Tecnológico de Costa Rica

Escuela de Ingeniería Electrónica

Caracterización de algoritmos para la predicción de la localización en sistemas empotrados

Documento de tesis sometido a consideración para optar por el grado académico de

Maestría en Electrónica con Énfasis en Sistemas Empotrados

Pedro Elías Alpízar Salas

Cartago Agosto, 2017

Declaro que la presente Tesis de Maestría ha sido realizado enteramente por mi persona, utilizando y aplicando literatura referente al tema e introduciendo conocimientos propios.

En los casos en que he utilizado bibliografía, procedo a indicar las fuentes mediante las respectivas referencia. En consecuencia, asumo la responsabilidad total por el trabajo de investigación realizado y por el contenido del correspondiente documento final.

Pedro Elías Alpízar S.

Pedro Elías Alpízar Salas

Cartago, 13 de agosto de 2017

Céd: 1-1360-0984

Instituto Tecnológico de Costa Rica

Escuela de Ingeniería Electrónica

Tesis de Maestría

Tribunal evaluador

Tesis de maestría defendida ante el presente Tribunal Evaluador como requisito para optar por el grado académico de maestría, del Instituto Tecnológico de Costa Rica.

Miembros del Tribunal

Máster Jeferson González Gómez

Profesor lector

Máster Yeiner Arias Esquivel

Profesor lector

Máster Carlos Adrián Salazar García

Director de Tesis

Los miembros de este Tribunal dan fe de que la presente tesis ha sido aprobada y cumple con las normas establecidas por la Escuela de Ingeniería Electrónica.

Cartago, 14 de agosto de 2017

Resumen

Caracterizar el desempeño de diversos algoritmos para la predicción de la ubicación en sistemas empotrados de bajo costo. La presente investigación tiene como objetivo caracterizar el desempeño de diversos algoritmos para la predicción de la ubicación en sistemas empotrados de bajo costo utilizados para mejorar la precisión de las lecturas de sensores de posicionamiento global. Estos algoritmos son necesarios para minimizar los errores generados por los cambios en la calidad de la señal de los satélites.

El análisis de los algoritmos se enfocará en determinar los siguientes factores:

- Eficacia del algoritmo para reducir el error en la lectura de los sensores
- Ajuste del algoritmo para un sistema empotrado de bajo costo

Para medir la eficacia del los algoritmos se utiliza la diferencia absoluta de la posición leída del GPS, contra la calculada para cada instante del recorrido. En cuanto al análisis de que tan apto es cada algoritmo para su uso en un sistema empotrado de bajo costo se analizan la utilización del CPU, tiempo de respuesta y uso de memoria.

Se optimizan los algoritmos mediante la división de los datos por dimensión para reducir el desperdicio de recursos y aislar los ejes para ser implementados de acuerdo con las necesidades de futuros proyectos empotrados. Además, se reduce la carga computacional mediante el uso de modelos matemáticos aproximados para el desplazamiento tal como el movimiento rectilíneo uniformemente acelerado. Así mismo, se define además una interfaz común para que los algoritmos puedan ser comparados de manera correcta y nuevos algoritmos puedan ser estudiados más adelante.

Se obtiene una aplicación empotrada que utilizando el fitro de extendido de Kalman logra tener el error por debajo de 1345.93 metros de error para cinco segundos de falla. Este cálculo se realiza en menos de 118μs en un Raspberry Pi 2 con una imagen de tamaño mínimo.

La misma plataforma logra entrenar 100 iteraciones de tres perceptrones multi capa con 34 neuronas cada uno en un tiempo de 5.45ms. Posterior al entrenamiento se somete el algoritmo a la predicción de la posición, consiguiendo una respuesta en un tiempo no mayor a 250μs. La error de la posición calculada promedio para una falla de 5s fue de 73.32 metros.

Palabras clave: GPS, ARMANN, predicción de localización, perfilado de algoritmos, optimización para sistemas empotrados, EKF, Kalman, ANN

Abstract

The main objective of this thesis is to characterize múltiple algorithms for location prediction to improve the precision of global positioning sensors readings. This algorithms are needed to compensate bad signal quality of the necessary satellites.

The analysis will be focus on:

- Effectiveness of the algorithm on reducing the reading error on the positioning sensors
- Fit-ability of the algorithm for a embedded system

In order to measure the effectiveness of the implemented methods the absolute difference between the real GPS and the calculated value for each instant of the track. In order the get the fit-ability for embedded system of each algorithm CPU usage, response time, and memory usage is determine for them.

Every method is optimize by dividing the data per dimension, in order to decrease the resources foot-print of the program to the minimum, and so they could scale correctly for different requirements. Also the CPU load is reduced by implementing approximated mathematical models like the uniformly accelerated motion when possible. Lastly an interface is defined to isolate the algorithm from the main program, this way there is a clean comparison between algorithms and new ones could be study later.

Using Extended Kalman filter the embedded application created was able to keep the error below 1345.93 meters for five seconds GPS failure. Each of this predictions has taken less than 118μs on a Raspberry Pi 2 with a custom OS.

The same platform is able to train, over 100 times on the same data, three Multi-Layer perceptron with 34 neurons each under 5.45ms. Once trained the MLP is able to predict within 250μs a position with an average error of 73.32 meters, all this in a five seconds failure of the GPS

Keywords: GPS, ARMANN, location prediction, algorithm profiling, embedded system optimization, ANN, EKF

a mi esposa

Agradecimientos

El presente trabajo existe gracias al impulso de mi querida esposa y a la oportunidad brindada por el profesor Carlos Salazar.

Pedro Elías Alpízar Salas

Cartago, 22 de agosto de 2017

Índice general

Índice de figuras

Índice de tablas

Lista de símbolos y abreviaciones

Abreviaciones

ANN	Red Neuronal Artificial
ARMANN	Red Neuronal Artificial Auto-Regresiva
BSP	Paquete para Soporte de Plataforma o Capa de Plataforma
CEKF	Filtro Extendido Correlacionado de Kalman
CNN	Redes Neuronales de Construcción
DD1	Diferencias Divididas en primer orden
DD2	Diferencias Divididas en segundo orden
DGPS	Sistema de Posicionamiento Global Diferencial
EGNOS	Sistema Europeo Geoestacionario de Sobre-capa para Navegación
EKF	Filtro Extendido de Kalman
ENN	Red Neuronal Artificial Evolutiva
GPS	Sistema de Posicionamiento Global
IMU	Unidad de Medición Inercial
INS	Sistema de Navegación Inercial
KF	Filtro de Kalman
MEMS	Sistemas Micro-Electromecánicos
MLP	Preceptrón Multiple Capa
MRUA	Movimiento Rectilíneo Uniformemente Acelerado
OS	Sistema Operativo
RBF	Redes Neuronales con Función de Base Radial
RNN	Red Neuronal Artificial Recurrente
SBAS	Sistema Aumentado Basado en Satélite
SVM	Máquina de Vectores de Soporte
UAV	Vehículo Aéreo no Tripulado

Notación general

$\mathbf{A}$	Matriz $\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{bmatrix}$
$\mathbb{R}$	Conjunto de los números reales
$\underline{\mathbf{x}}$	Vector

$\underline{\mathbf{x}} = [x_1\ x_2\ \ldots\ x_n]^T = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$

y Escalar

Capítulo 1

Introducción

1.1 Posicionamiento de sistemas embebidos

Conocer la posición de un objeto es un problema que se puede resolver utilizando varios enfoques. Para espacios interiores la solución usualmente proviene de la lectura de distancias o el reconocimiento de objetos. En el caso de aplicaciones exteriores la solución más implementada es la utilización de los sistemas de posicionamiento global (GPS por sus siglas en inglés). Un receptor GPS utiliza señales enviadas de cuatro o más satélites para identificar su posición en la faz de la tierra. En la Figura 1.1 se muestra cómo el receptor lee la posición y distancia entre al menos cuatro satélites.

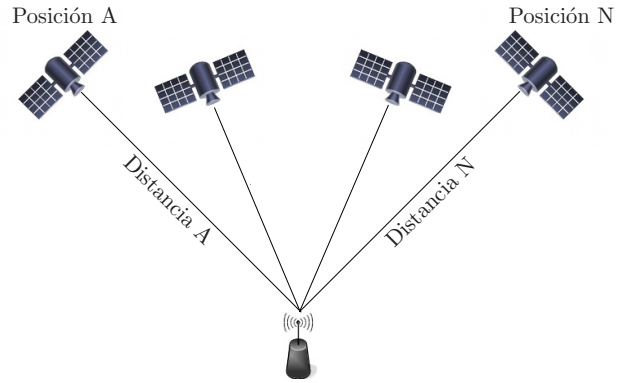


Figura 1.1: Funcionamiento conceptual de un receptor GPS

La posición exacta del satélite A hasta el satélite N, es transmitida al receptor donde el mismo calcula la distancia hacia cada una de ellas como se puede entender de la Figura 1.1 y se explica en [1]. La medición de la distancia se determina mediante el tiempo que toma en llegar la señal desde el satélite hasta el receptor. Los rebotes de la señal pueden incrementar ese tiempo llevando a errores en la medición.

Ninguna solución es perfecta, para el caso del posicionamiento mediante GPS depende de la recepción de la señal de al menos cuatro satélites de manera constante. En caso de disminución en la calidad de la señal (corrupción o pérdida total) la posición del objeto

puede tener un error mayor al esperado, o no podrá ser calculada del todo. Estos espacios vacíos en la confianza del aparato con respecto a su posición evita que pueda ser utilizado durante ese momento, y los mismos deben ser llenados de alguna manera.

La predicción de la posición para receptores GPS es un amplio campo de investigación que abarca desde vehículos no tripulados [2], botes de remos [3] o inclusive rastreo de animales [4], pero todos tienen una misma finalidad conseguir: la posición del objeto lo más certera posible y eliminar los vacíos de confianza generados por los problemas de la señal en el sensor de GPS.

En el área de investigación se han utilizado enfoques matemáticos ([2]), fusión de sensores ([5], [6], [7], entre otros) y hasta redes neuronales ([8]) para mejorar la presición del GPS y evitar que el sistema se quede en algún momento sin datos. Los estudios realizados en el área muestran cómo los diversos algoritmos aumentan la precisión de los datos en momentos sin señal de los satélites, la cantidad de tiempo sin señal o cómo se puede reducir la potencia de consumo mediante la reducción de muestras requeridas al GPS. Todas las investigaciones que se han enfocado en la precisión de la posición fueron realizadas sobre sistemas de alto costo, con alto poder computacional, y alta precisión en el receptor GPS.

La presente investigación pretende indagar cuáles son los efectos de los diversos algoritmos para incrementar la precisión de la posición de los sistemas empotrados de bajo consumo y costo. La misma forma parte de una serie de investigaciones independientes realizadas por estudiantes y profesores de la Escuela de Electrónica del Tecnológico de Costa Rica (TEC) que buscan entender cuales son los retos y los problemas que se presentan durante la lectura de la posición en sistemas empotrados.

1.2 Descripción del problema

La reducción de costo en sistemas empotrados acarrea implicaciones sobre el desempeño del sistema así como las capacidades que puede tener el mismo luego de la reducción de costo. Por ejemplo, los procesadores de menor costo tienen un menor desempeño, los sensores con menor costo tienen capacidades reducidas en precisión y frecuencia de las medidas, la reducción de costo en las baterías implica una reducción del tiempo de uso del sistema. Estos detalles implican que el comportamiento de algunos algoritmos podría cambiar entre plataformas de alto rendimiento y sistemas empotrados de alto costo.

Las investigaciones actuales como [8], [5], [9], y [10] utilizan plataformas de alto desempeño conectadas a los vehículos sobre los cuales está trabajando. En [11] y [8] incluso utilizan Sistemas de Posicionamiento Global Diferencial (DGPS por sus siglas en inglés) los cuales tienen un costo elevado, debido al cual se utilizan usualmente en aplicaciones militares.

Un ejemplo de costo de plataformas similares a las utilizadas en investigaciones como [8], [9] y [12] se desgloza a continuación:

- Sensor DGPS: sin antenas $595, kit $1 995 encontrados en [13]
- Sensor INS comercial: $800 encontrado en [14]

- CPU escritorio: Core 2 Duo (2008) $9.5 encontrado en [15]

Esto suma un costo total de $2 804.5 en únicamente tres componentes de la plataforma sin tomar encuenta donde los mismos se encuentren instalados, ni la forma de alimentación para dichos elementos dentro de un vehículo en movimiento.

La plataformas de alto rendimiento utilizadas hasta ahora para las investigaciones de predicción de la posición, así como la falta de detalles sobre la carga que dichos algoritmos ejercen sobre el sistema hacen muy difícil a los investigadores determinar cuál implementación se ajusta mejor a los problemas que buscan resolver utilizando plataformas de bajo costo.

La presente tesis pretende llevar el sistema a una plataforma con un sensor GPS sin antena externa, un IMU comercial para robótica de 9 grados de libertad y un CPU de al menos un núcleo corriendo debajo de los 2.0GHz de frecuencia.

1.3 Síntesis del problema

¿Cómo implementar algoritmos para predicción de la posición en sistemas empotrados de bajo costo manteniendo la precisión, consumo de CPU, y tiempo de respuesta en valores aceptables?

1.4 Descripción de la solución

La presente tesis pretende ser una guía para los investigadores que necesitan una mayor precisión en el cálculo de la posición para sistemas empotrados de bajo costo. Para conseguirlo se pretende trasladar los algoritmos utilizados en investigaciones anteriores a una plataforma de desarrollo de bajo costo para luego caracterizarlos. En la Figura 1.2 se muestran cuáles son los pasos a seguir para poder llevar esos algoritmos desde equipos de alto desempeño hacia unos con procesamiento reducido con la finalidad de ser caracterizados.

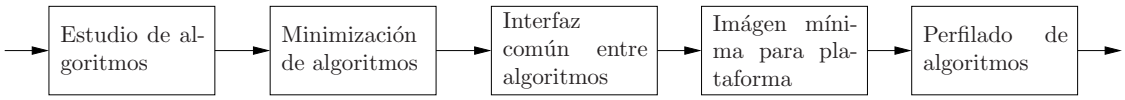


Figura 1.2: Solución propuesta

Se inicia con el estudio de cuáles son los algoritmos utilizados hasta el momento en el campo, este análisis permitirá conocer las ventajas y desventajas de las diferentes aproximaciones. Luego se debe escoger cuáles de estas aproximaciones pueden ser traducidas a sistemas empotrados de bajo costo e implementar en ellos optimizaciones que ayuden a la plataforma a obtener un buen resultado. Un paso vital para la comparación de las características de cada algoritmo es la creación de un programa común para obtener los datos de los sensores y desplegar los resultados, esto solo es posible si se oculta el algoritmo bajo una interfaz común. La imagen de tamaño mínimo permite maximizar el uso de los recursos y que la

caracterización obtenga resultados no afectados por el sistema operativo. Por último se pretende analizar las siguientes características para todos los algoritmos implementados:

- Precisión de la predicción de la ubicación
- Tiempo de respuesta
- Consumo de CPU

El costo de la plataforma a utlizar para llevar a cabo la investigación en sus componentes principales son:

- Receptor GPS: $39.95
- IMU: $19.95
- Raspberry Pi 2: $39.95

Esto da un costo parcial del sistema de $99.85, según [16]. Este costo es parcial, ya que no incluye la fuente de alimentación de 5V con conexión micro-USB, ni la interconexión de los elementos, requeridos para que el sistema funcione.

