m
    self.TM.Sigma = S
    self.timeup.append(self.TM)
```

El resto del algoritmo consiste en aplicar la reducción gaussiana mediante el algoritmo de Kullback-Leibler, con el propósito de limitar el número de componentes gaussianas del GMM a un rango dado por los parámetros $minG$ y $maxG$ del modelo G. La función *gaussian_reduction(LI, LS, LD, ps, md, vz)* es el mismo proceso y funciones que en la implementación de MATLAB, teniendo cuidado de aplicar las particularidades de Python para operaciones matriciales. Al completar el proceso de reducción, se deben almacenar las PDFs, los estados y covarianzas obtenidas en las estructuras apropiadas, dependiendo si deben utilizar como retroalimentación para la siguiente iteración, o retornarlos como resultado de la estimación.

Esta sección finaliza con la confirmación que la implementación del algoritmo filtro Suma de Gaussianas cumple con el requerimiento mínimo de funcionamiento, presente en la figura 7.12, donde nuevamente se puede observar que el vector de estados estimado posee error mínimo y cercano a cero en relación con el vector de estados del modelo. Al igual que en el filtro de partículas, la cantidad de puntos Kv elegidos para la aproximación por regla Gauss-Legendre del modelo no afecta los resultados de la estimación. Con esta cualidad determinada en cada método de estimación de estado implementado en Python, se puede proceder a evaluar su rendimiento bajo condiciones de ruido y cuantización.

Los código completos para la implementación en Python del filtro de Suma de Gaussianas, y sus funciones asociadas, se pueden encontrar en los anexos 9.59, 9.60, 9.61 y 9.62.

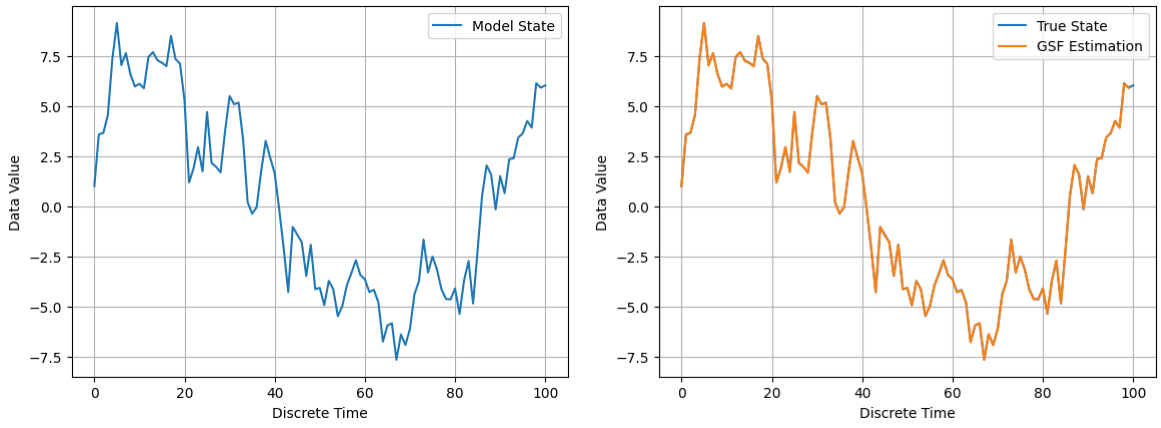


Figura 7.12: Comparación de gráficos de estado del modelo y estimación por filtro Suma de Gaussianas con cuantización y ruido bajo.

7.5. Evaluación y Comparación de Métodos de Estimación de Estados

En esta sección se busca comparar el rendimiento del filtro Suma de Gaussianas implementado en Python para solucionar un ejemplo de problema de estimación con información cuantizada en relación con su implementación en MATLAB y a ejemplos de los otros algoritmos estudiados en este documento. Los criterios de comparación elegidos son la exactitud de filtraje medida por el error cuadrático medio entre el vector de estado modelado y el vector estimado por el algoritmo, y el tiempo de ejecución requerido para estimar todos los estados en cada instante t, donde $t = 1, \ldots, N$. Además, se incluye una comparación de la cantidad máxima de memoria consumida durante la ejecución de cada algoritmo. Se decide incluir este criterio, pues puede entregar información de interés para elegir un método apropiado empleando un lenguaje de programación como Python, cuya naturaleza dinámica es intrínsecamente ineficiente para asignar recursos de memoria de forma automática.

Las ecuaciones (7.63), (7.64) y (7.65) describen el modelo del espacio de estados con salida cuantizada elegido para la evaluación de cada algoritmo de interés. Las matrices de asociación entre vectores son de orden 1, pues estas condiciones permiten encontrar los requerimientos mínimos del método de estimación y generar representaciones gráficas más simples. Las perturbaciones w_t y v_t son distribuciones gaussianas de media 0, y sus matrices de covarianzas son $Q = [1.0]$ y $R = [0.5]$ respectivamente. El cuantizador $\mathfrak{q}\{\cdot\}$ esta definido por la expresión (2.12), con un paso de cuantización $\Delta_q = 7$.

$$x_{t+1} = [0.9]\, x_t + [1.0]\, u_t + w_t \tag{7.63}$$

$$z_t = [2.0]\, x_t + [0.5]\, u_t + v_t \tag{7.64}$$

$$y_t = \mathfrak{q}\{z_t\} \tag{7.65}$$

La primera comparación es entre dos ejemplos del filtro de Kalman estándar, el algoritmo escrito con base en la reproducción de la implementación del método en MATLAB, y el código desarrollado y publicado en GitHub por el usuario Gavin Gao. En la figura 7.14 se grafican el vector de estado modelado con los parámetros y las estimaciones del estado obtenidas desde ambas implementaciones. La tabla 7.4 contiene el promedio obtenido de las mediciones provenientes de diez repeticiones del procedimiento de estimación, sin variar los parámetros del modelo o simulación. El algoritmo basado en MATLAB entrega mejores resultados promedios en términos de la exactitud de estimación y memoria máxima utilizada durante la ejecución del programa, mientras que el algoritmo de Gavin Gao se destaca en el tiempo requerido de ejecución. El beneficio de realizar la estimación en un menor tiempo se encuentra en situaciones con un elevado número de muestras, a diferencia de las pruebas realizadas, donde solo se calculan cien. Sin embargo, trabajos más delicados demandan colocar prioridad a la exactitud de la estimación y al uso eficiente de recursos. Por lo tanto, se opta por utilizar el primer algoritmo para la evaluación final entre métodos, pues entrega un nivel de exactitud mayor utilizando una cantidad de recursos computacionales significativamente menor.

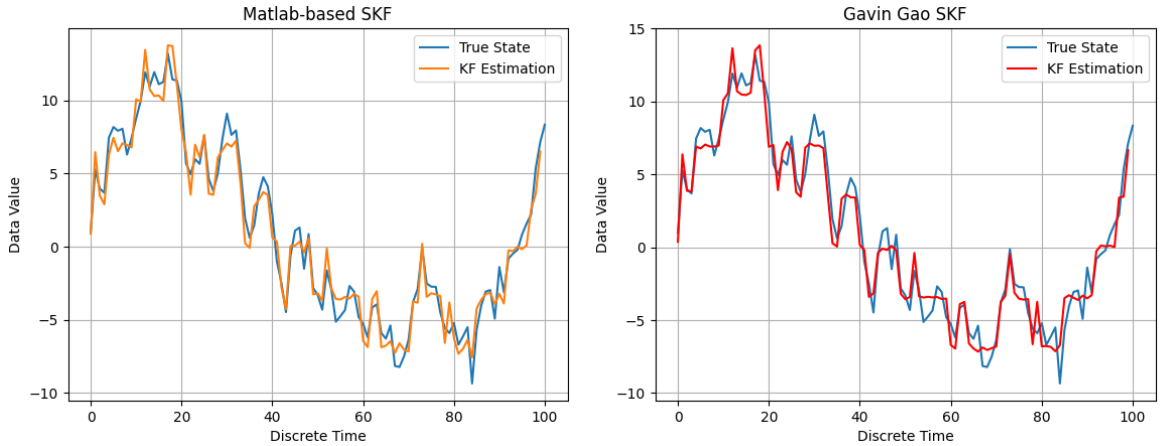


Figura 7.13: Gráficos de la estimación del estado en el tiempo mediante filtro de Kalman estándar con el algoritmo basado en MATLAB y el algoritmo de Gavin Gao.

Es importante destacar una peculiaridad en los resultados obtenidos en esta sección de la evaluación. El hecho que la implementación de Gavin Gao genere el vector de estimaciones en un menor tiempo promedio no es completamente consistente con el hecho que consume tres veces más memoria que el algoritmo basado en MATLAB. Las razones detrás de este comportamiento pueden ser varias, por ejemplo, el algoritmo de Gavin Gao puede estar declarando más variables que este ejercicio particular no utiliza, debido a que originalmente este diseño se creó para resolver problemas de localización en espacios de dos dimensiones, y, por lo tanto, busca entregar más información que la estimación de un sistema de primer orden. Otra razón puede ser una clase u operación definida en el algoritmo basado en MATLAB que requiere más tiempo para su ejecución.

Algoritmo	Filtraje		
	Error de estimación promedio (ECM)	Tiempo de ejecución promedio (segundos)	Memoria máxima promedio (bytes)
Filtro de Kalman estándar (Reproducción de MATLAB)	1.03165	0.02354	9278
Filtro de Kalman estándar (Algoritmo de Gavin Gao)	1.38149	0.01993	29256

Tabla 7.4: Tabla de promedios al evaluar el filtro de Kalman basado en la reproducción de código MATLAB y el algoritmo de Gavin Gao.

La comparación de las implementaciones del filtro de Kalman cuantizado muestran resultados similares al caso de la versión estándar. Se puede observar en la tabla 7.5 que el error de estimación promedio para el algoritmo basado en MATLAB incrementa levemente, mientras que el tiempo y memoria máxima promedio requeridos disminuyen. Este fenómeno se puede deber por la cuantización durante la etapa de corrección del algoritmo, números enteros redondeados requieren menor cantidad de bits para su almacenamiento y operaciones matemáticas con este tipo de datos tienden a ejecutarse en un menor tiempo. En contraste, al realizar una cuantización adicional la exactitud disminuye. El algoritmo de Gavin Gao, tras

realizar la modificación para la versión cuantizada, registra un incremento leve en el consumo de recursos computacionales promedio, en el error y en el tiempo de ejecución en comparación a la implementación del filtro de Kalman estándar, por lo que nuevamente se opta por utilizar el algoritmo basado en MATLAB.

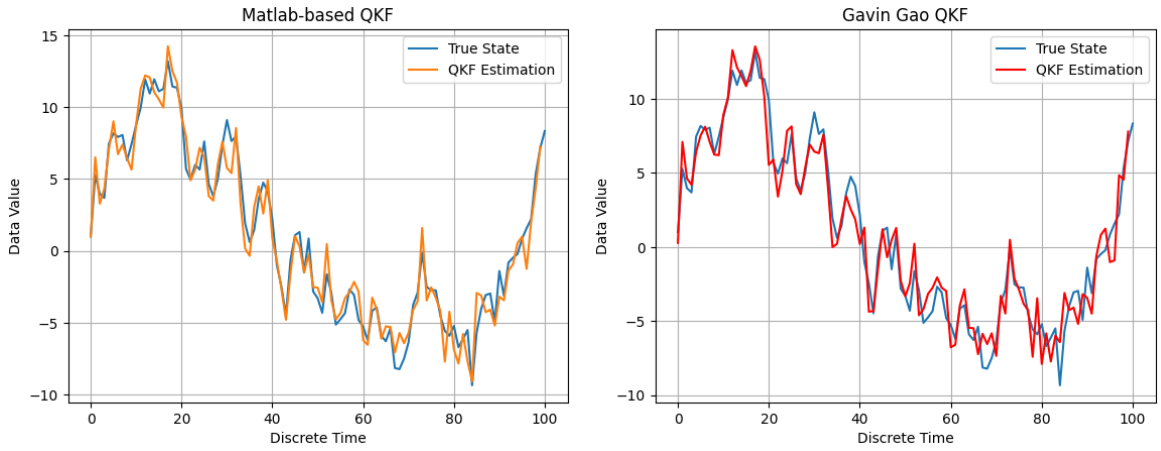


Figura 7.14: Gráficos de la estimación del estado en el tiempo mediante filtro de Kalman cuantizado con el algoritmo basado en MATLAB y el algoritmo de Gavin Gao.

Algoritmo	Filtraje		
	Error de estimación promedio (ECM)	Tiempo de ejecución promedio (segundos)	Memoria máxima promedio (bytes)
Filtro de Kalman cuantizado (Reproducción de MATLAB)	1.34904	0.02135	7644
Filtro de Kalman cuantizado (Algoritmo de Gavin Gao)	2.05668	0.02304	29690

Tabla 7.5: Tabla de promedios al evaluar el filtro de Kalman cuantizado basado en la reproducción de código MATLAB y el algoritmo de Gavin Gao.

Evaluar el filtro de partículas requiere un procedimiento más complicado que los casos anteriores. El diseño de Gavin Gao para el filtro de partículas requiere modificaciones para optimizarlo a un modelo que posee salida cuantizada. Se omiten varios comandos que no son requeridos para la comparación, tal como el observador en tiempo real y el bucle *for* aplicado para calcular individualmente los pesos, y se incluye la solución al problema de la cuantización dada por la fórmula (4.28), con comandos y herramientas idénticos a los utilizados en la implementación basada en MATLAB. El cómputo de los pesos se realiza con el código (7.42).

Código 7.42 : Cómputo de pesos para PF con salida cuantizada

```
lim_a = repmat(a,1,NP) - C*px[0] - D*u[0]
lim_b = repmat(b,1,NP) - C*px[0] - D*u[0]
w = mvn.cdf(lim_b.T, mean = 0, cov = 0.5) - mvn.cdf(lim_a.T, mean = 0, cov = 0.5)
pw[0] = w
```

Para probar su funcionamiento se realiza una estimación con ambos ejemplos seleccionados, tal que parámetros del modelo son los mismos que para las evaluaciones previas y se utilizan $Np = 100$ partículas. El resultado se puede observar en el gráfico de la figura 7.15.

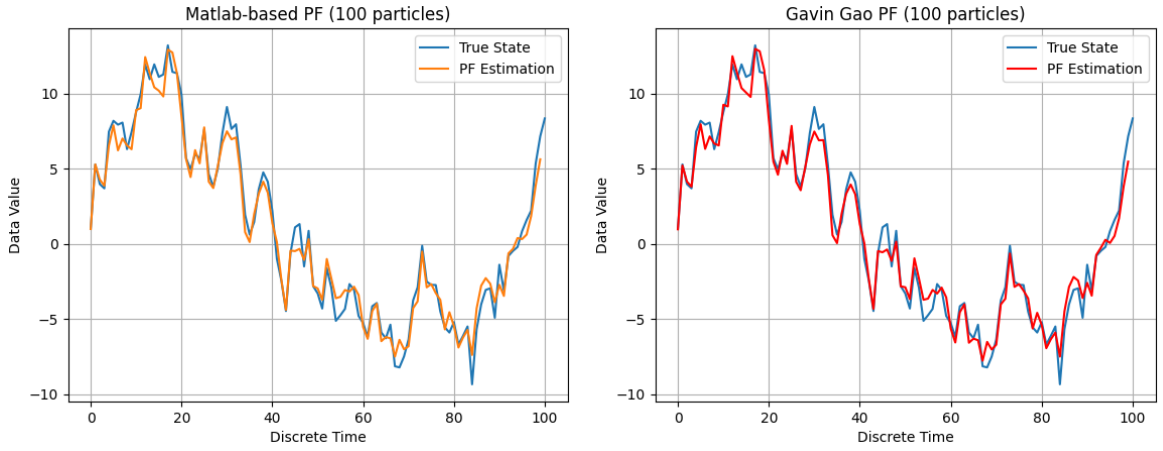


Figura 7.15: Gráficos de la estimación del estado en el tiempo mediante filtro de partículas con el algoritmo basado en MATLAB y el algoritmo de Gavin Gao, para 100 partículas.

Este método requiere realizar un mayor número de experimentos que el filtro de Kalman estándar y cuantizado, pues cuenta con una variable definida por usuario adicional, correspondiente al número de partículas para la estimación. Los resultados se pueden observar en la tabla 7.6, donde se organizan los promedios encontrados al realizar diez estimaciones en cada instancia con 100, 500, 1000 y 10000 partículas en cada implementación. De la información recopilada se puede identificar un comportamiento peculiar en la implementación basada en MATLAB, donde el error cuadrático medio tiende a incrementar junto con el número de partículas. Esto es contradictorio con el funcionamiento requerido del algoritmo, donde la exactitud sube al estimar con más partículas y sugiere una implementación incorrecta del algoritmo en este ejemplo. Este comportamiento no está presente en la implementación de Gavin Gao. En cuanto a las otras propiedades, se observa que el algoritmo de Gavin Gao requiere más tiempo de ejecución que el otro algoritmo, pero esto se compensa en su requerimiento de memoria, el cual es significativamente menor. Esta propiedad es consistente en ambos ejemplos evaluados, pues la cantidad de memoria promedio requerida para ejecutar el algoritmo sube al aumentar el número de partículas, y además, este incremento es aproximadamente a razón del aumento de partículas, excepto en el algoritmo de Gavin Gao cuando se pasa de 100 a 500 partículas. Por otro lado, la diferencia de exactitud no es drásticamente diferente entre ambos, entonces conviene utilizar la implementación de Gavin Gao para la comparación general de métodos.

Algoritmo	Nº de partículas	Filtraje		
		Error de estimación promedio (ECM)	Tiempo de ejecución promedio (segundos)	Memoria máxima promedio (bytes)
Filtro de partículas (Reproducción de MATLAB)	100	0.67461	0.31778	115441
	500	0.68030	1.33949	456185
	1000	0.68769	2.57461	886191
	10000	0.68837	25.25192	8511361
Filtro de partículas (Algoritmo de Gavin Gao)	100	0.71050	0.38244	46694
	500	0.69652	1.62125	139478
	1000	0.69468	2.81124	255318
	10000	0.68491	27.01589	2089758

Tabla 7.6: Tabla de promedios al evaluar el filtro de partículas basado en la reproducción de código MATLAB y el algoritmo de Gavin Gao, para diferente número de partículas.

El paso final del trabajo experimental es la evaluación y clasificación de cada método de estimación estudiado, según los criterios utilizados en el ejercicio de comparación anterior. La información correspondiente se extrae de un nuevo conjunto de experimentos con el mismo modelo del espacio de estados, que buscan encontrar un promedio definitivo para cada criterio. En la figura 7.16 se observan gráficos de los vectores de estimación resultantes al ejecutar cada algoritmo de Python seleccionado en función de tiempo. Los métodos presentes son la implementación basada en MATLAB del filtro de Kalman estándar y cuantizado, el filtro de partículas diseñado por Gavin Gao, y el filtro Suma de Gaussianas.

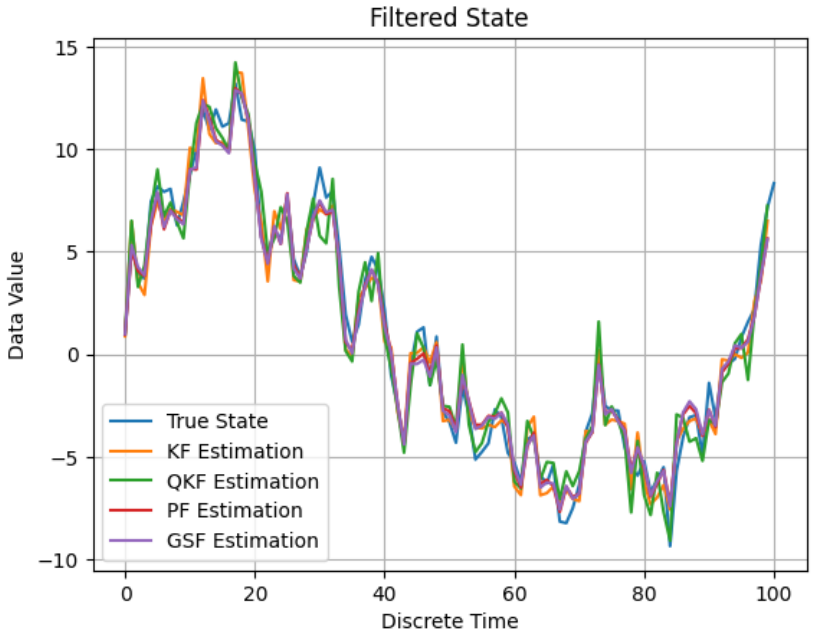


Figura 7.16: Gráfico del vector de estado del modelo y vectores de estimación para cada método implementado en función del tiempo discreto (filtro de partículas de 100 partículas).

La tabla 7.7 resume los resultados de la evaluación, estructurada de forma similar a la tabla 6.3. Se rescatan algunas observaciones sobre el rendimiento de Python y MATLAB, con respecto al único criterio que ambas tablas poseen en común, el error de estimación del filtraje obtenido mediante error cuadrático medio. El filtro Suma de Gaussianas muestra el menor error de estimación entre los métodos aplicados, y una diferencia entre el error en Python y el error en MATLAB de 0.0098, con la menor exactitud presente en la implementación en

Python. Para determinar si este es un patrón asociado a las técnicas generalmente empleadas en los códigos de Python analizados o al lenguaje de programación, se puede comparar el error promedio obtenido durante la ejecución del filtro de Kalman estándar y cuantizado, ambas implementaciones basadas en el código de MATLAB. La versión estándar posee una diferencia en el error de 0.01785 con respecto a MATLAB, mientras que en la versión cuantizada esta diferencia es de -0.51256. La implementación en Python exhibe una disminución leve en la exactitud de la estimación, con excepción del filtro de Kalman cuantizado, donde los resultados muestran un mejor rendimiento total con respecto a su implementación análoga en MATLAB. Las mediciones del filtro de partículas de Gavin Gao son consistentes con las observaciones, pues el error promedio con un mismo número de partículas es generalmente mayor en comparación las implementaciones en MATLAB de mejor rendimiento, aquellas que utilizan algoritmo MCMC del tipo Random-Walk Metropolis (RWM), y resampling sistemático (SYS) o multinomial (ML).

Posición	**Filtraje**					
	Error de estimación promedio (ECM)	**Algortimo**	**Tiempo de ejecución promedio (segundos)**	**Algoritmo**	**Memoria máxima promedio (bytes)**	**Algoritmo**
1	0.68220	GSF	0.02070	QKF	7644	QKF
2	0.68491	PF(10000)	0.02723	KF	9278	KF
3	0.69468	PF(1000)	0.39145	PF(100)	51435	GSF
4	0.69652	PF(500)	0.40296	GSF	55918	PF(100)
5	0.71050	PF(100)	1.66173	PF(500)	139518	PF(500)
6	1.03165	KF	2.82145	PF(1000)	255494	PF(1000)
7	1.34904	QKF	27.11921	PF(10000)	2088854	PF(10000)

Tabla 7.7: Comparación de implementación en Python de algoritmos de filtraje para estimación con datos cuantizados. Algoritmos evaluados: filtro de Kalman estándar (KF), filtro de Kalman cuantizado (QKF), filtro de partículas con X partículas (PF(X)) y filtro suma de Gaussianas (GSF).

El ejemplo que propone este documento para la implementación del filtro Suma de Gaussianas en Python es el método de estimación que ofrece el mejor rendimiento en comparación a los ejemplos de métodos clásicos evaluados, al combinar los tres criterios de comparación. Sus resultados presentan el menor error promedio de estimación de los experimentos realizados, compitiendo solo con el ejemplo del filtro de partículas implementado cuando este utiliza 10000 partículas. En términos de tiempo de ejecución es superado por ambas versiones del filtro de Kalman y por el filtro de partículas con un bajo número de partículas, pero tomando en cuenta el error, logra estimar cien muestras en un tiempo 67 veces menor que el ejemplo del filtro de partículas de 10000 partículas, aproximadamente. Finalmente, utiliza una cantidad promedio de memoria comparable al filtro de partículas de 100 partículas, mientras que su competencia, en términos del error de estimación, utiliza aproximadamente 40 veces más memoria en promedio durante su ejecución.

8. Resultados y Conclusiones

En este trabajo se realizó la implementación en Python de algoritmos de estimación de estados que resuelven el problema de filtraje cuando las mediciones de la salida observada es un conjunto de datos cuantizados, entre los cuales se incluyen varios métodos clásicos y el diseño para filtro Suma de Gaussianas descrito en [3]. El propósito general es encontrar un marco de referencia para comparar el rendimiento del filtro Suma de Gaussianas con ejemplos de algoritmos que cumplen el mismo propósito, en un entorno de programación distinto de MATLAB, uno que sea más frecuentemente utilizado en la industria para la solución problemas reales de control y administración de datos. Además, se buscaba conseguir que los resultados obtenidos fueran a lo menos comparable con el rendimiento de sus respectivas implementaciones en MATLAB, catalogadas en la tabla 6.3 según experimentos realizados en [5].