1.5 Investigaciones anteriores sobre predicción de la ubicación

Se pretende con esta sección resumir los hallazgos encontrados durante la investigación bibliográfica que han permitido definir el problema presente en el campo de la predicción de la posición. Los principales enfoques utilizados son:

- En [4], se reduce el consumo de potencia de los collares de rastreo para ganado mediante la reducción de la frecuencia en la toma de datos de GPS. Para aproximar la posición se reporta la última velocidad y dirección medida al controlador y este determinará por cuanto tiempo esto es aceptable antes de realizar la siguiente lectura del GPS. Este cambio en comunicación más constante con menores lecturas de GPS les permitió alargar la vida de la batería de cuatro semanas a seis meses.
- Los autores de [3] reportan la posición de cada bote mediante el uso de Unidades de Medición Inercial (IMU). Con el IMU de cada bote se mide la aceleración instantánea y la frecuencia de la misma (frecuencia de cada remada). La precisión conseguida por los autores es de 0.3m en 3s y 1.2m en 20s sin GPS.
- En [2], se describen los problemas que presenta un robot autómata al pasar por zonas de obstrucción de la señal. Para resolver dichos problemas los autores transforman las lecturas del GPS a coordenadas en un plano cartesiano y lo integran con el odómetro del vehículo mediante el filtro extendido de Kalman. Este método permitió a los autores alcanzar la precisión buscada (menos de 40cm de error) y su recomendación fue utilizar además un Sistema de Posicionamiento Inercial (INS) para evitar el error por deslizamiento en las llantas.
- Los autores de [5], utilizan un INS retro-alimentado por las lecturas de GPS, filtradas mediante Kalman, lo que les permite reducir el error a un 2.5%. Ellos sugieren que en caso de requerir una mayor precisión se requieren cámaras, odómetros o mapas.

- Mediante el filtro extendido de Kalman (EKF), en [11], se triangula la posición de una aeronave no tripulada (UAV) durante la pérdida de la señal GPS en la misma. Se requiere para ello que el escuadrón conste de tres UAVs y dos de ellas sigan en conexión mediante DGPS.
- En [6], se utiliza una IMU para insertar la aceleración directamente a un EKF y a un filtro extendido correlacionado de Kalman (CEKF). Los autores exponen el método, pero no la corrección del error.
- En [17], se comparan el EKF, el Filtro Sin Esencia de Kalman (UKF), diferencias divididas en primer orden (DD1), y diferencias divididas en segundo orden (DD2). De la comparación los autores concluyeron que los filtros UKF y DD2 son los deseables en caso de presentarse no linealidades fuertes, caso contrario DD1 y EKF son los que deben ser utilizados.
- Con el objetivo de reducir los costos, en [18], se utiliza una INS basada en sensores micro-mecánicos (MEMS) y para compensar la imprecisión de los nuevos sensores se utiliza un sistema de posicionamiento más preciso llamado Sistema Europeo Geoestacionario de Sobre-capa para Navegación (EGNOS). Los autores concluyeron que el INS basado en MEMS tiene demasiado ruido para periodos sin acceso al satélite.
- Para [7], los autores implementan una combinación del Filtro de Partículas (PF) con EKF y una comparación entre cada uno por separado. Ellos concluyeron que el filtro compuesto tiene 18% mejores resultados para ausencia de GPS por 60 segundos y equivalente periodos entre 10 y 30 segundos.
- En [8], se exponen los resultados de utilizar redes neuronales artificiales (ANN). Con ellos los autores afirman que las ANN logran entender las no-linealidades de la aceleración de un vehículo. La precisión se mejora en un 35% sobre el EKF. Se debe destacar que el vehículo utilizó un sensor DGPS.
- Los autores de [9] tratan de minimizar el tiempo de aprendizaje requerido por las redes neuronales. Se logró reducir el aprendizaje a 25 iteraciones y el error de Raíz Media Cuadrática (RMS) de 22 a 2.
- Otra prueba sobre redes neuronales es realizada en [10] mediante el uso de redes neuronales de construcción (CNN), pero sus resultados son incompatibles con las investigaciones anteriores.
- Para reducir el costo de procesamiento en sistemas de tiempo real en [19] se propone el uso de redes neuronales de base radial (RBF), con la cual se reduce el error de 571.8 metros usando EKF a 35.5 metros con RBF.
- En [12] se realiza una comparación de la precisión de las Redes Neuronales Artificiales de tipos: Evolutiva (ENN), Recurrente (RNN) y Auto Regresiva (ARMANN) como método para fusionar los datos del sensor DGPS con los datos del INS. Durante la comparación se confirma una precisión de la ubicación de 0.12 metros de error, pero no muestran los efectos de la ausencia del sensor DGPS por periodos largos de tiempo.
- Los autores de [20] demuestran cómo un sistema operativo a medida y de tiempo real puede solucionar problemas para el muestreo determinístico de datos. Utilizan una BeagleBoard Black y una imagen de tiempo real con el núcleo preventivo del kernel de Linux para leer la aceleración a una frecuencia determinada.

1.6 Objetivos de la investigación

1.6.1 Objetivo general

Caracterizar el desempeño de diversos algoritmos para la predicción de la ubicación en sistemas empotrados de bajo costo.

1.6.2 Objetivos específicos

- Estudiar y seleccionar diversos algoritmos para la predicción de la ubicación de un vehículo.
- Implementar los algoritmos seleccionados en lenguajes de alto nivel para utilizarse en un sistema empotrado.
- Comparar las características de consumo de potencia, uso del CPU, tiempo para obtener el resultado, y precisión de la predicción de los algoritmos implementados.
- Elaborar un régimen de pruebas que permita evaluar los algoritmos seleccionados para las características mencionadas anteriormente.

1.7 Estructura del documento

En el Capítulo 2 se aclaran todos los conceptos teóricos necesarios para comprender la solución planteada. Algunos elementos que destacan del capítulo son la explicación del filtro de Kalman y sus variantes, el concepto del proyecto llamado Yocto y las bibliotecas FANN y KFilter.

La solución se encuentra detallada en el Capítulo 3. El mismo describe todos los pasos necesarios para crear una aplicación empotrada en una imagen de tamaño mínimo, así como los obstáculos para llevar todas las piezas a una plataforma diferente a la de construcción. Ese capítulo se divide en las siguientes secciones: selección de algoritmos a caracterizar, creación de una aplicación empotrada, y banco de pruebas. En ellos se describen los algoritmos a utilizar, el flujo que utiliza la aplicación creada, y las pruebas que se van a realizar, respectivamente. Además en la sección 3.2 se describen todos los pasos necesarios para realizar la compilación cruzada para una imagen de tamaño mínimo.

En el Capítulo 4 se muestran los resultados obtenidos mediante la utilización del banco de pruebas definido en el Capítulo 3 para cada uno de los diversos algoritmos implementados. Además se analizan dichos resultados para determinar cuál es el efecto de cada algoritmo sobre un sistema empotrado de bajo costo. Por último en el Capítulo 5 se exponen las conclusiones obtenidas del presente trabajo, así como las recomendaciones para los trabajos posteriores.

Capítulo 2

Marco teórico

En el presente capítulo se explican los elementos teóricos que se utilizan como base para la solución del problema que en este documento se describe. Los primeros elementos a explicar son los algoritmos estudiados para la predicción de la posición. El Filtro de Kalman (KF) y su variante no lineal, el filtro extendido de Kalman (EKF). Seguidamente se trata el uso de las redes neuronales para predecir las lecturas del sistema a partir de un único sensor, así como el modelo matemático que describe el movimiento de un objeto.

Seguidamente, se procede a explicar las herramientas a utilizar para elaborar la aplicación empotrada para el cálculo de la posición. Se inicia con Yocto, CMake, y GNU Make los cuales son utilizados para la construcción. Además se detallan las bibliotecas que implementan los filtros para la predicción de mediciones, KFilter y FANN. Seguidamente se describen las herramientas utilizadas para facilitar los cálculos e interacciones dentro de la aplicación empotrada: "proj.4", wiringPi y GPSd. Por último se describe la plataforma a utilizar para método de pruebas y los sensores para la toma de los datos.

2.1 Filtro de Kalman

El filtro de Kalman como se define en [21], describe un sistema lineal que permite predecir y corregir no linealidades de fuerzas externas no incluidas dentro del modelo.

En [4], [5], [2], [22] y [21] se explica cómo el filtro es aplicable a cualquier sistema lineal discreto que pueda ser descrito por las ecuaciones (2.1) (2.2).

$$x_k = Ax_{k-1} + Bu_{k-1} + w_{k-1} \tag{2.1}$$
$$z_k = Hx_k + v_k \tag{2.2}$$

Donde (2.1) representa la predicción del estado k de la señal x a partir del estado anterior $k-1$. La variable A es la ganancia que relaciona el valor anterior con el nuevo valor. El factor Bu_k es también llamado el factor de control opcional externo. Las variables w y v

son las representaciones del ruido intrínseco de la medición, el cual debe ser ruido blanco con representación Gausiana. La variable H es una constante que define la lectura de los sensores z_k.

El KF, explicado en [23], se implementa mediante un ciclo de dos series de ecuaciones. La primera serie de ecuaciones son (2.3) y (2.4), las cuales describen la etapa de predicción y pre-cálculo de la covarianza del error respectivamente.

$$\hat{X}_k = A\hat{x'}_{k-1} + Bu_{k-1} \tag{2.3}$$
$$P_k = AP_{k-1}A^T + Q \tag{2.4}$$

Los valores de A y B (usualmente matrices) son los mismos que se utilizan para describir el sistema en (2.1), H está definida en (2.2) y los valores iniciales para $\hat{x'}_{k-1}$ y P_{k-1} determinan el comportamiento del sistema y qué tan rápido las lecturas converjan al valor correcto.

La siguiente etapa del filtro es la corrección de las medidas tomadas y la covarianza del error para lo cual se utilizan las ecuaciones (2.5), (2.6) y (2.7) que calculan la ganancia de Kalman para el presente ciclo, el valor estimado corregido y la nueva covarianza del error.

$$K_k = \frac{P_k H^T}{HP_kH^T + R} \tag{2.5}$$
$$\hat{X}'_k = \hat{X}_k + K_k(z_k - H\hat{X}_k) \tag{2.6}$$
$$P'_k = (I - K_kH)P_k \tag{2.7}$$

El elemento R es la covarianza del error de lectura, mientras que Q es el valor de la covarianza del error del sistema como un todo. La definición de este último valor puede causar el desfase del sistema. Ambos valores son considerados como variables de ajuste que pueden diferenciar sistemas si se cambian abruptamente como se explica en [21].

2.2 Filtro extendido de Kalman

El filtro extendido de Kalman, como se explica en [21], viene a habilitar KF para funciones no lineales. En [17], [18], [11], y [6], se utiliza para modelar el sistema de manera no lineal. EKF realiza una linealización del promedio y la covarianza, de manera que las funciones no lineales se convierten en funciones no lineales cuyas variables son las mismas que para las expresadas en la sección 2.1.

Las ecuaciones del modelo no lineal quedarían como se describen en (2.8) y (2.9)

$$x_k = f(x_{k-1}, u_{k-1}, w_{k-1}) \tag{2.8}$$
$$z_k = h(x_k, v_k) \tag{2.9}$$

Las funciones utilizadas para la predicción de la señal están descritas por las ecuaciones (2.10) y (2.11)

$$\hat{X}_k = f(\hat{x'}_{k-1}, u_{k-1}, 0) \tag{2.10}$$
$$P_k = A_k P_{k-1} A_k^T + W_k Q_{k-1} W_k^T \tag{2.11}$$

Para la corrección de los datos obtenidos de la estimación se utilizan (2.12), (2.13) y (2.14)

$$K_k = \frac{P_k H_k^T}{H_k P_k H_k^T + V_k R_k V_k^T} \tag{2.12}$$
$$\hat{X}'_k = \hat{X}_k + K_k(z_k - h(\hat{X}_k, 0)) \tag{2.13}$$
$$P'_k = (I - K_k H_k) P_k \tag{2.14}$$

Para las ecuaciones (2.10), (2.11), (2.12), (2.13) y (2.14) las antes constantes A, W, H y V son ahora Jacobianos de derivadas parciales de f con respecto a x, f con respecto a w, de h con respecto a x y de h con respecto a v respectivamente. Estos Jacobianos deben ser recalculados en cada paso por la cual todas las matrices fueron indexados por k como se indica en [21].

2.3 Filtro sin esencia de Kalman

Como se explica en [18] y [17] el filtro sin esencia de Kalman es un extensión de la Transformación Sin Esencia lo cual permite que no se tengan que calcular los Jacobianos necesitados por la variación extendida del filtro.

Las ecuaciones (2.15), (2.16), (2.17), (2.18), (2.19) y (2.20) son las que describen el procesamiento predictivo de la señal.

$$\chi_k = f(\chi_{k-1}, u_{k-1}) \tag{2.15}$$
$$\hat{X}_k = \sum_{i=0}^{2n} W_i^{(m)} \chi_{k,i} \tag{2.16}$$
$$P_k = \sum_{i=0}^{2n} W_i^{(c)} \left(\chi_{k,i} - \hat{X}_k\right)\left(\chi_{k,i} - \hat{X}_k\right)^T + Q_k \tag{2.17}$$
$$\chi'_k = \hat{X}'_k \left(\hat{X}'_k \pm \xi \sqrt{P'_{k-1}}\right) \tag{2.18}$$
$$Z_k = h(\chi'_k) \tag{2.19}$$
$$\hat{Z}_k = \sum_{i=0}^{2n} W_i^{(m)} Z_{k,i} \tag{2.20}$$

Ahora en la fase de corrección de la predicción las ecuaciones que describen los datos son las (2.21), (2.22), (2.23), (2.24) y (2.25).

$$P_{Z_k} = \sum_{i=0}^{2n} W_i^{(c)} \left(Z_{k,i} - \hat{Z}_k\right) \left(Z_{k,i} - \hat{Z}_k\right)^T \tag{2.21}$$

$$P_{X_k} = \sum_{i=0}^{2n} W_i^{(c)} \left(\chi'_{k,i} - \hat{X}_k\right) \left(Z_{k,i} - \hat{Z}_k\right)^T \tag{2.22}$$

$$K_k = \frac{P_{X_k}}{P_{Z_k}} \tag{2.23}$$

$$\hat{X}'_k = \hat{X}_k + K_k(Z_k - \hat{Z}_k) \tag{2.24}$$

$$P'_k = P_k - K_k P_{Z_k} K_k^T \tag{2.25}$$

Para ambos ciclos de esta variante del filtro ξ representa $\xi = \sqrt{(L+\lambda)}$ con L como la cantidad de estados, λ es una constante compuesta. En [24] se presentan las χ_i, $W_i^{(c)}$, y $W_i^{(m)}$ descritas por las ecuaciones (2.26) y (2.27)

$$\chi_i = \begin{cases} \bar{x} + \left(\sqrt{(L+\lambda)P_x}\right)_i & \text{if } i \in [1, L] \\ \bar{x} - \left(\sqrt{(L+\lambda)P_x}\right)_i & \text{if } i \in [L+1, 2L] \end{cases} \tag{2.26}$$

$$W_i^{(c)} = W_i^{(m)} = 1/[2(L+\lambda)] \; i \in [1, 2L] \tag{2.27}$$

2.4 Filtro de partículas

Los filtros de partículas como se explican en [25] son sistemas que permiten calcular la probabilidad y la distribución de las muestras tomadas. Estos filtros permiten la interacción de las partículas de manera que se puede aproximar la distribución correcta del error de una función.