Se utilizan ejemplos de cada algoritmo de interés en MATLAB, proporcionados por Angel L. Cedeño, para escribir códigos que implementen los métodos de estimación en Python, basándose en herramientas y estructuras de datos similares a las utilizadas en MATLAB. Luego, se comparan estas implementaciones, excepto el filtro Suma de Gaussianas, con ejemplos en Python de los mismos métodos de estimación publicados en línea para seleccionar aquellos que permitan ampliar el marco de referencia que se desea proporcionar, al emplear implementaciones que ofrezcan mejor rendimiento. Se seleccionaron los diseños del filtro de Kalman y filtro de partículas del programador y usuario de GitHub, Gavin Gao, para esta tarea, debido a su buena legibilidad de código que facilita la tarea de optimizarlos para trabajar con datos cuantizados.

Durante la evaluación de los algoritmos se emplean diferentes herramientas matemáticas y comandos de administración de sistemas para medir sus propiedades de ejecución. Los criterios utilizados son la exactitud de estimación promedio de cada algoritmo, medido con el error cuadrático medio entre el modelo y el resultado del algoritmo dado por la fórmula (6.62), el tiempo promedio de ejecución requerido por cada método para estimar cien muestras, medido en segundos con la biblioteca *time* de Python, y la memoria máxima promedio utilizada por cada método durante su ejecución, cuyo perfil se realiza con la biblioteca *tracemalloc* de Python.

Los resultados finales de la evaluación se tabulan en 7.7, ordenados en cada criterio de manera descendente según su rendimiento. Al comparar los resultados obtenidos de las evaluaciones del filtro de Kalman estándar, el filtro de Kalman cuantizado, el filtro de partículas MCMC con resampling sistemático y el filtro Suma de Gaussianas, con la información de la tabla 6.3, se pueden rescatar las siguientes conclusiones:

1. El filtro Suma de Gaussianas es el método de estimación con PDFs de filtraje que mejor estima las PDFs del modelo del espacio de estados en los experimentos realizados con Python, lo cual es consistente con los resultados obtenidos en los experimentos de MATLAB estudiados en secciones anteriores. El error cuadrático medio observado en ambos casos difiere en 0.0098 aproximadamente, con MATLAB mostrando mayor exactitud. El resto de los algoritmos se ordenan de la misma manera que en la tabla 6.3, con los filtros de partículas en posición descendente según su número de partículas, seguido del filtro de Kalman estándar y finalmente el filtro de Kalman cuantizado. En cada caso se consigue una estimación más exacta con MATLAB, excepto con el filtro de Kalman cuantizado, donde Python supera a MATLAB por un margen notable. Esto sugiere la existencia de un margen dentro de este experimento donde MATLAB exhibe un mejor procesamiento de datos. Basándose en estas observaciones, notando que los experimentos combinan códigos propios e implementaciones publicadas por otros programadores, se puede afirmar que la pérdida de rendimiento está asociada a técnicas utilizadas en la escritura de los códigos en Python o a características nativas del lenguaje de programación. Confirmar una causa definitiva requiere el estudio de más ejemplos y llevar a cabo un mayor número de experimentos.

2. En Python, los algoritmos basados en el cálculo directo del estado y covarianza en cada instante (filtro de Kalman estándar y cuantizado), y el filtro de partículas con pocas partículas pueden estimar el estado con datos cuantizados en un menor tiempo, sin embargo, estos métodos son deficientes en términos de la exactitud de sus estimaciones al trabajar con datos cuantizados. Tomando como referencia el tiempo de ejecución del ejemplo en Python del filtro de partículas evaluado con 10000 partículas, obtener un error similarmente pequeño al ejemplo del filtro Suma de Gaussianas requiere un tiempo de ejecución significativamente mayor.

3. Las tendencias encontradas en el tiempo de ejecución para cada algoritmo se repiten para la memoria máxima asignada al algoritmo durante su ejecución. El filtro Suma de Gaussianas computa las PDFs de filtraje con una cantidad de recursos significativamente menor que el filtro de partículas para obtener una estimación con el mismo nivel de exactitud. Se puede afirmar que la implementación en Python de este método de estimación cumple con el propósito que inspiró su diseño.

4. Si se desea afirmar que los resultados obtenidos de la evaluación de los algoritmos en Python son por lo menos comparable con sus respectivas implementaciones en MATLAB, es necesario considerar las características del problema de estimación que se desea resolver. En los experimentos realizados, los estados del modelo poseen un orden de magnitud de 10^1, y en estos casos, un aumento del error de estimación de 0.0098 en Python en comparación con MATLAB puede no tener un efecto significativo en el resultado final, mientras que en sistemas con órdenes de magnitud más bajos puede tener un efecto negativo. Por otro lado, existe la posibilidad que los ejemplos provistos estén sujetos a mejoras una vez que se aprendan mejores técnicas de programación. Es necesario tomar en cuenta estos puntos si se desea expandir este trabajo en el futuro, sin embargo, con la información que se tiene hasta el momento, el diseño del filtro Suma de Gaussianas escrito en Python muestra un rendimiento favorable, relativo a cada ejemplo analizado.

8.1. Trabajos Futuros

Los resultados obtenidos permiten obtener una idea inicial sobre la capacidad que tiene el filtro Suma de Gaussianas para competir con otros métodos de estimación que trabajan con datos cuantizados, reemplazando el entorno MATLAB por Python. Por el momento, los resultados han sido favorables al estudiar solamente el proceso de filtraje. Sin embargo, esta idea se puede expandir considerablemente, y la decisión de realizar el reemplazo se puede tomar con mayor seguridad, si se evalúan otras soluciones del problema de estimación con las mismas técnicas empleadas en el trabajo realizado. Con este fin, se proponen los siguientes trabajos futuros:

1. Estudiar técnicas de programación en Python para mejorar el diseño del filtro Suma de Gaussianas propuesto en este documento, con el fin de maximizar su rendimiento en términos de error de estimación, tiempo de ejecución y gasto computacional.

2. Estudiar otros ejemplos de métodos de estimación clásicos para el ampliar el marco de referencia con el que se realiza la comparación. En el caso de continuar utilizando los códigos diseñados por Gavin Gao, determinar las causas específicas detrás del alto consumo de memoria promedio en sus implementaciones del filtro de Kalman estándar y cuantizado.

3. Estudiar el proceso de suavizado con datos cuantizados y encontrar técnicas para implementar el suavizado de Kalman estándar, suavizado de Kalman cuantizado y suavizado de partículas en Python. Su propósito sería aprovechar plenamente los experimentos de la referencia [5] y demostrar si los datos observados para el tiempo de ejecución de los algoritmos son particulares del proceso de filtraje o una consecuencia directa del entorno de programación seleccionado.

4. Extender el filtro Suma de Gaussianas a un suavizado Suma de Gaussinas, donde el proceso pretende estimar los estados condicionados por el conjunto de todas las mediciones realizadas en t, donde $t = 1, \ldots, N$. Esto permite usar el diseño de doble filtro propuesto en [26] y desarrollado en [3], donde la ecuación de suavizado se divide en una recursión de filtraje en reversa y con base en esta estimación en reversa se estiman los estados y las distribuciones de suavizado. Implementar y evaluar apropiadamente este diseño permitirá rescatar conclusiones definitivas con respecto a la viabilidad de la migración a Python, y determinar si se puede competir con métodos clásicos para resolver problemas prácticos de estimación de estados con datos cuantizados.

9. Anexos

9.1. Códigos de MATLAB

Esta sección recopila los códigos de MATLAB usados para implementar los algoritmos de estimación estudiados en este documento. Cada ejemplo fue proporcionado por Angel L. Cedeño para ser usados como referencia para crear un código propio en Python, que implemente el algoritmo utilizando estructuras y herramientas equivalentes a las nativas de MATLAB.

El primer grupo son códigos para implementar el filtro de Kalman estándar y el filtro de Kalman cuantizado.

Código 9.43 : Filtro de Kalman Estándar (MATLAB)

```
function [F] = standard_kalman(G)
    % Initial values
    F.xtmt{1}=G.mm.Icon.xin;
    F.Stmt{1}=G.mm.Icon.Pin;
    % Standard Kalman Filter
    for t=1:1:G.sim.N
        % Kalman Gain
        F.K{t}=(F.Stmt{t}*G.mm.C')/(G.mm.R+G.mm.C*F.Stmt{t}*G.mm.C');
        % Measurement Update
        F.xtt{t}=F.xtmt{t}+F.K{t}*(G.sim.yt{t}-G.mm.C*F.xtmt{t}-G.mm.D*G.sim.ut{t});
        F.Stt{t}=(eye(length(G.mm.C))-F.K{t}*G.mm.C)*F.Stmt{t};
        % Time Update
        F.xtmt{t+1}=G.mm.A*F.xtt{t}+G.mm.B*G.sim.ut{t};
        F.Stmt{t+1}=G.mm.Q+G.mm.A*F.Stt{t}*G.mm.A';
    end
end
```

Código 9.44 : Filtro de Kalman Cuantizado (MATLAB)

```
function [F] = quantized_kalman(G)
    % Initial values
    F.xtmt{1}=G.mm.Icon.xin;
    F.Stmt{1}=G.mm.Icon.Pin;
    % Standard Kalman Filter
    for t=1:1:G.sim.N
        % Kalman Gain
        F.K{t}=(F.Stmt{t}*G.mm.C')/(G.mm.R+G.mm.C*F.Stmt{t}*G.mm.C');
```

```
        % Measurement Update
        F.xtt{t}=F.xtmt{t}+F.K{t}*(G.sim.yt{t}-Q(G,G.mm.C*F.xtmt{t}+G.mm.D*G.sim.ut{t
    }));
        F.Stt{t}=(eye(length(G.mm.C))-F.K{t}*G.mm.C)*F.Stmt{t};
        % Time Update
        F.xtmt{t+1}=G.mm.A*F.xtt{t}+G.mm.B*G.sim.ut{t};
        F.Stmt{t+1}=G.mm.Q+G.mm.A*F.Stt{t}*G.mm.A';
    end
end
function q=Q(G,zt)
    q=G.mm.Delta*round(zt/G.mm.Delta);
end
```

Continua con los códigos para la implementación de un filtro de partículas MCMC con método de resampling sistemático. El proceso de resampling se escribe como una función auxiliar con código separado del algoritmo principal.

Código 9.45 : Filtro de Partículas MCMC (MATLAB)

```
function [PF] = particle_filter(G)
    n= size(G.mm.A,1);

    y=cell2mat(G.sim.yt);
    a = y - G.mm.Delta/2;
    b = y + G.mm.Delta/2;
    xpar = repmat(G.mm.Icon.xin,1,G.sim.Npar)+chol(G.mm.Icon.Pin)*randn(n,G.sim.Npar);

    for t=1:G.sim.N
        lim_a = repmat(a(t),1,G.sim.Npar) - G.mm.C*xpar-G.mm.D*G.sim.ut{t};
        lim_b = repmat(b(t),1,G.sim.Npar) - G.mm.C*xpar-G.mm.D*G.sim.ut{t};
        w = transpose(mvncdf(lim_b',0,G.mm.R) - mvncdf(lim_a',0,G.mm.R));
        w = w/sum(w);

        xpar=systematic_resample(w,xpar);
        w=(1/G.sim.Npar)*ones(1,G.sim.Npar);

        PF.xpres{t} = xpar;
        PF.xtt{t} = sum(w.*xpar,2);
        xpar = G.mm.A*xpar + G.mm.B*G.sim.ut{t} + chol(G.mm.Q)*randn(n,G.sim.Npar);
    end
    PF.xpredN=xpar;
end
```

Código 9.46 : Función de Resampling Sistemático (MATLAB)

```
function xpar=systematic_resample(w,par)
    M=length(w);
    index=zeros(1,M);
    Q=cumsum(w);
    n=0;
    u0=unifrnd(0,1,1,1)/M;
    m=1;
    while (n<M)
        u=u0+n/M;
        while (Q(m)<u)
            m=m+1;
        end
        n=n+1;
        index(n)=m;
    end
    xpar = par(:,index);
end
```

El último grupo es el filtro Suma de Gaussianas y las funciones correspondientes a los procesos matemáticos utilizados para el cómputo de estados. Entre estas se encuentra el código para computar el estado y covarianza $1|0$ según propiedades de GMM, la aproximación de la PDF $p\left(y_t|x_t\right)$ por regla de Gauss-Legendre, la normalización de los pesos $\overline{\gamma}_{t|t}^{k}$ y la reducción de Gauss mediante algoritmo de Kullback-Leibler.

Código 9.47 : Filtro Suma de Gaussianas (MATLAB)

```
function [F]=gaussian_sum_filter(G)

    % Initial predictive pdf
    TU = G.mm.Icon;

    % Save the predicted pdf at time 1
    F.timeup{1} = TU;

    % Save the predicted state at time 1
    [x,P]=gmmTomuP(TU);
    F.xtmt{1}=x;
    F.Stmt{1}=P;

```

```
    n=size(G.mm.A,1);
    Mt_tm1=length(TU.w);

    [yx] = model_pytxt(G);
    for t=1:1:G.sim.N
        % Computing p(yt|xt)
        muj=yx.muj{t};
        Xj=yx.Xj{t};
        betaj=yx.betaj{t};

        % MEASUREMENT UPDATE STEP
        ell=0;
        Mtt=G.sim.Kv*Mt_tm1;
        vz=zeros(n,n,Mtt);
        md=zeros(n,Mtt);
        ps=zeros(1,Mtt);
        for k=1:1:Mt_tm1
            for j=1:1:G.sim.Kv
                ell=ell+1;
                my=Xj(j)-G.mm.C*TU.mu{k}-G.mm.D*G.sim.ut{t}-muj;
                Zy=G.mm.R+G.mm.C*TU.Sigma{k}*G.mm.C';
                K=(TU.Sigma{k}*G.mm.C')/Zy;
                vz(:,:,ell)=(eye(n)-K*G.mm.C)*TU.Sigma{k};
                md(:,ell)=TU.mu{k}+K*my;
                ps(ell)=log(betaj(j))+log(TU.w{k})-0.5*(my'*(Zy\my))-0.5*log(det(2*pi*
    Zy));
            end
        end
        % Weigths normalization
        ps=normalized_weights(ps);

        %Save the filtered pdfs
        MU=[];
        MU.w=num2cell(ps,1);
        MU.mu=num2cell(md,1);
        for s=1:1:Mtt
            MU.Sigma{s}=vz(:,:,s);
        end
        F.measup{t} = MU;

        % TIME UPDATE STEP
        S=zeros(n,n,Mtt);
        m=zeros(n,Mtt);
        w=zeros(1,Mtt);
```

```
57          for ell=1:1:Mtt
58              k=ell;
59              w(k)=MU.w{ell};
60              m(:,k)=G.mm.A*MU.mu{ell}+G.mm.B*G.sim.ut{t};
61              S(:,:,k)=G.mm.A*MU.Sigma{ell}*G.mm.A'+G.mm.Q;
62          end
63
64          % Save the predicted pdf
65          TM=[];
66          TM.w=num2cell(w,1);
67          TM.mu=num2cell(m,1);
68          for s=1:1:Mtt
69              TM.Sigma{s}=S(:,:,s);
70          end
71          F.timeup{t+1}=TM;
72
73          % Gaussian Sum Reduction
74          if Mtt>=G.sim.Kv
75              [w,m,S]=gaussian_reduction(G.sim.minG,G.sim.maxG,0.5,w,m,S);
76          end
77          Mt_tm1=length(w);
78
79          % Feedback for the loop
80          TU=[];
81          TU.w=num2cell(w,1);
82          TU.mu=num2cell(m,1);
83          for s=1:1:Mt_tm1
84              TU.Sigma{s}=S(:,:,s);
85          end
86
87          % Save the filtered state and covariance
88          [x,P]=gmmTomuP(MU);
89          F.xtt{t}=x;
90          F.Stt{t}=P;
91
92          % Save the predicted state and covariance
93          [x,P]=gmmTomuP(TM);
94          F.xtmt{t+1}=x;
95          F.Stmt{t+1}=P;
96
97          clear ps md vz w m S;
98      end
99  end
```

Código 9.48 : Cómputo de $x_{t|t}$ y $P_{t|t}$ con GMM (MATLAB)

```
function [x,P]=gmmTomuP(Data)
    % Get the weigths
    alphai=Data.w;
    % Get the means
    mui=Data.mu;
    % Get the variances
    Pi=Data.Sigma;

    n=size(Pi{1},1);
    x=zeros(n,1);
    P=zeros(n,n);
    tam=length(alphai);
    if tam==1
        x=mui{1};
        P=Pi{1};
    else
        for i=1:1:tam
            x=x+alphai{i}*mui{i};
        end
        for i=1:1:tam
            P=P+alphai{i}*(Pi{i}+(mui{i}-x)*(mui{i}-x)');
        end
    end
end
```

Código 9.49 : Aproximación de $p(y_t|x_t)$ según Regla Gauss-Legendre (MATLAB)

```
function [yx] = model_pytxt(G)
q = gauss_legendre(G.sim.Kv);
wj=q.Weights;
xj=q.Points;

for t=1:1:G.sim.N
    at=G.sim.yt{t}-0.5*G.mm.Delta;
    bt=G.sim.yt{t}+0.5*G.mm.Delta;
    yx.muj{t}=-(at+bt)/2;
    yx.Xj{t}=((bt-at)/2)*xj;
    yx.betaj{t}=wj*(bt-at)/2;
end
```

Código 9.50 : Normalización de Pesos (MATLAB)

```
function [sm,lse] = normalized_weights(x)
if ~isvector(x), error('Input x must be a vector.'), end
n = length(x);
e = zeros(1,n);
[xmax,k] = max(x);
a = xmax;
s = 0;
for i = 1:n
    e(i) = exp(x(i)-xmax);
    if i ~= k
       s = s + e(i);
    end
end
if nargout > 1
   lse = a + log1p(s);
end
sm = e/(1+s);
```

Código 9.51 : Algoritmo Kullback-Leibler para Reducción de Gauss (MATLAB)