En [7] utilizan un filtro de Monte Carlo con muestreo recurrente para implementar las funciones de distribución del error utilizadas en un EKF. La naturaleza de estos filtros no permite que sean aplicados directamente como predictores de la posición. Para más detalles de los filtros de partículas ver [25].

2.5 Red neuronal artificial

Las redes neuronales artificiales son sistemas concebidos para copiar la forma en la cual el cerebro procesa la información tal como se explica en [26]. El modelo utiliza múltiples elementos de procesamiento sencillo llamadas neuronas capaces de comunicarse entre ellas. En los sistemas más sencillos cada neurona tiene una entrada y produce una salida. La salida de cada neurona es evaluada y se obtiene un coeficiente que refleja su importancia en el sistema.

Las neuronas tienen la capacidad de cambiar su valor coeficiente de acuerdo a una entrada externa que evalúe su salida. A este proceso se le conoce como aprendizaje. La capacidad de aprender convierte a las redes neuronales en aproximadores naturales como se afirma en [26].

Existen varios tipos de redes neuronales de entre los cuales se estudiaron los siguientes modelos:

2.5.1 Perceptrón múltiple capa

Como se describe en [26] un perceptrón múltiple capa consta al menos de tres capas: entrada, salida, y oculta. En este modelo todas las neuronas se encuentran conectadas hacia adelante, lo cual significa que todas las neuronas están conectadas con todas las neuronas de la siguiente capa y únicamente la siguiente capa.

Un perceptrón múltiple capa (MLP) tiene dentro de sus neuronas procesamiento no lineal mediante funciones sigmoides como las que se muestran en (2.28) y (2.29).

$$f(x) = \frac{1}{1+e^{-x}} \tag{2.28}$$

$$f(x) = tanh(x) \tag{2.29}$$

La más usada de las funciones para las neuronas es la ecuación(2.28), como se muestra en [26]. El aprendizaje de este tipo de redes usualmente se realiza mediante un entrenamiento de propagación hacia atrás. Esta propagación es basada en el error presentado por (2.30)

$$E = 1/2 \times (x_0 - \hat{x}_0)^2 \frac{1}{1+e^{-x}} \tag{2.30}$$

mientras que la evaluación de los coeficientes y los pesos de las conexiones entre las neuronas están dados por (2.31) y (2.32) como se define en [26].

$$\omega_{ij}^{(k+1)} = \omega_{ij}^{(k)} - \lambda \left(\frac{\delta E}{\delta \omega_{ij}} \right)^{(k)} \tag{2.31}$$

$$\vartheta_{ij}^{(k+1)} = \vartheta_{ij}^{(k)} - \lambda \left(\frac{\delta E}{\delta \vartheta_{ij}} \right)^{(k)} \tag{2.32}$$

2.5.2 Función de base radial

Las redes neuronales de base radial son perceptrones múltiple capa cuya función de acción es la mostrada en (2.33), la cual dibuja una función Gausiana. Esta función remplaza a (2.28) dentro del perceptrón.

$$\varphi(x) = e^{-\beta||x-\mu||^2} \tag{2.33}$$

Con la función no lineal Gausiana se obtiene un resultado donde prefieren los datos cercanos al promedio, como se describe en [9].

2.5.3 Evolutiva

La red neuronal evolutiva utiliza algoritmos evolutivos para calcular todos los parámetros de cada una de las neuronas. A diferencia de las dos anteriores formas de redes neuronales, donde la topología, los pesos y los coeficientes de las neuronas son predefinidos; en esta red todos ellos son cambiados por el algoritmo evolutivo en cada una de sus iteracionres para obtener la mejor red neuronal. Esta red se explica en [12].

Los autores en [12] explican cómo los algoritmos evolutivos genéticos son utilizados para crecer la red al punto óptimo de aprendizaje. Al mismo tiempo los algoritmos editan los pesos y existencias de conexiones entre las neuronas. El usuario está encargado de definir las características que calificarán el error de la red, el algoritmo evolutivo, así como los elementos de la red que van a ser cambiados (cromosomas).

2.6 Máquina de vectores de soporte

Las máquinas de vectores de soporte son algoritmos de clasificación que utilizan la creación de vectores no lineales para dividir los elementos en clases que permiten definir categorías como se explica en [27].

Al igual que los filtros de partículas, las máquinas de vectores de soporte no proveen una aproximación de valores de una señal, sino una clasificación de la señal.

2.7 Movimiento rectilíneo uniformemente acelerado

El Movimiento Rectilíneo Uniformemente Acelerado (MRUA) es el que describe un objeto cuya velocidad varía de manera constante y proporcional al tiempo transcurrido desde la última medición, así se explica en [28]. El cálculo de la posición para un objeto que describe un MRUA se puede determinar mediante las ecuaciones (2.35) y (2.34).

$$x = x_0 + x'_0 \times t + \frac{1}{2} \times at^2 \tag{2.34}$$

$$x' = x'_0 + a \times t \tag{2.35}$$

En (2.35) y (2.34) x representa la ubicación actual calculada, x_0 la velocidad inicial, x'_0 la velocidad en el instante 0, a la aceleración constante que tendrá el objeto y t el tiempo transcurrido en el instante 0.

2.8 Yocto

En esta sección se pretende que el lector entienda los conceptos básicos que soportan a Yocto y que lo convierten en uno de los mayores aliados para los investigadores en sistemas empotrados, así como para las industrias dedicadas a la producción de hardware empotrado.

Yocto es un proyecto de código abierto como se explica en [29] adoptado por la "Linux Foundation". Este proyecto se basa en OpenEmbedded, con la ayuda de la herramienta llamada "BitBake", tiene como finalidad crear imágenes completas de Linux para un ordenador. En otras palabras Yocto le permite al usuario crear su propia distribución de Linux que se va a utilizar en su solución final evitando desperdicio de recursos. Proyectos grandes como OpenSwitch (también de la "Linux Fundation") basan la producción de sus imágenes en el.

Una de las bondades de Yocto en sistemas empotrados es que permite realizar compilaciones cruzadas en un solo paso. Una compilación cruzada, según [30], se define como aquella compilación que se ejecuta en una máquina con arquitectura y o bibliotecas diferentes de la cual el programa creado va a ser ejecutado. Esto permite a los programadores hacer uso de computadores más potentes para generar programas cuyo destino son plataformas donde crear programas no tiene las herramientas o el espacio requeridos. Generar una compilación cruzada de manera correcta es uno de los principales retos que trae programar para sistemas embebidos como se infiere de [31].

Aún cuando Yocto permite realizar una correcta compilación cruzada en un solo comando,no implica que sea una herramienta sencilla de usar. La herramienta utiliza un sistema de capas con múltiples archivos de configuración con la finalidad de reutilizar al máximo los archivos existentes para todas las plataformas. Esta reutilización implica que los archivos de configuración cambian conforme se baja por las capas del sistema, lo cual hace un reto determinar cuál es la configuración final para la tarjeta.

La curva de aprendizaje para utilizar esta herramienta es alta, debido a que la extensa documentación existente solamente brinda los conceptos, pero nunca guías oficiales de cómo realizar los cambios necesarios para agregar programas o cambiarlos de una versión a otra. La mayoría del aprendizaje en Yocto debe hacerse mediante el análisis de configuraciones existentes y los conceptos documentados.

Yocto se puede dividir en dos grandes bloques: el sistema de construcción (OpenEmbedded) y las recetas de construcción (BitBake) ambos bloques será explicados a continuación.

2.8.1 BitBake

BitBake es un ejecutor de tareas, descrito en [32], que tiene la capacidad de correr eficientemente tareas en paralelo y detectar las dependencias entre ellas. Este es un programa escrito en lenguaje Python. Además de ejecutar tareas simples como lo haría GNU Make, BitBake agrega: soporte para tareas como obtener código y bibliotecas de diferentes fuentes, definición de dependencias, herencia de comandos de ejecución, entre otras.

Para realizar la construcción del programa BitBake hace uso de "recetas". Se puede identificar las recetas en Yocto buscando los archivos ".bb". Estas recetas son archivos con la meta-información básica del programa a ser compilado. En [32] se define la información se encuentra dentro de la misma. Algunos de esos elementos son:

- SUMMARY: título que tiene la receta
- DESCRIPTION: es la descripción en palabras simples que define la receta
- HOMEPAGE: página web donde se encuentra el programa
- LICENSE: tipo de licencia utilizada por el autor para la distribución (MIT, LGPL, entre otras)
- LIC_FILES_CHKSUM: el archivo y su identificador único para la licencia distribuida dentro del código fuente
- SECTION: sección en la cual se compilará, esto afecta el momento en el cual se construye
- RDEPENDS: dependecias del programa, en otras palabras elementos que deben ser compiladas antes que la presente receta
- inherit: clase de la cual depende. Este valor usualmente es el tipo de construcción que requiere el programa
- SRC_URI: dirección y protocolo que debe ser utilizado para descargar el código fuente del programa, cambios personles al código y como realizarlos
- PV: versión del programa
- FILES_${PN}: lugar de instalación del ejecutable

Los siguientes elementos de información utilizados por BitBake son los archivos de configuración (.conf). Estos archivos dirigen desde el compilador hasta los programas que van a ser instalados.

BitBake hace uso de clases. Al igual que los lenguajes de programación orientados a objetos, definen el comportamiento básico de ciertos elementos. En el caso de BitBake, como se explica en [32], definen los pasos de ejecución de tareas como compilar, empaquetar, copiar archivos de diferentes fuentes, entre otros. El usuario puede definir una clase para una tarea en particular. La ventaja de las clases es que son heredables por las recetas, lo cual hace que diversas recetas se comporten siempre de la misma manera. Por ejemplo una clase para ejecutables tipo CMake permitiría que se defina el esqueleto de como crear un programa con dicha herramienta una sola vez.

Con toda herencia viene el problema de diferenciación. BitBake implementa el concepto de capas mediante archivos .bbappend. Como se explica en [32] estos archivos sobreescriben las partes de las recetas o clases descritas por ellos. Esto permite realizar pequeñas modificaciones para diferentes plataformas sin tener que cambiar el programa o la forma como se compila para cada una de ellas.

2.8.2 OpenEmbedded

En [33] se describe como OpenEmbedded era un proyecto individual que Yocto adoptó como su sistema para construcción llamado Poky. OpenEmbedded es además una distribución genérica y empotrada de Linux. OpenEmbedded define la estructura de capas que permiten generar las diversas imágenes de sistemas para múltiples arquitecturas.

Una reestructuración de OpenEmbedded fue necesaria para que escalara a las necesidades de Yocto. Ahora, Yocto tiene diferentes repositorios para las Capas de Plataformas (BSP) de manera que los fabricantes pudieran agregar nuevas tarjetas como se explica en [34].

El sistema de construcción puede generar imágenes para la plataforma o el conjunto de herramientas necesarias para compilación cruzada ("toolchain").

Por último uno de los elementos que se destacan del sistema para la construcción de distribuciones es su capacidad de guardar en el servidor donde se construye la información para no volver a compilar todos los paquetes cada vez que se requiera una imagen. A estos elementos se describen en [34] con el nombre de "sstates". Estos archivos son generados utilizando la distribución del servidor específico de compilación y la plataforma destino. Los "sstates" pueden ser compartidos entre servidores de la misma distribución para reducir el tiempo de generación de una imagen. Por ejemplo OpenSwitch tiene un repositorio remoto de "sstates" que permite reducir el tiempo de compilación de los colaboradores utilizando Ubuntu de cuatro horas a una hora y media, aproximadamente.

2.9 CMake

CMake, descrita en [35], es un sistema ampliable para el manejo de la construcción de programas y de código abierto que es independiente del sistema operativo (OS) y del compilador a utilizar. El sistema permite generar desde bibliotecas hasta código envolvente tomando el compilador y herramientas necesarias del ambiente de desarrollo. Esta última característica es la que la hace que sea una de las principales formas para compilar de manera cruzada mediante Yocto.

CMake nace como la gran mayoría de proyectos de código abierto, debido a la necesidad creada en el proyecto "Visible Human" dentro del contexto del conjunto de herramientas para registro y segmentación de visión (ITK por sus siglas en inglés). Inicia el uso a mediados del 2000.

El sistema se basa en múltiples archivos llamados "CMakeLists.txt" distribuidos a lo largo del proyecto en curso. La jerarquía del proyecto será descrita por el mismo sistema de construcción (CMakeList.txt), dentro de cada una de las carpetas, así como mediante módulos que permiten la búsqueda e identificación de las bibliotecas necesarias. Los detalles del uso y cómo nació CMake se pueden encontrar en [35]

2.9.1 pkg-config

Es la herramienta utilizada por CMake para realizar la búsqueda de programas instalados en el ambiente donde se va a realizar la compilación. Es un programa GPL versión 2, cuya primera implementación fue realizada por James Henstridge. Este luego fue traducido a C por Havo Pennington. Las versiones siguen evolucionando como se describe en [36].

2.10 GNU Make

Como se describe en [37], Make es una herramienta para la construcción automatizada de programas. Make tiene como finalidad eliminar el manejo de dependencias de los comandos para construir y enmascarar dichas dependencias debajo de uno solo comando ("make"). Para poder conseguir la meta Make requiere mantener control de la última modificación de los archivos y ejecutar de nuevo en caso de que alguno de ellos cambie.

Tanto en [37] y [38] se encuentran los detalles necesarios para poder utilizar dicha herramienta como forma de construcción de programas pequeños y medianos. Un make file se compone de elemento (reglas) con el formato "resultado: dependencias". Donde las dependencias son revisas por un posible cambio. Cada programa requiere una serie diferente de reglas para ser construido, todas esas reglas se encuentran en un archivo llamado "makefile" En [38] se puede encontrar un ejemplo para construir un programa que depende de ocho archivos diferentes.

2.11 KFilter

El proyecto KFilter es una biblioteca que implementa el Filtro Extendido de Kalman en C++. Este proyecto es una implementación operacional y utilizada, a conocimiento del autor, en dos investigaciones del "Ecole Polytechnique de Montreal". Es una biblioteca con licencia LGPLv2 y su código se puede encontrar en [39].

Esta biblioteca es construida mediante un "makefile" lo que facilita su compilación cruzada e inclusión en la plataforma final. Aún cuando es un proyecto que no se ha actualizado en varios años es la biblioteca mejor documentada para filtros extendidos de Kalman que se encontró durante la investigación. KFilter utiliza una construcción del filtro que se apega al modelo matemático del KF, lo que ayuda a eliminar errores humanos durante la traducción del modelo al software. Además la biblioteca viene equipada con gran cantidad de ejemplos.

2.12 FANN

FANN (Fast Artificial Neural Network Library) es una biblioteca de código abierto, bajo licencia LGPL, que implementa Redes Neuronales Multi-Capa en lenguaje C. Es una biblio-

teca con una excelente documentación, además de ser un proyecto que se encuentra activo. FANN tiene capas de traducción para más de 20 lenguajes de programación diferentes, así como soporte para arquitecturas con punto fijo o punto flotante. Dicha biblioteca se puede encontrar en [40]

FANN es una biblioteca que se construye utilizando la herramienta CMake la cual está soportada por Yocto, su documentación permite una rápida creación de preceptrones múltiple capa. Además permite al usuario guardar y precargar aprendizajes a los modelos de redes neuronales que presentan una ventaja. Los autores afirman que los métodos utilizados en la biblioteca FANN obtiene resultados más rápido que otras 150 soluciones existentes.

2.13 GPSd

GPSd es un programa que corre como parte del sistema operativo y revisa todos los sensores de GPS conectados mediante consolas seriales o USB. El uso de GPSd pone una capa de aislamiento entre el tipo de sensor GPS y los datos requeridos por parte del programa. GPSd es una de las herramientas a escoger para proyectos empotrados, incluyendo Android que requieren de la posición como se afirma en [41].

GPSd lee la salida en texto, en varios formatos estándar, y lo convierte en variables de memoria que pueden ser leídas mediante llamadas a funciones de C, C++ y Python. En [41] también se describe cómo existen bibliotecas de otros autores que lo hacen disponibles en Java y Perl.

2.14 proj.4

El proyecto proj.4 como se explica en [42] es un filtro estándar que convierte la latitud y la longitud en coordenadas cartesianas y de vuelta. A este tipo de cambio se le llama proyección. En la presente investigación proj.4 es utilizado para convertir coordenadas en grados, dadas por el GPS, a metros en un plano cartesiano. Con las coordenadas en metros la velocidad, aceleración, y la posición se expresan en unidades del sistema internacional.

2.15 wiringPi

Como se explica en [43] es una biblioteca que permite el acceso a pines de GPIO expuestos por el chip BCM2835. Este último es el que se encuentra dentro de la plataforma Raspberry Pi. La biblioteca se encuentra disponible como código libre bajo licencia LGPL.

Esta biblioteca asocia y enumera los diversos pines de entrada y salida que tiene una tarjeta Raspberry Pi disponible. Además de eso permite al usuario configurar cada uno de ellos como elemento exclusivo de entrada o salida. El autor expuso también la funcionalidad para

manipular, leer o escribir, los objetos de entrada. Estos mapas están documentados en [43], donde se incluyen además una serie de ejemplos para el uso de la misma biblioteca.

2.16 Raspberry Pi 2

La Raspberry Pi 2 es una plataforma que representa una forma económica y sencilla de prototipar controladores para vehículos y aeronaves. Es por estas razones que se selecciona como medio de desarrollo para la presente investigación.

Las características de la plataforma se describen en [44]:

- Procesador Cortex A7 de cuatro núcleos a 900MHz
- 1GHz de memoria RAM
- 4 Puertos USB
- 40 GPIO pines
- Puerto Ethernet
- Lector de tarjeta SD
- Interfaz de vídeo (Salida HDMI y vídeo)
- Interfaz para cámara
- Procesador de vídeo 3D

Estas características le permiten a la plataforma ser utilizada desde controladores empotrados hasta correr servidores pequeños para centros de entretenimientos caseros con Windows 10 o Ubuntu Mate.

Dada la popularidad de la tarjeta existen fabricantes de sensores con lineas de productos dedicadas a dicha plataforma como la presente en [16]. Estos sensores especializados hacen uso de pines específicos de la tarjeta lo cual permite un prototipado rápido sin necesidad de elementos extra como reguladores de tensión. Estos sensores, aun cuando son especializados, siguen manteniendo un bajo costo.

2.17 Sensor GPS

El sensor GPS seleccionado es el MTK3339 empacado para Adafruit en el producto que se encuentra en [45]. Dicho producto facilita un sensor GPS con 66 canales y una frecuencia de refrescamiento máxima de 10Hz mediante una interfaz serial de lectura.

Se seleccionó esta presentación de sensor GPS dadas las siguientes características:

- Interfaz de comunicación sencilla
- Taza de refrescamiento hasta 10Hz
- 66 canales de seguimiento
- Compatibilidad con antenas activas.
- Antena integrada

La interfaz serial es una de las interfaces más utilizadas para sensores GPS debido a lo sencillo de enviar comandos, así como la facilidad para ser conectado virtualmente en cualquier plataforma gracias a los adaptadores USB-Serial. Durante la investigación para la selección del receptor GPS no se pudo encontrar ninguno cuya interfaz no fuera serial.

La taza de refrescamiento es un factor importante ya que permite aumentar la cantidad de mediciones realizadas a altas velocidades, por lo cual se tiene una mayor densidad de puntos de posición.

De la investigación sobre sensores se obtiene que en promedio tiene la mitad de canales (33 de búsqueda y 11 de seguimiento). Los sensores con alta frecuencia de refrescamiento tiene como mínimo la misma distribución de canales (66 de búsqueda y 22 de seguimiento) permitiendo al receptor desempeñarse mejor en áreas de cambio constante de satélites.

Por último la compatibilidad del sistema con antenas activas permite, mediante bajo costo, incrementar la precisión del sistema sin alterar el software ya elaborado. Además la antena interna permite hacer pruebas de concepto sin antenas externas y pruebas de baja confianza de la señal GPS.

2.18 Sensores IMU

Un sensor de INS requiere al menos un acelerómetro en tres ejes y algo que permita determinar el giro (cambio de dirección) en los mismos tres ejes. A esto se le conoce como un IMU de seis grados de libertad. Para el sistema que compete a esta investigación requerimos que el objeto a medir pueda determinar su orientación con respecto al plano cartesiano de la tierra, además del cambio en la orientación. Se requiere como mínimo para el sistema un IMU con acelerómetro y magnetrómetro.

Dada la dependencia del sistema en el IMU cuando se pierde la señal del GPS se decide incrementar en tres grados de libertad (DOF por sus siglas en inglés) el sensor. En otras palabras utilizar un acelerómetro, giroscopio y magnetrómetro en tres ejes. El sensor seleccionado es el que provee Adafruit en [46], el cual mediante dos sensores se obtienen las medidas requeridas.

Es importante que todas las mediciones se realicen con la misma orientación, para no tener problemas en los cálculos de la ubicación. Esto se soluciona mediante la plataforma implementada en [46], ya que ambos sensores se encuentran colocados en la misma base y probados por el fabricante.

La siguiente razón por la cual el presente chip sobresale en la investigación, es que utiliza I^2C como medio de comunicación sin intermediación de ningún procesamiento. Esto permite que el programa creado pueda utilizar las señales sin alterar y ajustar los filtros para cada señal como sea necesario. El uso de I^2C bus como medio de comunicación también permite eliminar costos innecesarios ya que la plataforma seleccionada solo cuenta con un puerto serial (el otro tipo común de interfaz para IMUs con 9 DOF) que va a ser utilizado por el GPS.

Capítulo 3

Caracterización de algoritmos para predicción de la ubicación en plataformas de bajo costo

En este capítulo se explican todos los pasos necesarios para conseguir la caracterización de algoritmos para la predicción de la posición en sistemas empotrados de bajo costo. Cada uno de los pasos principales toma una de las secciones del presente capítulo. Estos pasos se muestran en la Figura 3.1. El primer paso es "seleccionar algoritmos". Dentro de este primer paso se define el modelo del sistema a utilizar, se justifica la selección de los algoritmos y se detallan las definiciones necesarias para poner el algoritmo en la plataforma.

El siguiente paso es la creación de una aplicación empotrada capaz de utilizar los algoritmos previamente seleccionados ("crear aplicación empotrada"). Los elementos requeridos en este paso son el diseño de una aplicación capaz de leer y predecir la posición de un vehículo, el diseño del algoritmo que realiza la predicción y la creación de una imagen de tamaño mínimo.

El último paso es la definición del "banco de pruebas" que permite caracterizar los algoritmos. Este paso está compuesto por la definición de un circuito físico donde el vehículo pueda realizar las pruebas necesarias, así como las variables que se necesitan para documentar los resultados.

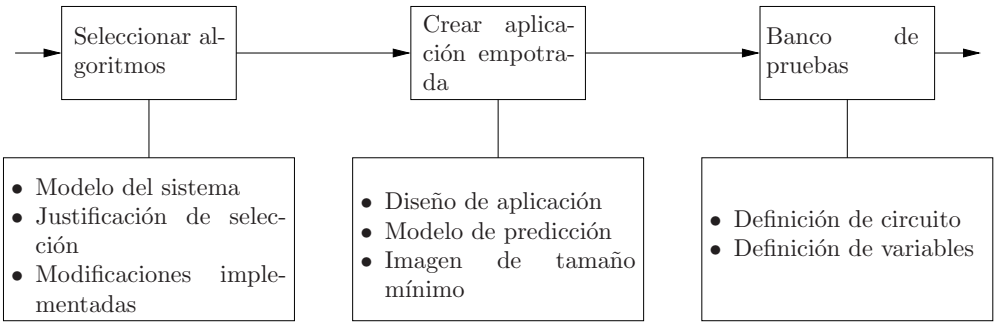


Figura 3.1: Detalles de solución propuesta

3.1 Selección de algoritmos a caracterizar

Esta sección pretende justificar cuáles son los algoritmos a caracterizar mediante la presente investigación. Antes de poder seleccionar cuáles son los algoritmos a utilizar, se debe definir cuál es el modelo que representa la posición del sistema para un tiempo dado. Dicha definición afecta los resultados proporcionados por los algoritmos.

3.1.1 Modelo matemático del sistema

El modelo matemático del sistema determina las decisiones a tomar en cada uno de los elementos en el cálculo de la posición. Es en este modelo que se decide hacer la primera optimización para reducir el costo computacional del algoritmo. Dada la periodicidad de las mediciones se decide convertir las ecuaciones de MRUA (2.34) y (2.35). La ecuaciones periódicas son (3.1) para la ubicación y (3.2) para la velocidad.

$$x_k = x_{k-1} + x'_{k-1} \times T + \frac{1}{2} \times a_k T^2 \tag{3.1}$$

$$x'_k = x'_{k-1} + a_k \times T \tag{3.2}$$

En (3.1), (3.2), k es la medición actual, mientras que $k-1$ es la medición anterior. Estas ecuaciones representan una aproximación en la cual se hace lineal el comportamiento de la aceleración entre cada medición a_k. Entre más pequeño el periodo T, mejor se aproxima el comportamiento al sistema real, como se afirma en el teorema de Nyquist explicado en [47]. Para la presente investigación se utiliza un periodo cercano a 1s ($T \approx 1$s). Este periodo es seleccionado debido a que es el periodo de lectura que tiene el sensor GPS utilizado.

3.1.2 Algoritmos seleccionados

La diversidad de algoritmos para la predicción de la ubicación de un objeto para que no dependa de la señal GPS, como se mostró en el Capítulo 2, es amplia. Se decide que para la presente investigación se va a realizar el esfuerzo de llevar los algoritmos que puedan presentar diferencias en el estrés de la plataforma empotrada. Estos algoritmos con diferencias sustanciales son EKF, con su enfoque de matrices, y ANN con su procesamiento no lineal en cada neurona.

Filtro extendido de Kalman

Se seleccionó el EKF debido a que es uno de los filtros más utilizados por los autores visitados durante la investigación bibliográfica, esto permite establecer una referencia con las investigaciones anteriores. Además de su uso, como se describió en la sección 2.2, es el elemento que mejor se ajusta a un problema con entradas no lineales.

Para la implementación de dicho algoritmo se seleccionó la biblioteca explicada en la sección 2.11. Dicha biblioteca requiere que el usuario defina todas y cada una de las ecuaciones, así como los valores de error de las lecturas. Es por esa razón que se debe acoplar el modelo expresado por (3.1) y (3.2) a (2.8) y (2.9) utilizadas para predecir el sistema mediante la variante extendida del filtro de Kalman. El resultado de este acople se muestra en (3.3) y (3.4).

$$\underline{\mathbf{x}}_k = \mathbf{A} \times \underline{\mathbf{x}}_{k-1} + \underline{\mathbf{B}} \times u + \mathbf{W} \tag{3.3}$$

$$\underline{\mathbf{z}}_k = \mathbf{H}_k + \mathbf{V}_k \tag{3.4}$$

Donde

$$\underline{\mathbf{x}} = \begin{bmatrix} x \\ x' \end{bmatrix}, \mathbf{A} = \begin{bmatrix} 1 & T \\ 0 & T \end{bmatrix}, \underline{\mathbf{B}} = \begin{bmatrix} T^2/2 \\ T \end{bmatrix}, \mathbf{W} = \mathbf{H} = \mathbf{V} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \underline{\mathbf{z}} = \begin{bmatrix} z \\ z' \end{bmatrix}$$

En ellas se mantiene la posición como x, mientras la velocidad cambia a la derivada de la posición x'. La variable u representa la entrada externa del sistema (la aceleración). Por último z y z' son las mediciones de los sensores de ubicación y velocidad respectivamente.

Como usuario de la biblioteca KFilter, en [39], se debió definir los valores de $\mathbf{A}, \mathbf{W}, \mathbf{H}, \mathbf{V}, \mathbf{R}$, y $\mathbf{Q}$ mediante funciones del objeto a crear por la biblioteca. En el caso de $\mathbf{R}$ y $\mathbf{Q}$ se mantuvo el error definido por [39] en su ejemplo de filtro para la posición de un avión.