```
function [pso,med,var]=gaussian_reduction(LI,LS,LD,ps,md,vz)
% Kullback-Leibler reduction algorithm
% wills2017 - a bayesian filtering algorithm for gaussian mixture models

    N=length(ps);
    k=N;  B=Inf*ones(N,N);
    B = boundS(ps,md,vz,B);
    while (k>LS || (k>LI && min(min(B))<LD))
        [iaste,jaste]=find(B==min(min(B)));  iast=iaste(1); jast=jaste(1);
        [w,mu,P] = merged(ps(iast),ps(jast),md(:,iast),md(:,jast),vz(:,:,iast),vz(:,:,
    jast));
        ps(iast)=w;    md(:,iast)=mu;    vz(:,:,iast)=P;
        ps(jast)=[];   md(:,jast)=[];    vz(:,:,jast)=[];
        B(jast,:)=[];     B(:,jast)=[];
        B = bound(ps,md,vz,iast,B);
        k=k-1;
    end
    pso=ps;     med=md;    var=vz;
end

function [wij,muij,Pij] = merged(psi,psj,mdi,mdj,vzi,vzj)
```

```
    wij=psi+psj;
    wiIij = (psi)/(psi+psj);
    wjIij = (psj)/(psi+psj);
    muij = wiIij*mdi+wjIij*mdj;
    Pij = wiIij*vzi + wjIij*vzj + wiIij*wjIij*(mdi-mdj)*(mdi-mdj)';
end

function B = boundS(ps,md,vz,B)
    L=size(B,1);
    for i=1:1:L-1
        for j=i+1:1:L
            [wij,~,Pij] = merged(ps(i),ps(j),md(:,i),md(:,j),vz(:,:,i),vz(:,:,j));
            B(i,j)=0.5*(wij*log(det(Pij))-ps(i)*log(det(vz(:,:,i)))-ps(j)*log(det(vz
    (:,:,j))));
        end
    end
end

function B = bound(ps,md,vz,iast,B)
    N=size(B,1);
    for j=1:1:N
        if iast<j
            [wij,~,Pij] = merged(ps(iast),ps(j),md(:,iast),md(:,j),vz(:,:,iast),vz
    (:,:,j));
            B(iast,j)=0.5*(wij*log(det(Pij))-ps(iast)*log(det(vz(:,:,iast)))-ps(j)*log
    (det(vz(:,:,j))));
        end
    end
    jiast=iast;
    for i=1:1:N
        if jiast>i
            [wij,~,Pij] = merged(ps(i),ps(jiast),md(:,i),md(:,jiast),vz(:,:,i),vz(:,:,
    jiast));
            B(i,jiast)=0.5*(wij*log(det(Pij))-ps(i)*log(det(vz(:,:,i)))-ps(jiast)*log(
    det(vz(:,:,jiast))));
        end
    end
end
```

9.2. Códigos de Python

En esta sección se recopilan los códigos de Python utilizados para la evaluación de los métodos de estimación. Se separa en dos sub-secciones: Códigod publicados extraídos de repositorios en linea y códigos propios, escritos en base a la reproducción de estructuras empleadas en la implementaciones de MATLAB.

9.2.1. Códigos publicados

Los códigos en esta sección son utilizados para comparar el rendimiento de las implementaciones basadas en MATLAB y reemplazarlas en la etapa de evaluación en caso de presentar mejores propiedades. El filtro de Kalman y filtro de partículas corresponden la diseño del programador Gavin Gao, quien emplea estos métodos para estimar la posición de un objeto en base a mediciones aleatorias y resolver problemas de localización robótica.

Fuente: state_estimation, repositorio GitHub de usuario cggos (Gavin Gao).

Código 9.52 : Filtro de Kalman estándar (implementación de Gavin Gao)

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-
"""
@reference: Implementation of Kalman Filter with Python Language
"""

import time
from numpy import *
from numpy import dot, sum, tile
from numpy.linalg import inv, det
from numpy.random import randn
import pylab as pl


def kf_predict(X, P, A, Q, B, U):
    X = dot(A, X) + dot(B, U)
    P = dot(A, dot(P, A.T)) + Q
    return (X, P)


```

```
21 def kf_update(X, P, Y, H, R):
22     IM = dot(H, X)
23     IS = R + dot(H, dot(P, H.T))
24     K = dot(P, dot(H.T, inv(IS)))
25     X = X + dot(K, (Y-IM))
26     P = P - dot(K, dot(IS, K.T))
27     LH = gauss_pdf(Y, IM, IS)
28     return (X, P, K, IM, IS, LH)
29
30
31 def gauss_pdf(X, M, S):
32     if M.shape[1] == 1:
33         DX = X - tile(M, X.shape[1])
34         E = 0.5 * sum(DX * (dot(inv(S), DX)), axis=0)
35         E = E + 0.5 * M.shape[0] * log(2 * pi) + 0.5 * log(det(S))
36         P = exp(-E)
37     elif X.shape[1] == 1:
38         DX = tile(X, M.shape[1])- M
39         E = 0.5 * sum(DX * (dot(inv(S), DX)), axis=0)
40         E = E + 0.5 * M.shape[0] * log(2 * pi) + 0.5 * log(det(S))
41         P = exp(-E)
42     else:
43         DX = X-M
44         E = 0.5 * dot(DX.T, dot(inv(S), DX))
45         E = E + 0.5 * M.shape[0] * log(2 * pi) + 0.5 * log(det(S))
46         P = exp(-E)
47     return (P[0], E[0])
48
49
50 # time step of mobile movement
51 dt = 0.1
52
53 # Initialization of state matrices
54 X = array([[0.0], [0.0], [0.1], [0.1]])
55 P = diag((0.01, 0.01, 0.01, 0.01))
56 A = array([[1, 0, dt, 0], [0, 1, 0, dt], [0, 0, 1, 0], [0, 0, 0, 1]])
57
58 Q = eye(X.shape[0])
59 B = eye(X.shape[0])
60 U = zeros((X.shape[0], 1))
61
62 # Measurement matrices
63 Y = array([[X[0, 0] + abs(randn(1)[0])], [X[1, 0] + abs(randn(1)[0])]])
64 H = array([[1, 0, 0, 0], [0, 1, 0, 0]])
```

```
R = eye(Y.shape[0])

x1 = [X[0]]
x2 = [X[1]]
y1 = [Y[0]]
y2 = [Y[1]]

# Number of iterations in Kalman Filter
N_iter = 100

# Applying the Kalman Filter
for i in arange(0, N_iter):
    (X, P) = kf_predict(X, P, A, Q, B, U)
    (X, P, K, IM, IS, LH) = kf_update(X, P, Y, H, R)
    Y = array([[X[0, 0] + abs(0.1 * randn(1)[0])], [X[1, 0] + abs(0.1 * randn(1)[0])
    ]])
    x1.append(X[0])
    x2.append(X[1])
    y1.append(Y[0])
    y2.append(Y[1])


pl.plot(x1, x2)
pl.plot(y1, y2)
pl.show()
```

Código 9.53 : Filtro de partículas (implementación de Gavin Gao)

```
import numpy as np
import math
import matplotlib.pyplot as plt

# Estimation parameter of PF
Q = np.diag([0.1]) ** 2  # range error
R = np.diag([1.0, np.deg2rad(40.0)]) ** 2  # input error

#  Simulation parameter
Qsim = np.diag([0.2]) ** 2
Rsim = np.diag([1.0, np.deg2rad(30.0)]) ** 2

DT = 0.1  # time tick [s]
SIM_TIME = 50.0  # simulation time [s]
MAX_RANGE = 20.0  # maximum observation range
```

```
16
17  # Particle filter parameter
18  NP = 10  # Number of Particle
19  NTh = NP / 2.0  # Number of particle for re-sampling
20
21  def calc_input():
22      v = 1.0  # [m/s]
23      yawrate = 0.1  # [rad/s]
24      u = np.array([[v, yawrate]]).T
25      return u
26
27
28  def observation(xTrue, xd, u, RFID):
29      xTrue = motion_model(xTrue, u)
30
31      # add noise to gps x-y
32      z = np.zeros((0, 3))
33
34      for i in range(len(RFID[:, 0])):
35          dx = xTrue[0, 0] - RFID[i, 0]
36          dy = xTrue[1, 0] - RFID[i, 1]
37          d = math.sqrt(dx ** 2 + dy ** 2)
38          if d <= MAX_RANGE:
39              dn = d + np.random.randn() * Qsim[0, 0]  # add noise
40              zi = np.array([[dn, RFID[i, 0], RFID[i, 1]]])
41              z = np.vstack((z, zi))
42
43      # add noise to input
44      ud1 = u[0, 0] + np.random.randn() * Rsim[0, 0]
45      ud2 = u[1, 0] + np.random.randn() * Rsim[1, 1]
46      ud = np.array([[ud1, ud2]]).T
47
48      xd = motion_model(xd, ud)
49
50      return xTrue, z, xd, ud
51
52
53  def motion_model(x, u):
54      F = np.array([[1.0, 0, 0, 0],
55                    [0, 1.0, 0, 0],
56                    [0, 0, 1.0, 0],
57                    [0, 0, 0, 0]])
58
59      B = np.array([[DT * math.cos(x[2, 0]), 0],
```

```
                    [DT * math.sin(x[2, 0]), 0],
                    [0.0, DT],
                    [1.0, 0.0]])

    x = F.dot(x) + B.dot(u)

    return x


def gauss_likelihood(x, sigma):
    p = 1.0 / math.sqrt(2.0 * math.pi * sigma ** 2) * math.exp(-x ** 2 / (2 * sigma ** 2))
    return p


def calc_covariance(xEst, px, pw):
    cov = np.zeros((3, 3))
    for i in range(px.shape[1]):
        dx = (px[:, i] - xEst)[0:3]
        cov += pw[0, i] * dx.dot(dx.T)
    return cov


def pf_localization(px, pw, z, u):
    """
    Localization with Particle filter
    """
    for ip in range(NP):
        x = np.array([px[:, ip]]).T
        w = pw[0, ip]
        #  Predict with random input sampling
        ud1 = u[0, 0] + np.random.randn() * Rsim[0, 0]
        ud2 = u[1, 0] + np.random.randn() * Rsim[1, 1]
        ud = np.array([[ud1, ud2]]).T
        x = motion_model(x, ud)

        #  Calc Importance Weight
        for i in range(len(z[:, 0])):
            dx = x[0, 0] - z[i, 1]
            dy = x[1, 0] - z[i, 2]
            prez = math.sqrt(dx ** 2 + dy ** 2)
            dz = prez - z[i, 0]
            w = w * gauss_likelihood(dz, math.sqrt(Q[0, 0]))