Perceptrón múltiple capa

Se selecciona esta forma de predicción de señales debido a su efectividad en [8]. Este algoritmo fue implementado mediante el uso de la biblioteca explicada en la sección 2.12 (FANN). Dicha biblioteca redujo la interacción del usuario a tres características: cantidad de capas, número de neuronas por capa, y función de activación de las neuronas.

Para la cantidad de capas se utilizó de nuevo como referencia [8], donde los autores declaran que se utiliza una capa escondida, una capa de entrada y una de salida. Ellos utilizan un perceptrón por variable a calcular, este patrón se mantiene en la presente investigación. Esto implica que en la solución existen tres perceptrones uno para la posición, otro para la velocidad y uno para la aceleración.

La cantidad de neuronas en la capa oculta como se explica en [48] es un proceso que depende de cada problema y usualmente de relaciones matemáticas o experimentales. En el caso de la presente tesis se toma la vía experimental. Con esa finalidad se realiza una prueba por minuto y medio de entrenamiento. Se define que el entrenamiento debe realizarse sobre el mismo dato no más de 100 veces. Además este debe ejecutarse solamente si el error de la red es mayor a 0.1 y debe detenerse si es menor al mismo valor.

En la Tabla 3.1 se muestran los resultados del experimento incial para determinar la estructura. Se utiliza para todos los experimentos 3 neuronas de entrada, una neurona de salida y una función de activación a la salida lineal. La columna de estructura muestra los cambios

Tabla 3.1: Prueba previa de estructuras de la red neuronal

Estructura	Iteraciones promedio	Error promedio
Sigmoide (1x2)	2.062	0.0202
Sigmoide (1x30)	5.056	0.0228
Sigmoide (2x15)	2.162	0.0273
Gausiana (1x2)	2.322	1238.8
Gausiana (1x30)	1.335	0.0571
Gausiana (2x15)	0.989	0.0323
Linear (1x2)	1.411	0.0232
Linear (1x30)	1.169	0.0173

en la función de activación de las capas ocultas y entre parentesis el "número de capas x neuronas por capa".

Con base en el experimento anterior se define observa que las mejores opciones son: función de activación lineal con 30 neuronas, sigmoide con 2 neuronas y sigmoide con 30 neuronas. Se decide utilizar la última ya que, como se explica en [49], es una de las funciones de activación más común. Además la cantidad de iteraciones para 30 neuronas es la más alta, por lo que será una carga más representativa para el sistema.

En resumen el número de neuronas en cada capa se define de la siguiente manera:

- Entrada: Tres neuronas (una por cada elemento de entrada).
- Capa oculta: 10 veces el número de neuronas a la entrada, en este caso 30.
- Salida: Una neurona (una por variable de salida).

3.2 Creación de una aplicación empotrada

El resultado final de la investigación es una aplicación que es ejecutada en la plataforma Raspberry Pi 2 y es capaz de predecir la posición de un objeto en movimiento en caso que el GPS pierda alguna lectura. En esta sección se expone cuáles son las partes básicas de esa aplicación, así como el detalle del elemento que realiza la predicción, y el proceso requerido para llevarla a la plataforma con una imagen de tamaño mínimo.

3.2.1 Diseño de aplicación

En la Figura 3.2 se muestra la secuencia de eventos que realiza la aplicación para hacer posible la predicción de la ubicación.

El primer elemento de la Figura 3.2 es el que describe la inicialización. En este paso se crean todos los archivos donde se van a guardar las variables necesarias para su posprocesamiento. Además, es el paso encargado de configurar los pines de la tarjeta para el uso requerido (entradas, salidas, puerto serial, e I^2C). Ese bloque también es el que levanta los sensores

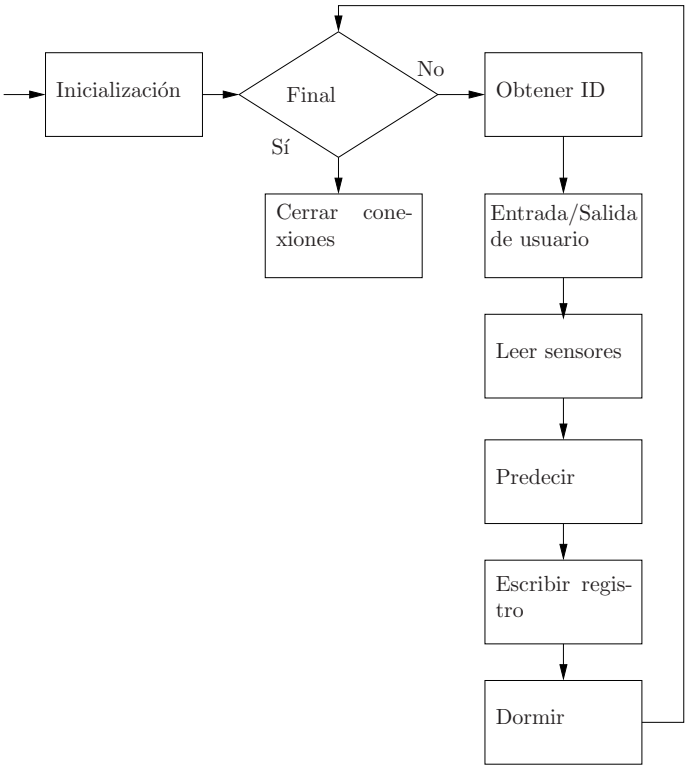


Figura 3.2: Diseño de aplicación para predicción de la posición

y crea el objeto que va a realizar la predicción de la posición. También realiza una toma inicial de datos de manera que se pueda centrar al objeto en el plano cartesiano para poder reducir el valor de la posición y sea más sencillo para los algoritmos ejecutarse.

El siguiente elemento mostrado es una decisión si hemos llegado al final. Esta comprobación se realiza mediante la intercepción de interrupciones por el programa. La intención es permitirle al usuario declarar el final del programa y que todas las conexiones abiertas al sistema (archivos y sockets) se cierren sin que exista corrupción de datos.

El primer elemento dentro del filtro es obtener un identificador para los datos. Este identificador es únicamente requerido para el programa de pruebas. El identificador es lo que va a permitir, en caso de que alguna lectura falle, poder identificar cuál lectura del GPS se tomó en el mismo instante que del INS y cuál es el resultado filtrado por el algoritmo. Esto asegura una comparación correcta de los datos al final.

El siguiente elemento presente dentro del ciclo de la Figura 3.2 es la determinación de la entrada del usuario (el usuario pide simular un fallo). Esta lectura se realiza mediante la detección de un valor lógico en uno de los pines del Raspberry Pi 2. Además este bloque permite notificar al usuario mediante un LED, pin de salida, si existe un fallo en la lectura del GPS.

Una vez determinado si se desea o no realizar un fallo de GPS, se ejecutan las lecturas respectivas de ambos sensores. En el sensor INS se utiliza el dato anterior y el presente para calcular la posición con las integrales de la aceleración. En el lado contrario, el GPS utiliza las lecturas mediante GPSd, las proyectadas al plano cartesiano mediante proj.4 y deriva

esos resultados para obtener la velocidad y la aceleración.

El bloque "predecir" será explicado más adelante en detalle ya que afecta la forma en cómo los datos son fusionados para ambos sensores.

Los últimos dos pasos del ciclo presentado en la Figura 3.2 son: "escribir registro" y "dormir". El primero se refiere a escribir todos los valores tanto fusionados como los originales de los sensores en archivos para que puedan ser utilizados fuera de línea para determinar el error del sistema. El segundo, "dormir", lleva el programa a reposo hasta que sea el nuevo periodo de ejecución.

3.2.2 Modelo para la predicción de la posición

La predicción de la posición implica la intervención de elementos para cambiar la señal (coordenadas del sistema) antes de su salida. En la presente investigación se define el modelo de intervención presentado por la Figura 3.3. El esquema planteado consiste en leer la posición inercial y las coordenadas provistas por el GPS para fusionarlos. El dato fusionado es ingresado al filtro. Cada filtro predice de manera diferente la posición de salida de manera que la unidad de cálculo debe adecuarse a cada uno de ellos. Además la posición final se envía al sensor inercial para minimizar el error acumulado del mismo.

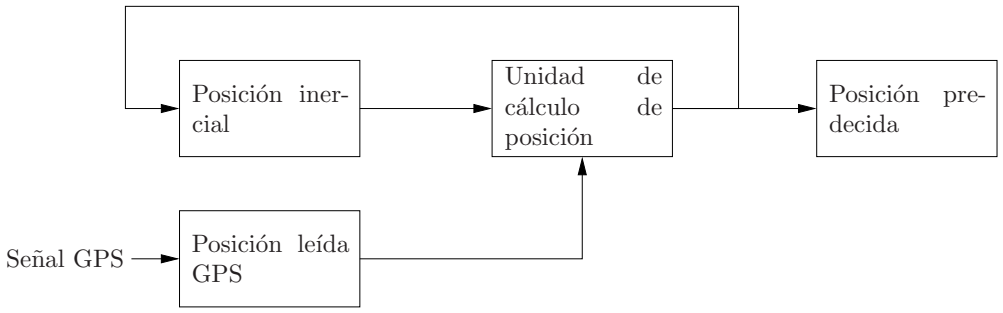


Figura 3.3: Modelo para la predicción de la posición

El bloque principal del modelo es el sistema de posicionamiento inercial (INS), mostrado en la Figura 3.3 como "posición inercial". Este bloque genera una posición relativa mediante la lectura de acelerómetros, giroscopios y magnetrómetros. Dado que es una posición relativa, requiere inicialización cada vez que se utilice. Con la finalidad de minimizar el error, de manera similar a como se realiza en [5], se decide inyectar de nuevo la posición al INS.

El bloque de "posición leída GPS" es el encargado de leer la posición provista por el sensor GPS y trasladarla a una posición relativa a la inicializada en el INS. Ambos bloques deben tener la misma referencia para que la tarea de la unidad de cálculo pueda ser realizada. Tanto el sensor GPS como el INS serán leídos al mismo tiempo.

La "unidad de cálculo de posición" es la encargada de tomar la decisión de cómo fusionar ambas lecturas. En esta unidad se encuentra el elemento encargado de la predicción, y las constantes para la fusión de ambos sensores. Los algoritmos seleccionados requieren diferentes tipos de entradas y serán explicados más adelante.

Con la finalidad de mejorar la portabilidad a un sistema empotrado se definió que la unidad de cálculo de la posición utiliza cada eje como una entidad independiente. Esto permite que el sistema pueda reducir el tamaño de las matrices matemáticas a resolver en los diversos algoritmos. Esto también permite que el sistema pueda ser paralelizado, esto no es realizado para la presente investigación con la intención de obtener una línea base del comportamiento en un único CPU.

Ya que se utilizarán diversos algoritmos sobre la misma aplicación empotrada se definió que ellos deben mantener la misma interfaz hacia la aplicación. Así es como se puede minimizar las diferencias en el programa principal y a la hora de evaluar los tiempos de cálculo se tome en cuenta únicamente la diferencia entre ambos algoritmos.

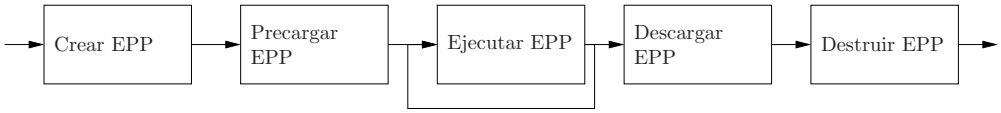


Figura 3.4: Funciones para interfase de algoritmos

La interfaz para los elementos procesadores debe cumplir con los pasos expuestos en la Figura 3.4. En ella se explica que el primer elemento requerido es una función de inicialización del Elemento Predictor de la Posición (EPP) “crear EPP”. En esta función se crea el modelo y se reserva toda la memoria necesaria para fusionar los datos de forma correcta. La siguiente llamada es la encargada de calcular la posición, llamada “ejecutar EPP”. Estos son los únicos elementos indispensables para crear un programa básico que permita predecir la posición en caso de falla del GPS.

Los otros bloques mostrados por la Figura 3.4 son deseables pero no indispensables para conseguir una predicción correcta. El bloque “precargar EPP”, es el encargado de llenar todas las constantes de los algoritmos, de manera que se pueden retomar las predicciones sin tener que volver a ajustar el EPP. El siguiente bloque que puede ser ignorado es el llamado “destruir EPP”, este bloque limpia la memoria del EPP antes de cerrar el programa de manera que no se pierda memoria. La pérdida de memoria es grave si el programa piensa cerrarse y abrirse múltiples veces sin reiniciar la tarjeta, en cualquier otro caso una pérdida de memoria es aceptable. Por último “descargar EPP” es una función que permita guardar la información de las constantes internas del EPP.

En la presente investigación se omiten “precargar EPP”, “destruir EPP”, y “descargar EPP”, y se dejan en caso que el sistema quiera ser optimizado para múltiples ejecuciones en diferentes momentos del tiempo.

Además de enfocar la diferencia del esfuerzo de ejecución debajo de la interfaz, esta capa permite extender la presente investigación bajo nuevos algoritmos sin requerir cambiar el programa completo. El nombre de la biblioteca es libPrediction.

3.2.3 Imagen de tamaño mínimo

Para asegurar que la imagen del sistema que estamos realizando no presenta ningún servicio que pueda dañar el rendimiento de los programas que se corren en dicha plataforma se debe crear manualmente una imagen de tamaño mínimo. Cada proceso extra puede afectar el rendimiento del sistema y por lo tanto los resultados obtenidos durante la comparación de algoritmos. Esta imagen debe incluir todos los elementos necesarios para habilitar la plataforma.

Cada plataforma de desarrollo tiene diversas capacidades y componentes internos que deben ser expuestos. Para poder generar un binario que pueda ser comprendido por los procesadores en cada una de ellas, Yocto requiere un paquete de soporte de tarjeta (BSP por sus siglas en inglés). El BSP requerido para una Raspberry Pi 2 se encuentra disponible en [50]. Esta capa brinda todo el soporte necesario para las particularidades de dicha tarjeta, así como configura la imagen para las capacidades y la arquitectura de la misma.

3.2.4 Compilación cruzada del programa

En esta sección se explica el proceso implementado para llevar todos los programas necesarios a la plataforma de pruebas, la cual tiene una arquitectura diferente a la máquina utilizada para el desarrollo de los mismos. En la Figura 3.5 se muestra el proceso utilizado para llevar un paquete a una plataforma empotrada mediante Yocto. Cada uno de los bloques serán explicados en detalle dentro de esta sección.

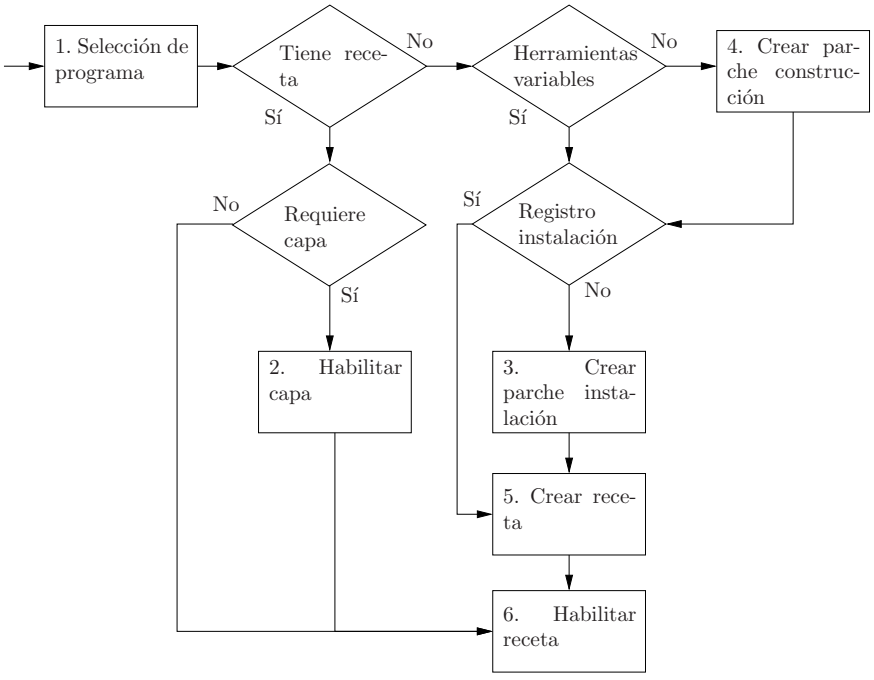


Figura 3.5: Proceso conceptual para llevar un programa a imagen de Yocto

En la Figura 3.5 el bloque “selección de programa” es el primer elemento que se basa en

encontrar el programa y su código fuente, así como determinar su forma de descarga. Con estos valores se debe empezar a buscar si el elemento ya fue utilizado en Yocto, este es el siguiente bloque llamado "tiene receta". En caso de que ya haya sido utilizado se necesita entender si la dicha receta requiere múltiples programas y clases, cuya respuesta es la que se busca por el bloque "requiere capa". Este bloque lleva al resultado final pasando o evitando el bloque "habilitar capa", el cual requiere como su nombre lo dice descargar e instalar toda una nueva capa en Yocto.

Continuando con el flujo presentado en la Figura 3.5, en el caso de que no exista una receta, el flujo requiere intervención directa sobre el código del programa a utilizar. El bloque "herramientas variables" pretende buscar dentro del método de construcción la posibilidad de cambiar las banderas y las herramientas de compilación con el fin de correr sin problemas dentro del ambiente de Yocto. Este bloque permite evitar el paso por el bloque "crear parche construcción", cuyo fin es permitir al programa ser construido dentro del ambiente de Yocto.

El siguiente elemento de búsqueda dentro del flujo es llevado a cabo mediante el bloque "registro instalación", el cual tiene la finalidad de determinar si existe un elemento que permita la búsqueda del programa dentro de la imagen de la plataforma. Esa búsqueda permite determinar si es necesario ingresar al paso "crear parche instalación" o se puede pasar directamente a "crear receta". Una vez con la receta terminada se procede al paso final "habilitar receta".

Los pasos que requieren creación de elementos o configuración del sistema son explicados a continuación.

Habilitar capa

Como su nombre lo dice, en esta etapa se busca obtener una serie de herramientas necesarias para leer y determinar cuál es la forma correcta de construir nuevos programas. Una nueva capa también incluye una serie de programas que pueden ser utilizados en la plataforma de desarrollo. Para habilitar una capa se debe conseguir el código fuente de la misma, así como ingresar la dirección de búsqueda dentro del archivo "bblayers.conf".

Crear parche construcción

La finalidad de la creación del parche de construcción es cambiar todos los elementos requeridos dentro del código fuente y su forma de construcción para que pueda ser parte de un sistema de compilación cruzada. Entre ellos incluye agregar reglas para la instalación, agregar archivos de licencia, y mayormente agregar variables que puedan sobreescribir las herramientas y las banderas de compilación.

Crear parche instalación

En este proceso se busca la forma de incluir todos los archivos necesarios tanto para la ejecución del programa, en caso de que esté diseñado para ejecución local, como para la búsqueda del mismo dentro de la plataforma. Ambos elementos son requeridos para que la imagen final pueda utilizar el binario de manera correcta.

Crear receta

En este paso se busca llenar todos los elementos que debe poseer una receta, estos pasos se describen en [34]. De ellos los más importantes ya fueron listados en la sección 2.8.1. Este elemento es la clave para utilizar Yocto como método constructivo para el programa deseado. Todo programa que desee ser llevado a la plataforma mediante Yocto debe tener una receta que describa su forma de construcción e instalación.

Habilitar receta

La forma de habilitar una receta varía dependiendo de lo que se requiera para el programa. Si el programa es una dependencia del ejecutable final que debe estar instalada se utiliza la variable "RDEPENDS" dentro de la receta del programa original. En caso de necesitar únicamente el elemento provisto por la receta se utiliza "CORE_IMAGE_EXTRA_INSTALL" dentro de la configuración de la tarjeta.

3.2.5 Programas que son requeridos por la aplicación empotrada

La aplicación empotrada requiere de varia bibliotecas, como se ha explicado durante el capítulo. Todas ellas tuvieron diferentes requerimientos para ser llevados a la imagen de tamaño mínimo, así como aportaron diferentes funcionalidades a la aplicación. A continuación se explican los aportes y los pasos para compilar de manera cruzada cada una de ellas.

Interfaz de lectura

Los sensores explicados en la sección 2.18 son utilizados mediante la lectura de registros en direcciones de memoria interna de cada uno de los dispositivos. Este tipo de acceso requiere seguir la documentación al pie de la letra y ser cuidadoso en los accesos. Es por esa razón que se decide utilizar una biblioteca que oculte esa complejidad de la lectura de la posición.

Adafruit provee en [51], [52], [53] y [54] archivos que permiten realizar consultas a los datos del sensor mediante una única llamada. Las fuentes encontradas están pensadas para ser utilizadas dentro del ambiente de compilación de un Arduino y no en Yocto.

Para llevar esos archivos a Linux construido dentro de Yocto se requiere cambiar el método implementado de lectura y la forma de distribución. La forma de distribución seleccionada es

una biblioteca independiente generada por el autor. Esta biblioteca además de un método de construcción, requiere un elemento de instalación. Para el primero se utiliza CMake y para el segundo un archivo ".pc". Seguidamente se debe crear la receta de la biblioteca y registrarla como dependencia del programa global. El nombre de dicha biblioteca es libAdaSensor.

Como programa nuevo que será utilizado en Yocto, libAdaSensor utiliza el proceso más largo mostrado en la Figura 3.5. Se tiene que ejecutar los pasos: "crear parche construcción", "crear parche instalación", "crear receta", y "habilitar receta"; en donde los parches de construcción e instalación son realmente los elementos completos ya que no existen para el programa seleccionado.

KFilter

Esta es la biblioteca, como se mencionó anteriormente, que será utilizada para habilitar el filtro como un algoritmo de predicción de la posición. Este necesita estar presente para que la biblioteca de predicción se ejecute correctamente.

La biblioteca aun cuando ya se encuentra preparada mediante su "makefile" para ser compilada con otras herramientas no tiene una forma para que otros programas la puedan buscar durante un proceso de instalación. Esto deja que KFilter requiere de los pasos 4, 5, y 6 ("crear parche construcción", "crear receta", y "habilitar receta" respectivamente) mostrados en la Figura 3.5. Para paquete de instalación se utiliza de nuevo un archivo ".pc" que es actualizado mediante el proceso de compilación. Además para la creación de la receta se requiere que el código fuente tenga en sus archivos la licencia, lo que requiere la búsqueda e inserción de la licencia LGPL especificada por el autor en [39].

FANN

Este es el paquete que provee la implementación del MLP que se utiliza dentro de la biblioteca de predicción para la aplicación empotrada.

La biblioteca FANN se encuentra preparada para la distribución y compilación en diversas plataformas de desarrollo. La anterior es la razón por la que el código fuente únicamente requiere de la ejecución de los pasos cinco y seis (crear y habilitar la receta para la biblioteca).

GPSd

GPSd es utilizado para eliminar la necesidad de leer el texto del puerto serial y traducirlo a datos válidos de latitud, longitud y altitud para la predicción de la posición.

El servidor GPSd tiene un código fuente que no es capaz de ser compilado de manera cruzada. Se realiza una búsqueda y se encuentra que ya existe una receta creada con los pasos necesarios cuatro, cinco y seis ("crear parche construcción", "crear receta", y "habilitar receta" respectivamente) de la Figura 3.5. Cuando se intenta copiar y habilitar la receta en el

ambiente de desarrollo utilizado, falla al incluir clases que no están presentes en el ambiente. Este fallo implica que la receta debe ser copiada con toda la capa donde fue encontrada.

La capa que es requerida para compilar correctamente se encuentra en [55]. Dicha capa es un proyecto activo y con muchas variaciones así como lo son Yocto y el BSP para el Raspberry Pi. Este movimiento de los archivos requiere que las versiones de los tres proyectos deban ser sintonizadas para poder generar una imagen. En Yocto se define el lanzamiento llamado "Yethro", para OpenEmbeded se utiliza la versión llamada "krogoth", y el BSP se detiene en el commit "2745399f75d7564fcc586d0365ff73be47849d0e" dado que no tiene una forma de guardar las versiones.

proj.4

El esfuerzo requerido para GPSd permite que la utilización de proj.4 dentro de la imagen solo requiere la implementación del paso seis de la Figura 3.5. El programa seleccionado para las transformaciones presenta un problema que debe ser resuelto, la documentación no coincide con la versión disponible en la receta seleccionada.

La aplicación proj.4 tiene una receta implementada para la versión 4.8.0, mientras que la documentación que se encuentra en [42] usa la versión 4.9.3 cuya interfaz para ser programada cambió completamente. El esfuerzo para llevar la versión 4.9.3 a Yocto es mucho mayor que realizar las pruebas para utilizar la versión ya existente. Esto implica que se debió revisar la implementación de la biblioteca para obtener el conocimiento de cómo la interfaz debe ser utilizada.

wiringPi

wiringPi es una biblioteca que se encuentra incluida dentro del BSP de Raspberry Pi, por lo que solamente requiere activación de la receta existente. Sin embargo se requiere instalar dicha aplicación en la maquina huésped si se quieren realizar pruebas de compilación local antes de llevar el programa al ambiente de Yocto. Es la utilizada para integrar las entradas del usuario dentro de la plataforma de pruebas, así como notificarle de los eventos al mismo.

3.3 Banco de pruebas

Con la idea de caracterizar los algoritmos es requerido definir una forma para probar el rendimiento y el consumo de recursos de cada uno de ellos. Para cumplir con ese objetivo debe establecer un circuito constante de pruebas, así como los parámetros de comparación para los datos obtenidos.

3.3.1 Definición de circuito de pruebas

El primer elemento a definir es la forma en que se deben realizar las pruebas. Esto implica los momentos y el tiempo durante el cual se debe eliminar la presencia del sensor GPS.

Los pasos que deben ser definidos para realizar y obtener resultados son los siguientes:

- Fallo simulado de GPS
- Definición de error de medición
- Tiempo de interrupción GPS
- Momentos de interrupción GPS
- Circuito de prueba

Los pasos se explican con detalle a continuación:

Lectura simulada de GPS

El fallo de GPS requiere ser controlable y causado por el usuario, sin que ninguno de los otros servicios del sistema se vea afectado. Se define para ello que el GPS siga funcionando de manera usual sin detenerse y que el bloque "posición leída GPS" de la Figura 3.3 devuelva un error al sistema. Ese mismo error es el que sería enviado en un error de lectura real en caso de fallo de comunicación.

El bloque mencionado anteriormente debe mantener registro de las lecturas en el cual se marque claramente cuáles lecturas fueron enviadas como fallos simulados hacía el resto del sistema y cuáles fueron consideradas reales.

Definición de error de medición

La investigación requiere poder determinar el error de la posición calculada en caso de fallo del GPS. Se define el error como la diferencia que tenga el sistema contra la medición directa del GPS. Esto es posible debido a que el GPS se mantiene realizando lecturas.

Tiempo de interrupción GPS

Debido a que el sistema que está realizando las mediciones utiliza la posición anterior como referencia implica que la nueva posición va a acumular error conforme pasa el tiempo. Los diferentes algoritmos van a tener sus propias formas de eliminar el error del sistema. Además, el tiempo de apagado debe mantenerse entre las diversas pruebas de manera que puedan ser comparadas una con otra. Se define para ello como tiempos de interrupción 5, 10, 15, 20, 25 y 30 segundos. Se realizan pruebas estáticas para todos los tiempos, pero las pruebas dinámicas se ejecutan únicamente para 5, 15 y 30 segundos.

Para mantener el tiempo controlado se habilita un puerto como entrada del sistema que permita al usuario disparar el inicio de un fallo simulado. Una vez iniciado el fallo este

se mantiene en fallo, durante el tiempo asignado por el usuario a la hora de la llamada del sistema. En caso de no especificar, se avanzará entre los tiempos descritos anteriormente por cada petición de fallo. Como segunda fuente de control se habilita una salida que manejará un LED y lo mantendrá encendido durante todo el tiempo que el fallo ocurra. Ambas fuentes de control se habilitan mediante [43].

Momentos de interrupción GPS

La funcionalidad del sistema en un momento sin GPS depende exclusivamente de la capacidad del acelerómetro para detectar los cambios de dirección del objeto. Esa es la razón por la cual los momentos en los cuales se pierda el GPS puede causar diversos tipos de error al sistema. Se define que el sistema debe ser probado durante los siguiente tipos de trayectorias:

- Detenido
- Movimiento en línea recta
- Movimiento en curva

Estos son seleccionados como movimientos a cubrir debido a los resultados mostrados en [56] en donde el desplazamiento en curva y recta presentan diferencia en los resultados para el mismo algoritmo. Se agrega además una prueba con el sistema detenido para identificar posibles problemas en la implementación del INS.

Circuito de prueba

El circuito de pruebas debe ser capaz de cubrir los tres tipos de rutas y durante el tiempo que se desea eliminar el GPS. Se busca dentro de las zonas cercanas y con poco tránsito de manera que el experimento pueda ser reproducido de manera consistente. El circuito utilizado se define en la Figura 3.6.

En la Figura 3.6 además se muestran los puntos aproximados donde se va a iniciar la falla de GPS. El sentido de circulación es el mismo que el de las manecillas del reloj. Los colores amarillo y rojo representan zonas donde el transito usualmente disminuye la velocidad. Los puntos seleccionados son secciones donde durante el tiempo máximo de 30 segundos se mantenga realizando el movimiento seleccionado para ese punto.