```

```
103             px[:, ip] = x[:, 0]
104             pw[0, ip] = w
105
106         pw = pw / pw.sum()  # normalize
107
108         xEst = px.dot(pw.T)
109         PEst = calc_covariance(xEst, px, pw)
110
111         px, pw = resampling(px, pw)
112
113         return xEst, PEst, px, pw
114
115
116     def resampling(px, pw):
117         """
118         low variance re-sampling
119         """
120         Neff = 1.0 / (pw.dot(pw.T))[0, 0]  # Effective particle number
121         if Neff < NTh:
122             wcum = np.cumsum(pw)
123             base = np.cumsum(pw * 0.0 + 1 / NP) - 1 / NP
124             resampleid = base + np.random.rand(base.shape[0]) / NP
125
126             inds = []
127             ind = 0
128             for ip in range(NP):
129                 while resampleid[ip] > wcum[ind]:
130                     ind += 1
131                 inds.append(ind)
132
133             px = px[:, inds]
134             pw = np.zeros((1, NP)) + 1.0 / NP  # init weight
135
136         return px, pw
137
138
139     def main():
140         print(__file__ + " start!!")
141
142         time = 0.0
143
144         # RFID positions [x, y]
145         RFID = np.array([[10.0, 0.0],
146                          [10.0, 10.0],
```

```
147                         [0.0, 15.0],
148                         [-5.0, 20.0]])
149
150     # State Vector [x y yaw v]'
151     xEst = np.zeros((4, 1))
152     xTrue = np.zeros((4, 1))
153     PEst = np.eye(4)
154
155     px = np.zeros((4, NP))  # Particle store
156     pw = np.zeros((1, NP)) + 1.0 / NP  # Particle weight
157     xDR = np.zeros((4, 1))  # Dead reckoning
158
159     # history
160     hxEst = xEst
161     hxTrue = xTrue
162     hxDR = xTrue
163
164     for i in range(N):
165         u = calc_input()
166         xTrue, z, xDR, ud = observation(xTrue, xDR, u, RFID)
167         xEst, PEst, px, pw = pf_localization(px, pw, z, ud)
168         hxEst = np.hstack((hxEst, xEst))
169         hxDR = np.hstack((hxDR, xDR))
170         hxTrue = np.hstack((hxTrue, xTrue))
171
172
173 if __name__ == '__main__':
174     main()
```

9.2.2. Códigos basados en MATLAB

Aquí se recopilan los códigos propios, escritos al reproducir las estructuras y herramientas utilizadas en MATLAB con la semántica de Python y bibliotecas disponibles en línea.

El primer código contiene el proceso para modelar un sistema dinámico con salida cuantizada, donde sus propiedades y configuraciones dadas por el usuario del programa se almacenan en un objeto único denominado *GaussSystem*. Además se encuentras los comandos utilizados para calcular el error cuadrático medio entre el modelo y la estimación, el tiempo de ejecución de los algoritmos y la memoria asignada a cada proceso de interes, y para graficar los resultados.

Código 9.54 : Ejemplo de estimación de estados con salida cuantizada (Python)

```
#!/usr/bin/python3

import time
import tracemalloc
import numpy as np
import matplotlib.pyplot as plt
from gmmTomuP import *
from systematic_resample import *
from python_standard_kalman_filter import *
from python_quantized_kalman_filter import *
from python_particle_filter import *
from python_gaussian_sum_filter import *

class GaussSystem(object):
    def __init__(self, A = None, B = None, C = None, D = None, Q = None, R = None, N =
     None, Kv = None, minC = None, maxC = None, minG = None, maxG = None):

        if(A is None or C is None):
            raise ValueError("Set proper system dynamics.")
        self.n = A.shape[1]
        self.m = C.shape[1]
        self.A = A
        self.B = 0 if B is None else B
        self.C = C
        self.D = 0 if D is None else D
        self.Q = np.eye(self.n) if Q is None else Q
        self.R = np.eye(self.n) if R is None else R
        self.N = 100 if N is None else N
```

```
28          self.Kv = 10 if Kv is None else Kv
29          self.minC = 2 if minC is None else minC;
30          self.maxC = 2 if maxC is None else maxC;
31          self.minG = 1 if minG is None else minG;
32          self.maxG = 1 if maxG is None else maxG;
33
34
35  def example():
36      np.random.seed(0)
37      N = 100
38      Kv = 20
39
40      A = np.array([0.9]).reshape(1, 1)
41      B = np.array([1]).reshape(1, 1)
42      C = np.array([2]).reshape(1, 1)
43      D = np.array([0.5]).reshape(1, 1)
44      Q = np.array([1]).reshape(1, 1)
45      R = np.array([0.5]).reshape(1, 1)
46      G = GaussSystem(A, B, C, D, Q, R, N, Kv)
47
48      mu = np.array([1]).reshape(1,1)
49      Sigma = np.dot(0.01, Q)
50      w = np.array([1]).reshape(1,1)
51      Delta = 7
52
53      G.mu = mu
54      G.Sigma = Sigma
55      G.w = w
56      G.Delta = Delta
57
58      # Initial condition for KF
59      xin, Pin = gmmTomuP(G)
60      G.xin = xin
61      G.Pin = Pin
62
63      # Input of the System
64      ut = np.dot(np.sqrt(2), np.random.normal(0, 1, N))
65
66      # Model Noise
67      wt = np.dot(np.sqrt(Q), np.random.normal(0, 0.1, (A.shape[0],N)).reshape(A.shape
        [0],N))
68
69      # Measurements noise
70      vt = np.dot(np.sqrt(R), np.random.normal(0, 0.5, N).reshape(1,N))
```

```
71
72      # Simulated data
73      xt = np.linspace(0, 0, N+1)
74      xt[0] = xin
75      zt =  np.linspace(0, 0, N)
76      for i in range(N):
77          xt[i+1] = np.dot(A, xt[i]) + np.dot(B, ut[i]) + wt[0, i]
78          zt[i] = np.dot(C, xt[i]) + np.dot(D, ut[i]) + vt[0, i]
79      yt = Delta*np.around(np.dot(zt,1/Delta))
80
81      G.ut = ut
82      G.zt = zt
83      G.yt = yt
84
85      tracemalloc.start()
86
87      tracemalloc.clear_traces()
88      # Standard Kalman Filtering
89      # KFPrediction(G, xin, Pin)
90      KF_start = time.time()
91      KF = KFEstimation(G)
92      KF_end = time.time()
93      KF_size, KF_peak = tracemalloc.get_traced_memory()
94
95      tracemalloc.clear_traces()
96      # Quantized Kalman Filtering
97      # QKFPrediction(G, xin, Pin)
98      QKF_start = time.time()
99      QKF = QKFEstimation(G)
100     QKF_end = time.time()
101     QKF_size, QKF_peak = tracemalloc.get_traced_memory()
102
103     tracemalloc.clear_traces()
104     # Particle Filtering (Npar = 100)
105     # PFPrediction(G, Npar, xin, Pin)
106     PF_start = time.time()
107     PF = PFEstimation(G, 1000)
108     PF_end = time.time()
109     PF_size, PF_peak = tracemalloc.get_traced_memory()
110
111     tracemalloc.clear_traces()
112     # Gaussian Sum Filtering
113     # PFPrediction(G, xin, Pin)
114     GSF_start = time.time()
```

```
115     GSF = GSFEstimation(G)
116     GSF_end = time.time()
117     GSF_size, GSF_peak = tracemalloc.get_traced_memory()
118 
119     tracemalloc.stop()
120 
121     print('KF Time:', KF_end - KF_start, '- KF Accuracy:', (np.square(KF - xt[0:100]))
        .mean(axis=0))
122     print('QKF Time:', QKF_end - QKF_start, '- QKF Accuracy:', (np.square(QKF - xt
        [0:100])).mean(axis=0))
123     print('PF Time:', PF_end - PF_start, '- PF Accuracy:', (np.square(PF - xt[0:100]))
        .mean(axis=0))
124     print('GSF Time:', GSF_end - GSF_start, '- GSF Accuracy:', (np.square(GSF - xt
        [0:100])).mean(axis=0))
125     print('\nMax Memory Usage')
126     print('KF:', KF_peak, 'bytes')
127     print('QKF:', QKF_peak, 'bytes')
128     print('PF:', PF_peak, 'bytes')
129     print('GSF:', GSF_peak, 'bytes')
130 
131 
132     # Plot Results
133     plt.plot(range(len(zt)), G.zt, label = 'Model Output')
134     plt.plot(range(len(yt)), G.yt, '.', label = 'Measurements')
135     plt.xlabel("Discrete Time")
136     plt.ylabel("Data Value")
137     plt.grid()
138     plt.legend()
139     plt.show()
140 
141     plt.plot(range(len(xt)), xt, label = 'True State')
142     plt.plot(range(len(KF)), np.array(KF), label = 'KF Estimation')
143     plt.plot(range(len(QKF)), np.array(QKF), label = 'QKF Estimation')
144     plt.plot(range(len(PF)), np.array(PF), label = 'PF Estimation')
145     plt.plot(range(len(GSF)), np.array(GSF), label = 'GSF Estimation')
146     plt.xlabel("Discrete Time")
147     plt.ylabel("Data Value")
148     plt.grid()
149     plt.legend()
150     plt.show()
151 
152 if __name__ == '__main__':
153     example()
```

De este punto en adelante, los códigos presentados son análogos a sus respectivas implementaciones en MATLAB, pero con la sintaxis específica de Python.

Código 9.55 : Filtro de Kalman Estándar (Python)

```python
#!/usr/bin/python3

import numpy as np

class KalmanFilter(object):
    def __init__(self, G, xin = None, Pin = None):
        self.G = G
        self.K = 0
        self.xtmt = G.xin if xin is None else xin
        self.Stmt = G.Pin if Pin is None else Pin
        self.xtt = np.zeros((self.G.n, 1))
        self.Stt = np.eye(self.G.n)

    def MeasurementUpdate(self, t):
        # Kalman Gain
        S = self.G.R+np.dot(self.G.C, np.dot(self.Stmt, self.G.C.T))
        self.K = np.dot(np.dot(self.Stmt,self.G.C.T), np.linalg.inv(S))
        # Measurement Update
        y = self.G.yt[t] - np.dot(self.G.C, self.xtmt) - np.dot(self.G.D, self.G.ut[t
    ])
        self.xtt = self.xtmt + np.dot(self.K, y)
        self.Stt = np.dot(np.eye(self.G.m) - np.dot(self.K, self.G.C), self.Stmt)

    def TimeUpdate(self, t):
        # Time Update
        self.xtmt =  np.dot(self.G.A, self.xtt) + np.dot(self.G.B, self.G.ut[t])
        self.Stmt = self.G.Q +  np.dot(np.dot(self.G.A, self.Stt), self.G.A.T)
        return self.xtt

def KFEstimation(G, xin = None, Pin = None):
    # Kalman Filter (Standard)
    KF = KalmanFilter(G, xin, Pin)
    predictions = np.linspace(0, 0, G.N)
    for t in range(G.N):
        KF.MeasurementUpdate(t)
        predictions[t] = KF.TimeUpdate(t)
    return predictions
```