3.3.2 Comparación entre algoritmos

Los resultados de ambos algoritmos van a ser comparados por las siguientes características:

- Tiempo de cálculo
- Precisión de la posición calculada
- Uso de CPU

En la primera característica se busca tanto el tiempo durante el fallo de GPS, como durante la ejecución normal, según el algoritmo ambos tiempos pueden ser diferentes o relativa-

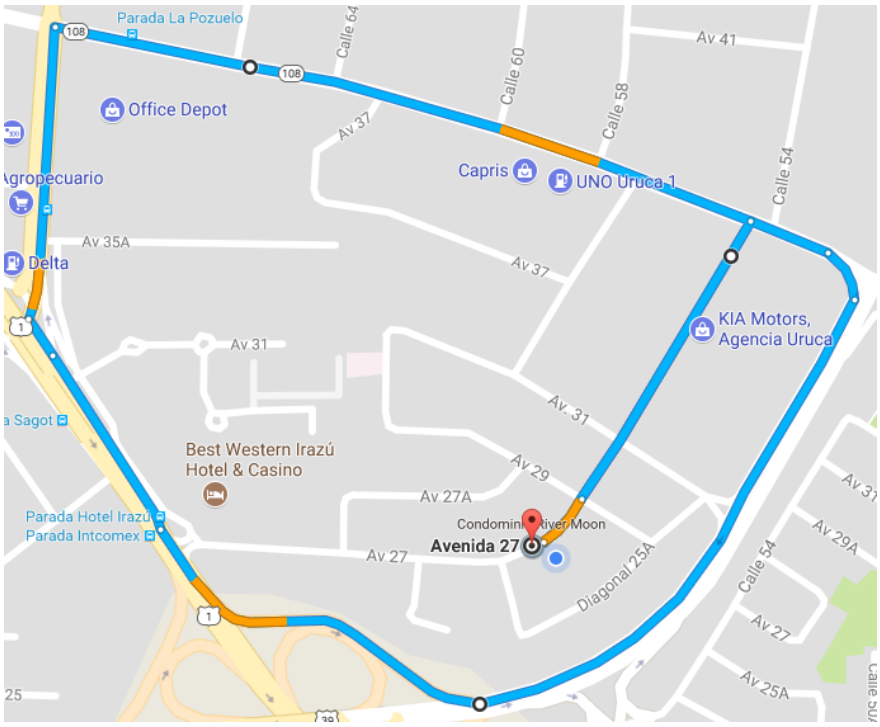


Figura 3.6: Mapa de la ruta a utilizar

mente similares. Esta característica es importante ya que limita el uso del algoritmo a los requerimientos del problema.

Para la precisión de la posición se definirá como la diferencia absoluta en metros con respecto a la posición otorgada por el sensor GPS. Esta característica se va a presentar como el promedio durante la falla, así como sus máximos.

Por último el uso del CPU refleja la carga computacional de los algoritmos durante el periodo de ejecución. Esta medición se encuentra estrechamente relacionada con la primera característica.

Capítulo 4

Resultados y análisis

En el presente capítulo se exponen todas las pruebas realizadas durante la redacción del presente documento. Estas pruebas se van a clasificar de acuerdo con la finalidad que estén buscando y la comparación de los elementos. La primera sección 4.1 demuestra las pruebas realizadas entre un programa corriendo en un sistema de tamaño mínimo o un sistema operativo comercial (Raspbian Jessy). La sección 4.2 muestra los resultados de ejecutar con la búsqueda de encontrar la mejor precisión del sistema. La sección 4.3 se explica cuál es el efecto que tienen los algoritmos en la plataforma seleccionada.

4.1 Optimización del sistema operativo

La utilización de un sistema operativo (OS por sus siglas en inglés), como se explicó en la sección anterior puede cambiar el rendimiento de una aplicación por completo. Es por esta razón que se realizó un experimento sencillo con la intención de identificar el impacto del mismo en el programa a correr.

En la Tabla 4.1 se utiliza un pequeño programa de ejemplo que filtra mediante Kalman 500 muestras de posición para una única variable. La medición del programa se realiza de inicio a fin, incluyendo la carga y escritura de archivos a la aplicación. Con este experimento se puede ver cómo con OS comerciales se obtiene un tiempo de corrida al menos cinco veces mayor que al utilizar uno a medida. Se utiliza para esta prueba los dos sistemas comerciales más populares disponibles para la tarjeta de desarrollo seleccionada: Ubuntu Mate y Raspbian. Como medida de comprobación se realizan diez mediciones en cada imagen para obtener el tiempo promedio de la duración del programa.

Se puede observar que los sistemas operativos a medida solo tienen una diferencia de un 11.01% entre ellos. Esto se debe a que el uso de CPU en tareas del sistema operativo son mínimas. Estas imágenes tienen únicamente los servicios señalados para cada compilación donde la propietaria no tiene más que SSH y SATO tiene más servicios activos. De los datos obtenidos se ve que una imagen de tamaño mínimo ofrece una reducción de un 83.05% en el tiempo de ejecución, contra Ubuntu. Incluso contra la mejor de las imágenes comerciales la

Tabla 4.1: Tiempo promedio de programa de ejemplo según OS

Sistema Operativo	Tiempo promedio (ms)
Tamaño mínimo	8.0
Yocto Sato	8.3
Raspbian	42.0
Ubuntu Mate	47.2

reducción del tiempo es de un 80.95%. Esto nos confirma la necesidad de utilizar un OS a medida, cuando se quiera maximizar el rendimiento.

En cuanto al consumo de potencia, en [57] se realiza un experimento donde se revisa el consumo de potencia con el sistema inactivo para sistemas operativos comerciales e imágenes creadas con Yocto. La plataforma utilizada en [57] es similar a la utilizada en esta investigación debido a que su consumo de potencia está definido en mayor parte por los sensores. Aún con esa condición se muestra como la imagen de tamaño mínimo produjo una disminución de 37.23%. Se realiza el mismo experimento en esta investigación y se obtienen los resultados mostrados en la Tabla 4.2.

Tabla 4.2: Consumo de potencia para OS

Sistema Operativo	Potencia (W)
Tamaño mínimo	1.15
Ubuntu Mate	1.84

En la Tabla 4.2 se muestran únicamente dos OS. Ubuntu Mate representando a los sistemas operativos comerciales y la imagen de tamaño mínimo utilizada para las siguientes mediciones. Los datos recolectados muestran una reducción en el consumo de potencia de 37.34% valor similar al obtenido en [57]. La reducción del consumo de potencia es otra razón por la cual se decide utilizar una imagen a medida para la caracterización del sistema.

4.2 Precisión de algoritmos

Durante la presente sección se muestran los resultados para los dos algoritmos implementados. Dichos algoritmos son Filtro Extendido de Kalman (EKF) y Perceptrón Múltiple Capa (MLP), cuyas siglas aquí descritas serán utilizadas para todas las figuras y tablas de la presente sección.

El primer experimento a realizar es revisar el comportamiento del sistema cuando se encuentra nivelado y detenido por completo. En este experimento se puede generar una base de cuál es el error a esperar de cada algoritmo. Los resultados obtenidos se muestran en la Tabla 4.3.

En la Tabla 4.3 se puede observar cómo el error mediante la utilización de EKF crece de manera exponencial con respecto al tiempo de falla. Esto describe de manera correcta

Tabla 4.3: Error máximo para objeto detenido durante falla de GPS

Duración de falla (s)	EKF (m)	MLP (m)
5	4.30	1.62
10	15.99	10.02
15	38.53	37.12
20	85.03	86.70
25	168.11	40.04
30	269.76	30.42

la acumulación del error calculado por un sistema INS. El error para la red neuronal se mantiene bajo en comparación directa con EKF. En las iteraciones superiores incluso es un 89.74% menor que el EKF. Esto se debe a que el experimento del sistema detenido se realiza mediante una única corrida con una separación de 30 segundos entre cada fallo. En una red neuronal este tiempo extra de entrenamiento permite que se adapte a la situación y elimine el problema del error acumulado. Es por eso que la medición durante un fallo de 25 segundos tiene un error 31.62% mayor que una falla de 30 segundos.

El siguiente paso a medir es el comportamiento del sistema mediante fallas en movimiento. Como se explica en la sección 3.3.1 se deben realizar pruebas en tres momentos diferentes. Cada uno de estos se realizará para diferentes tiempos de falla (5, 15 y 30 segundos). Con el fin de mantener la consistencia con el entrenamiento para el MLP se realiza el circuito de la Figura 3.6 realizando las tres fallas para el mismo tiempo durante todo el recorrido. Esto implica que se realizaron seis recorridos en total, uno por cada experimento usando EKF y MLP como algoritmos.

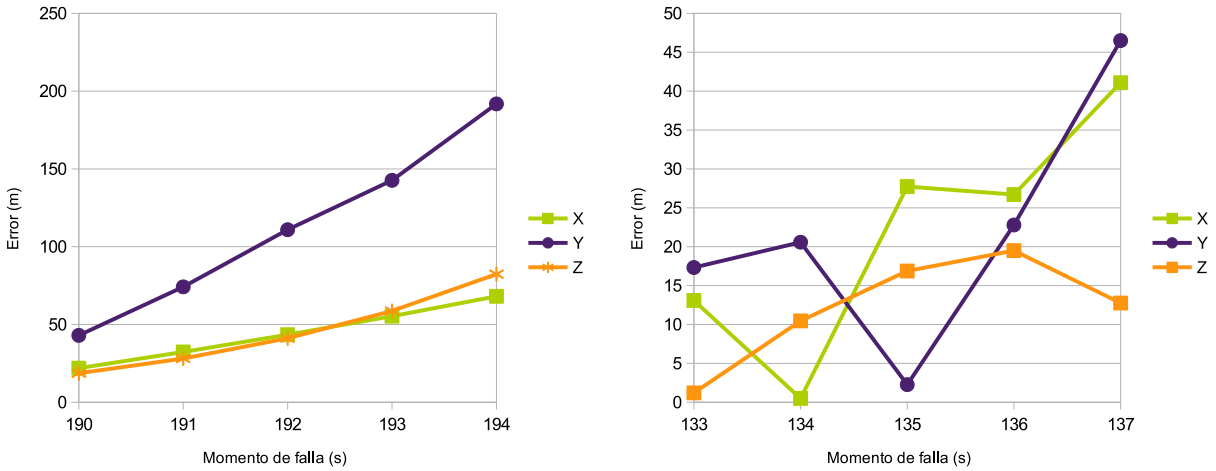
Las Figuras 4.1, 4.2, 4.3, 4.4, 4.5, 4.6, 4.7, 4.8 y 4.9 muestran el resultado de dichos experimentos de manera gráfica. En ellos se muestra como primer dato el error con un segundo de sistema sin GPS.

En la Figura 4.1 se muestra lado a lado el error de las predicciones para EKF y MLP durante una falla de cinco segundos en curva.

La Figura 4.1 muestra el comportamiento esperado para el EKF donde por cada segundo que pasa el error se va acumulando hasta alcanzar 191.8 metros de error en el eje Y. Por el contrario el MLP dependiendo de la entrada y del aprendizaje obtenido para dicha entrada el error se reduce o aumenta. Eso permite que el sistema mantenga el error controlado del INS y su valor máximo 46.50 metros en el mismo eje.

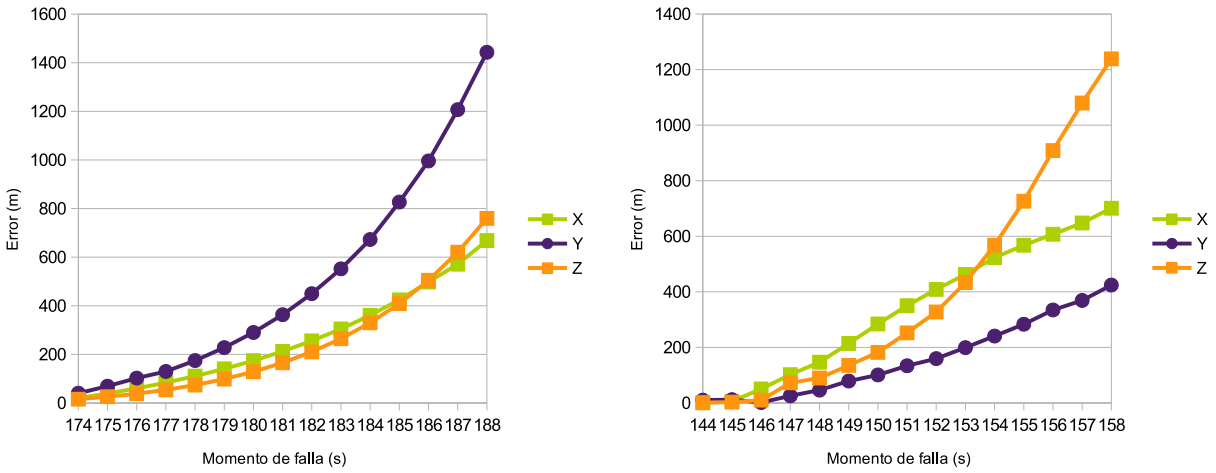
Realizando la falla en el mismo punto, pero durante un mayor tiempo se obtienen los resultados mostrados en la Figura 4.2. En ella se puede ver cómo ambas predicciones acumulan el error durante todo el tiempo que el GPS se mantiene en falla. En el caso de MLP dos de sus ejes (X y Z) cambian la pendiente en el segundo 152. Esto implica que el sistema se encontraba mejor entrenado para las entradas hasta el segundo 152 que para las que ocurrieron después en el eje X, y al revés para el eje Z.