Código 9.56 : Filtro de Kalman Cuantizado (Python)

```python
#!/usr/bin/python3

import numpy as np

class QuantizedKalmanFilter(object):
    def __init__(self, G, xin = None, Pin = None):
        self.G = G
        self.K = 0
        self.xtmt = G.xin if xin is None else xin
        self.Stmt = G.Pin if Pin is None else Pin
        self.xtt = np.zeros((self.G.n, 1))
        self.Stt = np.eye(self.G.n)

    def MeasurementUpdate(self, t):
        # Kalman Gain
        S = self.G.R+np.dot(self.G.C, np.dot(self.Stmt, self.G.C.T))
        self.K = np.dot(np.dot(self.Stmt,self.G.C.T), np.linalg.inv(S))
        # Measurement Update
        y = self.G.yt[t] - self.Quantization(np.dot(self.G.C, self.xtmt) + np.dot(self.G.D, self.G.ut[t]))
        self.xtt = self.xtmt + np.dot(self.K, y)
        self.Stt = np.dot(np.eye(self.G.m) - np.dot(self.K, self.G.C), self.Stmt)

    def TimeUpdate(self, t):
        # Time Update
        self.xtmt =  np.dot(self.G.A, self.xtt) + np.dot(self.G.B, self.G.ut[t])
        self.Stmt = self.G.Q +  np.dot(np.dot(self.G.A, self.Stt), self.G.A.T)
        return self.xtt

    def Quantization(self, zt):
        y = self.G.Delta*np.around(zt/self.G.Delta)
        return y

def QKFEstimation(G, xin = None, Pin = None):
    # Kalman Filter (Quantized)
    QKF = QuantizedKalmanFilter(G, xin, Pin)
    predictions = np.linspace(0, 0, G.N)
    for t in range(G.N):
        QKF.MeasurementUpdate(t)
        predictions[t] = QKF.TimeUpdate(t)
    return predictions
```

Código 9.57 : Filtro de Partículas MCMC (Python)

```python
#!/usr/bin/python3

import numpy as np
from numpy.matlib import repmat as repmat
from scipy.linalg import cholesky as chol
from scipy.stats import multivariate_normal as mvn
from systematic_resample import *


class ParticleFilter(object):
    def __init__(self, G, Npar = None, xin = None, Pin = None):

        self.G = G

        self.xin = G.xin if xin is None else xin
        self.Pin = G.Pin if Pin is None else Pin

        self.Npar = G.N if Npar is None else Npar
        self.xtt = np.zeros((self.G.n, 1))
        self.xpredN = np.zeros((self.G.n, 1))
        self.xpres = []

        self.xpar = (repmat(self.xin, 1, self.Npar) + np.dot(chol(self.Pin), np.random.normal(0, 1, (self.G.n,self.Npar))))

        self.a = self.G.yt - self.G.Delta/2
        self.b = self.G.yt + self.G.Delta/2

    def ParticleUpdate(self, t):
        lim_a = repmat(self.a[t], 1, self.Npar) - np.dot(self.G.C, self.xpar) - np.dot(self.G.D, self.G.ut[t])
        lim_b = repmat(self.b[t], 1, self.Npar) - np.dot(self.G.C, self.xpar) - np.dot(self.G.D, self.G.ut[t])

        w = (mvn.cdf(lim_b.T, mean = 0., cov = self.G.R) - mvn.cdf(lim_a.T, mean = 0., cov = self.G.R)).reshape(1,self.Npar)
        if np.sum(w) == 0:
            raise ValueError("Not proper CDF calculation.")
        w = np.dot(w, 1/np.sum(w))

        self.xpar = systematic_resample(w, self.xpar)
```

```
38          w = np.dot(1/self.Npar, np.ones((1,self.Npar)))
39
40          self.xpres.append(self.xpar)
41          self.xtt = np.sum(w*self.xpar, axis = 1)
42
43          self.xpar = (np.dot(self.G.A, self.xpar) + np.dot(self.G.B, self.G.ut[t]) + np
        .dot(chol(self.G.Q), np.random.normal(0, 1, (self.G.n, self.Npar))))
44
45          return self.xtt
46
47  def PFEstimation(G, Npar = None, xin = None, Pin = None):
48
49      # Particle Filter
50      PF = ParticleFilter(G, Npar, xin, Pin)
51      predictions = np.linspace(0, 0, G.N)
52
53      for t in range(G.N):
54          predictions[t] = PF.ParticleUpdate(t)
55
56      PF.xpredN = PF.xpar
57
58      return predictions
```

Código 9.58 : Función de Resampling Sistemático (Python)

```
1   #!/usr/bin/python3
2
3   import numpy as np
4
5   def systematic_resample(w, par):
6       n = 0
7       m = 0
8       M = w.shape[1]
9       index = np.zeros((1, M))
10      xpar = np.zeros((1, M))
11      Q = np.cumsum(w)
12      u0 = np.random.uniform(0,1,(1,1))/M
13
14      while n < M:
15          u = u0 + n/M
16
17          while Q[m] < u:
18              m = m+1
```

```
19
20          index[0, n]= m
21          n = n+1
22
23      for i in range(M):
24          xpar[0, i] = par[0, int(index[0,i])]
25
26      return xpar
```

Código 9.59 : Filtro Suma de Gaussianas (Python)

```
1  #!/usr/bin/python3
2
3  import numpy as np
4  from gmmTomuP import *
5  from model_pytxt import *
6  from normalized_weights import *
7  from gaussian_reduction import *
8  from numpy.linalg import solve as solve
9  from numpy.linalg import det as det
10
11
12 class GaussianSumFilter(object):
13     def __init__(self, G, xin = None, Pin = None):
14
15         self.G = G
16         self.TU = self.TimeUp(G)
17         self.MU = self.TimeUp()
18         self.TM = self.TimeUp()
19         x, P = gmmTomuP(G)
20
21         self.timeup = [self.TU]
22         self.xtmt = x
23         self.Stmt = P
24         self.measup = []
25         self.xtt = np.zeros((self.G.n, 1))
26         self.Stt = np.eye(self.G.n)
27
28         self.n = G.A.shape[0]
29         self.Mt_tm1 = self.TU.w.shape[1]
30         self.Mtt = self.G.Kv*self.Mt_tm1
31         self.yx = model_pytxt(G)
32
```

```
33      def MeasurementUpdate(self, t):
34
35          # Computing p(yt|xt)
36          muj = self.yx.muj[t]
37          Xj = self.yx.Xj[t];
38          betaj = self.yx.betaj[t]
39
40          # MEASUREMENT UPDATE STEP
41          ell = 0
42          self.Mtt = self.G.Kv*self.Mt_tm1
43          vz = np.zeros((self.Mtt, self.n, self.n))
44          md = np.zeros((self.n, self.Mtt))
45          ps = np.zeros((1, self.Mtt))
46
47          for k in range(self.Mt_tm1):
48              for j in range(self.G.Kv):
49                  my = Xj[j] - np.dot(self.G.C,self.TU.mu[:, k]) - np.dot(self.G.D,self.
        G.ut[t]) - muj
50                  Zy = self.G.R + np.dot(self.G.C,np.dot(self.TU.Sigma[k],self.G.C.T))
51                  K = np.dot(self.TU.Sigma[k],self.G.C.T) / Zy
52                  vz[ell] = np.dot(np.eye(self.n) - np.dot(K,self.G.C),self.TU.Sigma[k])
53                  md[:, ell] = self.TU.mu[:, k] + np.dot(K,my)
54                  ps[0, ell] = np.log(betaj[j]) + np.log(self.TU.w[0, k]) - 0.5*(np.dot(
        my.T, solve(Zy,my)) + np.log(det(2*np.pi*Zy)))
55                  ell = ell + 1
56          # Weigths normalization
57          ps = normalized_weights(ps)
58
59          # Save the filtered pdfs
60          self.MU.w = ps
61          self.MU.mu = md
62          self.MU.Sigma = vz
63          self.measup.append(self.MU)
64
65
66      def TimeUpdate(self, t):
67
68          # TIME UPDATE STEP
69          S = np.zeros((self.Mtt,s elf.n, self.n))
70          m = np.zeros((self.n, self.Mtt))
71          w = np.zeros((1, self.Mtt))
72
73          for k in range(self.Mtt):
74              w[0, k] = self.MU.w[0,k]
```

```
75              m[:, k] = np.dot(self.G.A,self.MU.mu[:,k]) + np.dot(self.G.B,self.G.ut[t])
76              S[k] = np.dot(self.G.A,np.dot(self.MU.Sigma[k],self.G.A.T)) + self.G.Q
77
78          # Save the predicted pdf
79          self.TM.w = w
80          self.TM.mu = m
81          self.TM.Sigma = S
82          self.timeup.append(self.TM)
83
84          # Gaussian Sum Reduction
85          if self.Mtt >= self.G.Kv:
86              w, m, S = gaussian_reduction(self.G.minG, self.G.maxG, 0.5, w, m, S)
87          self.Mt_tm1 = w.shape[1]
88
89          # Feedback for the loop
90          self.TU.w = w
91          self.TU.mu = m
92          self.TU.Sigma = S
93
94          # Save the filtered state and covariance
95          x, P = gmmTomuP(self.MU)
96          self.xtt = x
97          self.Stt = P
98
99          # Save the predicted state and covariance
100         x, P = gmmTomuP(self.TM)
101         self.xtmt = x
102         self.Stmt = P
103
104         return self.xtt
105
106     class TimeUp:
107         def __init__(self, G = None):
108
109             self.w = [] if G is None else G.w
110             self.mu = [] if G is None else G.mu
111             self.Sigma = [] if G is None else G.Sigma
112
113
114 def GSFEstimation(G, xin = None, Pin = None):
115
116     # Gaussian Sum Filter
117     GSF = GaussianSumFilter(G)
118     predictions = np.linspace(0, 0, G.N)
```

```
119
120     for t in range(G.N):
121         GSF.MeasurementUpdate(t)
122         predictions[t] = GSF.TimeUpdate(t)
123
124     return predictions
```

Código 9.60 : Aproximación de $p(y_t|x_t)$ según Regla Gauss-Legendre (Python)

```
1  #!/usr/bin/python3
2
3  import numpy as np
4  from numpy.polynomial.legendre import leggauss as GaussLegrende
5
6  class Model:
7      def __init__(self):
8
9          self.muj = []
10         self.Xj = []
11         self.betaj = []
12
13 def model_pytxt(G):
14
15     xj, wj = GaussLegrende(G.Kv)
16     xj = np.array(xj).reshape(G.Kv, 1)
17     wj = np.array(wj).reshape(G.Kv, 1)
18     yx = Model()
19
20     for t in range(G.N):
21         at = G.yt[t] - np.dot(0.5, G.Delta)
22         bt = G.yt[t] + np.dot(0.5, G.Delta)
23
24         yx.muj.append((-at - bt)/2)
25         yx.Xj.append(np.dot((bt-at)*0.5, xj))
26         yx.betaj.append(np.dot(wj, (bt-at)*0.5))
27
28     return yx
```

Código 9.61 : Normalización de Pesos (Python)

```python
#!/usr/bin/python3

import numpy as np

def normalized_weights(x):

    n = len(x[0])
    e = np.zeros((1, n))
    a = np.max(x)
    k = np.argmax(x)
    s = 0

    for i in range(n):
        e[0,i] = np.exp(x[0,i] - a)
        if i != k:
            s = s + e[0,i]

    return e/(1 + s)
```

Código 9.62 : Algortimo Kullback-Leibler para Reducción de Gauss (Python)

```python
#!/usr/bin/python3

import numpy as np
from numpy.linalg import det as det

def gaussian_reduction(LI, LS, LD, ps, md, vz):

    N = ps.shape[1]
    k = N
    B = np.dot(np.Inf, np.ones((N,N)))
    B = boundS(ps, md, vz, B)
    while (k > LS or (k > LI and np.min(B) < LD)):
        ijaste = np.argwhere(B == np.min(B))
        iast = ijaste[0,0]
        jast = ijaste[0,1]
        w,mu,P = merged(ps[0,iast],ps[0,jast],md[:,iast],md[:,jast], vz[iast],vz[jast
    ])
        ps[0,iast] = w
        md[:,iast] = mu
        vz[iast] = P
```

```
20          ps = np.delete(ps,jast,1)
21          md = np.delete(md,jast,1)
22          vz = np.delete(vz,jast,0)
23          B = np.delete(B,jast,0)
24          B = np.delete(B,jast,1)
25          B = bound(ps,md,vz,iast,B)
26          k = k - 1
27
28      pso = ps
29      med = md
30      var = vz
31      return pso, med, var
32
33  def bound(ps, md, vz, iast, B):
34      N = B.shape[0]
35      for j in range(N):
36          if iast < j:
37              wij,muij,Pij = merged(ps[0,iast],ps[0,j],md[:,iast],md[:,j],vz[iast],vz[j
        ])
38              B[iast,j] = 0.5*(np.dot(wij, np.log(det(Pij))) - np.dot(ps[0, iast], np.
        log(det(vz[iast]))) - np.dot(ps[0, j], np.log(det(vz[j]))))
39      jiast = iast
40      for i in range(N):
41          if jiast > i:
42              wij,muij,Pij = merged(ps[0,i],ps[0,jiast],md[:,i],md[:,jiast],vz[i],vz[
        jiast])
43              B[i,jiast] = 0.5*(np.dot(wij,np.log(det(Pij))) - np.dot(ps[0,i],np.log(det
        (vz[i]))) - np.dot(ps[0,jiast],np.log(det(vz[jiast]))))
44
45      return B
46
47  def boundS(ps, md, vz, B):
48      L = B.shape[0]
49      for i in range(L-1):
50          for j in range(i+1, L):
51              wij,muij,Pij = merged(ps[0,i],ps[0,j],md[:,i],md[:,j],vz[i],vz[j])
52              B[i,j] = 0.5*(np.dot(wij,np.log(det(Pij))) - np.dot(ps[0,i],np.log(det(vz[
        i]))) - np.dot(ps[0,j],np.log(det(vz[j]))))
53
54      return B
55
56  def merged(psi,psj,mdi,mdj,vzi,vzj):
57      wij = psi + psj
58      wiIij = psi/(psi + psj)
```

```
59      wjIij = psj/(psi + psj)
60      muij = np.dot(wiIij,mdi) + np.dot(wjIij,mdj)
61      Pij = np.dot(wiIij,vzi) + np.dot(wjIij,vzj) + np.dot(np.dot(wiIij,wjIij),np.dot(
        mdi - mdj,(mdi - mdj).T))
62
63      return wij, muij, Pij
```

Referencias

[1] Arasaratnam, I., Haykin, S., and Elliott, R. J. (2007). Discrete-time nonlinear filtering algorithms using gauss–hermite quadrature. Proceedings of the IEEE, 95(5):953–977.

[2] Cedeño, A. L., Carvajal, R., & Agüero, J. C. (2021). A Novel Filtering Method for Hammerstein-Wiener State-Space Systems. 2021 IEEE CHILEAN Conference on Electrical, Electronics Engineering, Information and Communication Technologies (CHILECON), 1–7.

[3] Cedeño, A. L., Albornoz, R., Carvajal, R., Godoy, B. I., & Agüero, J. C. Agüero. (2021). A Two-Filter Approach for State Estimation Utilizing Quantized Output Data. Sensors, 21, 7675.

[4] Cedeño, A. L., Albornoz, R., Carvajal, R., Godoy, B. I., & Agüero, J. C. (2021). On Filtering Methods for State-Space Systems having Binary Output Measurements. IFAC-PapersOnLine, 54(7), 815–820.

[5] Cedeño, A. L., Albornoz, R., Carvajal, R., Godoy, B. I., & Agüero, J. C. (2023). On Filtering and Smoothing Algorithms for Linear State-Space Models Having Quantized Output Data. Mathematics, 11(6), 1327.

[6] Cohen, H. (2011), Numerical Approximation Methods, Springer: Berlin/Heidelberg, Germany.

[7] Del Moral, Pierre (1996). Non Linear Filtering: Interacting Particle Solution. Markov Processes and Related Fields. 2 (4): 555–580.

[8] Doucet, A., Godsill, S., and Andrieu, C. (2000). On sequential Monte Carlo sampling methods for Bayesian filtering. Statistics and computing, 10(3):197–208.

[9] Doucet, A., De Freitas, N., Gordon, N. (2001). Sequential Monte Carlo methods in practice. Springer Science & Business Media.

[10] Friedland, B. (2012). Control system design: an introduction to state-space methods. Courier Corporation, North Chelmsford, Massachusetts.

[11] Funk, C., Noack, B. y Hanebeck, U. D. (2021). Conservative Quantization of Covariance Matrices with Applications to Decentralized Information Fusion. Sensors Journal, 21(9), 3059.

[12] Gagniuc, Paul A. (2017). Markov Chains: From Theory to Implementation and Experimentation. USA, NJ: John Wiley & Sons. pp. 1–235.

[13] Gelb A., Kasper J., Nash, R., Price C. and Sutherland, A. (1974). Applied optimal estimation. Cambridge, MA: MIT Press.

[14] Gersho, A. and Gray, R. M. (2012). Vector Quantization and Signal Compression, volume 159. Springer Science & Business Media.

[15] Grewal, S. Mohinder and Andrews Angus. (2010). Applications of Kalman Filtering in Aerospace 1960 to the Present. IEEE Control Systems Magazine, Volume: 30, Issue: 3.

[16] Gómez, J. C. and Sad, G. D. (2020). A State Observer from Multilevel Quantized Outputs. Argentine Conference on Automatic Control (AADECA), pages 1–6.

[17] Gordon, N. J., Salmond, D. J., and Smith, A. F. M. (1993). Novel approach to nonlinear/non-gaussian bayesian state estimation. IEE Proceedings F - Radar and Signal Processing, 140(2):107–113.

[18] Grewal, M. S. and Andrews, A. P. (2014). Kalman filtering: Theory and Practice with MATLAB. Wiley-IEEE Press.

[19] Jerker Nordh. (2017) pyParticleEst: A Python Framework for Particle-Based Estimation Methods. (Technical Reports TFRT-7628). Department of Automatic Control, Lund Institute of Technology, Lund University.

[20] Kalman, R. E. (1960). A New Approach to Linear Filtering and Prediction Problems. Journal of Fluids Engineering, Transactions of the ASME, 82(1):35–45.

[21] Laaraiedh, Mohamed. (2012). Implementation of Kalman Filter with Python Language. IETR Labs, University of Rennes.

[22] Leong, A. S., Dey, S., and Nair, G. N. (2013). Quantized Filtering Schemes for Multi-Sensor Linear State Estimation: Stability and Performance Under High Rate Quantization. IEEE Transactions on Signal Processing, 61(15):3852–3865.

[23] Liu, Jun S., Chen, Rong (1998). Sequential Monte Carlo Methods for Dynamic Systems. Journal of the American Statistical Association. 93 (443): 1032–1044.

[24] Hostettler, R. (2015). A two filter particle smoother for wiener state-space systems. IEEE Conference on Control Applications (CCA), pages 412–417. IEEE.

[25] Kitagawa, G. (1996). Monte carlo filter and smoother for non-Gaussian nonlinear state space models. Journal of Computational and Graphical Statistics. 5 (1): 1–25.

[26] Kitagawa, G. (1994). The two-filter formula for smoothing and an implementation of the Gaussian-sum smoother. Ann. Inst. Stat. Math. 46, 605–623.

[27] M. S. Arulampalam, S. Maskell, N. Gordon, and T. Clapp. (2002). A tutorial on particle filters for online nonlinear/non-Gaussian Bayesian tracking. IEEE Trans. on Signal Processing, vol. 50(2): 174–188.

[28] Runnalls, A.R. (2007). Kullback-Leibler approach to Gaussian mixture reduction. IEEE Trans. Aerosp. Electron. Syst. 43, 989–999.

[29] Salgado, Mario E., Yuz, Juan I. y Rojas, Ricardo A. (2014). Análisis de Sistemas Lineales. Departamento de Electrónica, Universidad Técnica Federico Santa María, Valparaíso, Chile.

[30] Särkkä, S. (2013). Bayesian filtering and smoothing, volume 3. Cambridge University Press.

[31] Singh, A. K. and Bhaumik, S. (2015). Higher degree cubature quadrature kalman filter. International Journal of Control, Automation and Systems, 13(5):1097–1105.

[32] Sorenson, H. W. and Alspach, D. L. (1971). Recursive Bayesian Estimation using Gaussian Sums. Automatica, vol.7, issue 4, pg. 465-479.

[33] T. Li, M. Bolic, P. M. Djuric. (2015). Resampling Methods for Particle Filtering: Classification, implementation, and strategies. IEEE Signal Processing Magazine, vol. 32, no. 3, pp. 70–86.

[34] Valera Fernández, Ángel. (2016). Modelado y Control en el Espacio de Estados. 1º Edición, Editorial: Universidad Politécnica de Valencia.

[35] Wen, C., Wang, Z., Liu, Q., and Alsaadi, F. E. (2018). Recursive Distributed Filtering for a Class of State- Saturated Systems With Fading Measurements and Quantization Effects. IEEE Transactions on Systems, Man, and Cybernetics: Systems, 48(6):930–941.

[36] Welch, G. and Bishop, G. (2001). An Introduction to the Kalman Filter. Department of Computer Science, University of North Carolina, NC 27599-3175.

[37] Y. Zhai, M. Yeary. (2004). Implementing particle filters with Metropolis-Hastings algorithms. Annual Technical and Leadership Workshop, pp. 149–152.