Incrementando nuevamente el tiempo de falla para el mismo punto en el recorrido se obtiene



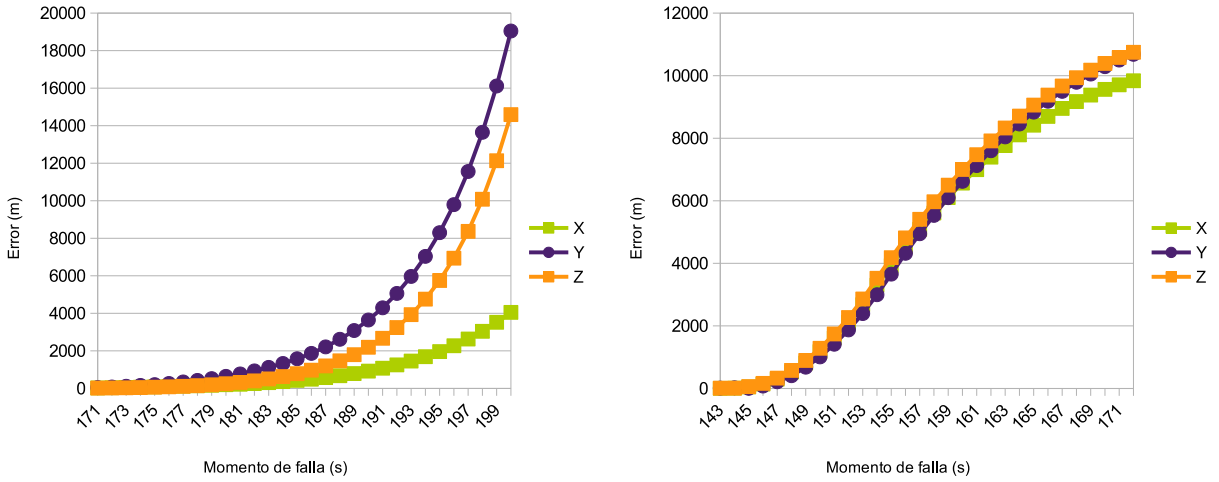
(a) EKF (b) MLP

Figura 4.1: Comportamiento de algoritmos en curva durante fallo de 5s



(a) EKF (b) MLP

Figura 4.2: Comportamiento de algoritmos en curva durante fallo de 15s

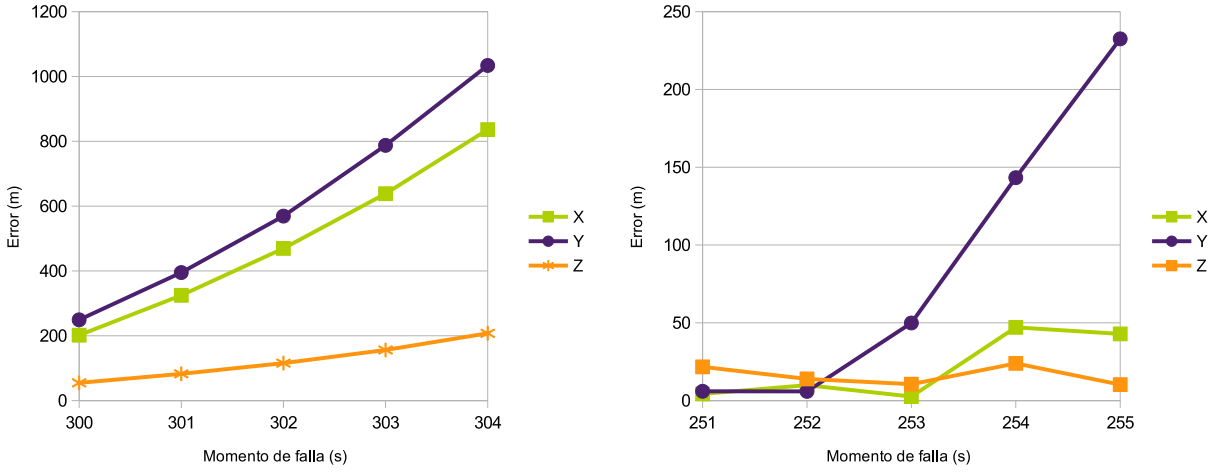


(a) EKF (b) MLP

Figura 4.3: Comportamiento de algoritmos en curva durante fallo de 30s

el gráfico de error que se muestra en la Figura 4.3. En ella se observa que para esta duración de falla ambos algoritmos acumulan el error sobrepasando los 1 000 metros para todos los ejes. En el caso de EKF se acumula 19 044.25 metros en su eje Y mientras que su contendiente alcanza 10 748.00 metros en el eje Z. En este punto hace inútil la posición del sistema.

El siguiente escenario a analizar es el fallo en pendiente. En este escenario se pone un esfuerzo mayor al sistema para la cancelación de la gravedad en el cálculo inercial de la posición. La Figura 4.4 muestra el rendimiento de ambos algoritmos para 5 segundos de falla.



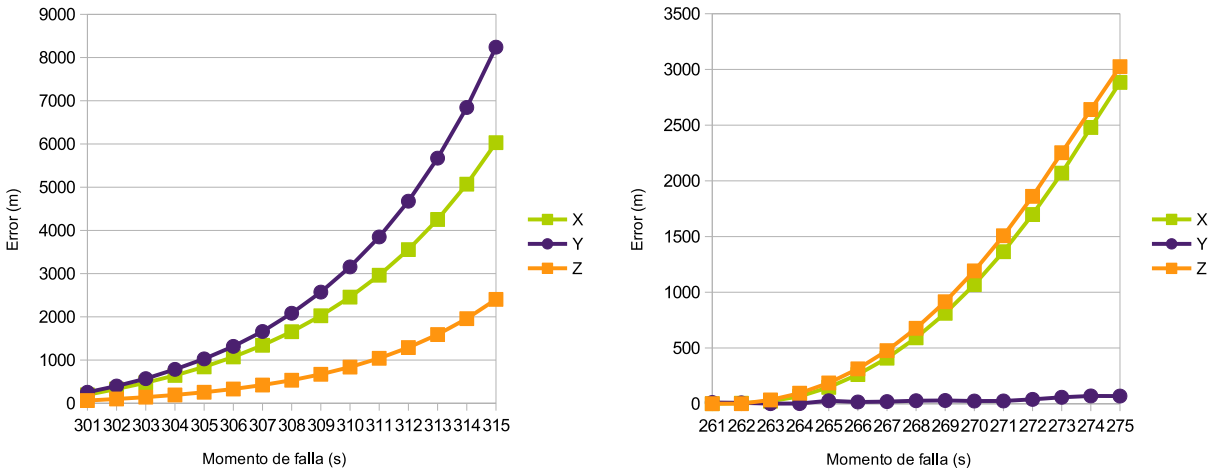
(a) EKF (b) MLP

Figura 4.4: Comportamiento de algoritmos en pendiente durante fallo de 5s

Se puede afirmar de la Figura 4.4 que gracias al entrenamiento los errores en el eje X y Z se mantienen menores a 50 metros, mientras que su componente en Y no tuvo suficiente entrenamiento para poder contrarestar la afectación de la gravedad. En el caso de EKF la gravedad afectó directamente todos los ejes de manera que la aceleración se propagada

produjo un error mayor a los 200m al final de la prueba. La afectación de la gravedad para EKF hizo que el error en Y llegara hasta 1 034 metros.

Se incrementa el tiempo de falla a 15 segundos y sus resultados se muestran en la Figura 4.5.



(a) EKF (b) MLP

Figura 4.5: Comportamiento de algoritmos en pendiente durante fallo de 15s

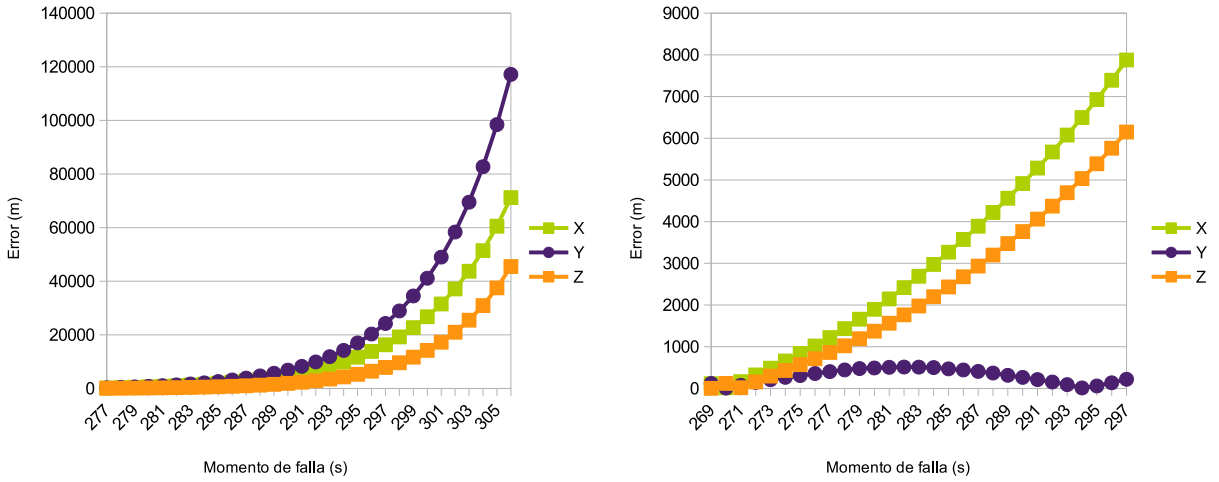
El error en la predicción de los algoritmos durante una falla de 15 segundos, como se espera de los datos analizados anteriormente, crece como se muestra en la Figura 4.5. En esta ocasión solamente uno de los ejes fue correctamente entrenado en MLP, mientras que los otros alcanzan un error aproximado de 3000 metros. Para el caso de EKF los tres superan los 2 000 metros de error llegando 8 240.50 metros en el eje Y. El EKF sigue manteniendo un crecimiento exponencial con respecto al tiempo como se mostró en los resultados en curva.

Los resultados de la Figura 4.6 obedecen al experimento realizado en pendiente para 30 segundos.

Durante los 30 segundos de falla, como se muestra en la Figura 4.6, el MLP se mantiene para el eje Y con un error a los 1 000m. Esto indica que la red neuronal fue capaz de entrenar de manera correcta el eje Y, y puede reducir el error incluso en pendiente. Al igual que los datos mostrados en la Figura 4.5 los ejes X y Z tienen un error que crece exponencialmente por todo el recorrido. Los valores máximos de error para el trayecto son 7 876 metros para el eje X en MLP y de 19 044.26 metros para el eje Y en EKF.

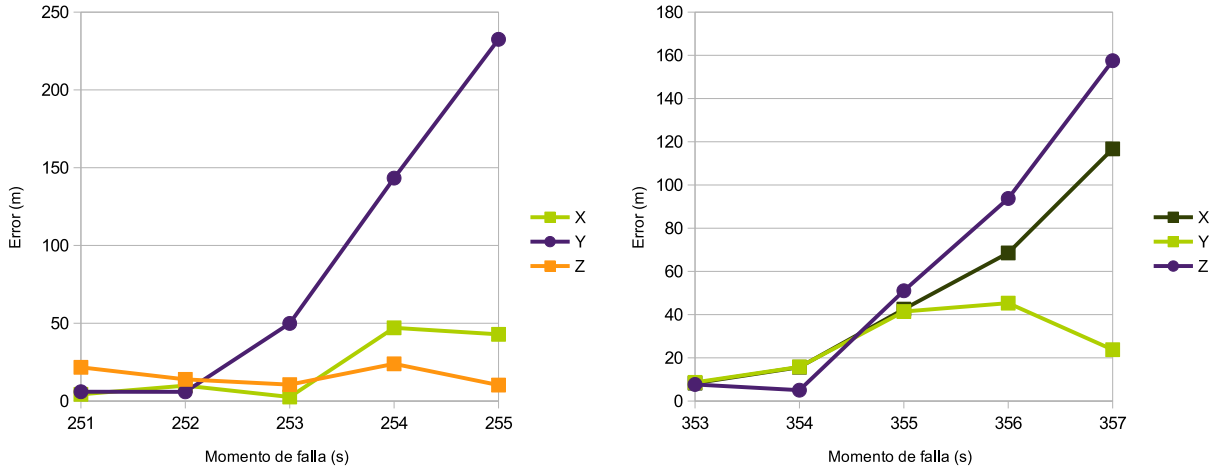
Durante el último periodo del recorrido se realiza una falla en línea recta y se trata que el vehículo se desplace a velocidad constante. Esto debería mostrar los problemas de cálculo en el INS, así como la acumulación normal de error en el sistema en modo de falla. La Figura 4.7 muestra dicho recorrido para un fallo de 5 segundos.

Se muestra en la Figura 4.7 cómo en esta oportunidad el EKF obtiene mejores resultados en el eje X. En los otros dos ejes el sistema se comporta mejor en MLP como anteriormente se estaba dando. En EKF el único eje en presentar un crecimiento exponencial es el eje Y, los otros valores se mantienen menores a 50 metros. En el caso de MLP los ejes Z y X son los



(a) EKF (b) MLP

Figura 4.6: Comportamiento de algoritmos en pendiente durante fallo de 30s



(a) EKF (b) MLP

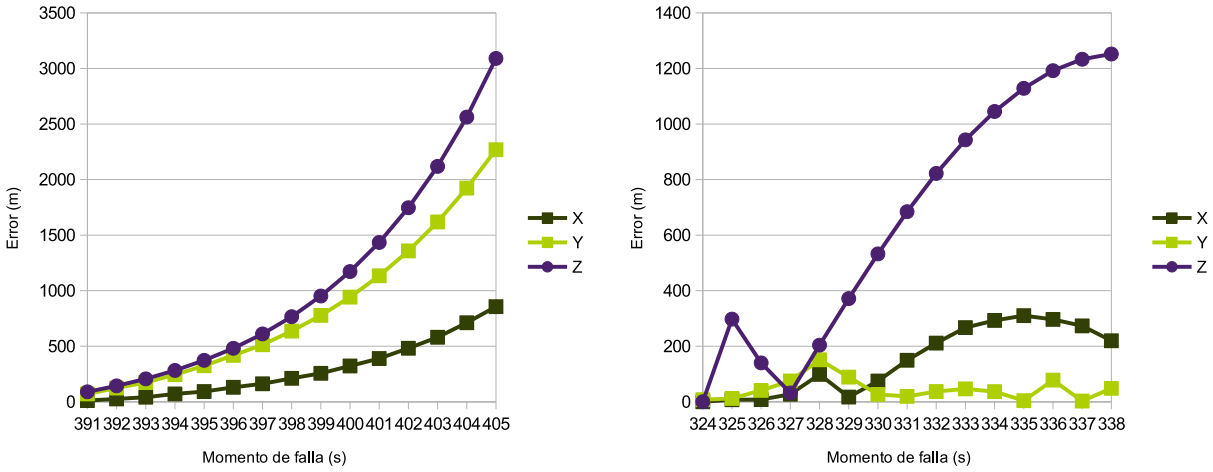
Figura 4.7: Comportamiento de algoritmos en recta durante fallo de 5s

que presentan un crecimiento exponencial.

Los errores máximos presentados en la Figura 4.7 son 353.04 metros en el eje Z para el EKF y 157.13 metros en el mismo eje para MLP.

Se genera la Figura 4.8 con los datos generados para el movimiento en recta con fallo de GPS por 15 segundos.

La predicción por MLP consigue mantener el error en el eje Y inferior a los 200m y decrementar ese valor hacia el final del recorrido, como se muestra en la Figura 4.8. Además se muestra cómo el eje X muestra una tendencia similar, pero debajo de 400 metros de error para el mismo algoritmo. El EKF sigue manteniendo la misma tendencia de incrementar exponencialmente durante todo el tiempo de falla, alcanzando 3 090.3 metros de error para el eje Z. La red neuronal tiene un error al final de la falla que alcanza los 1 251.87 metros

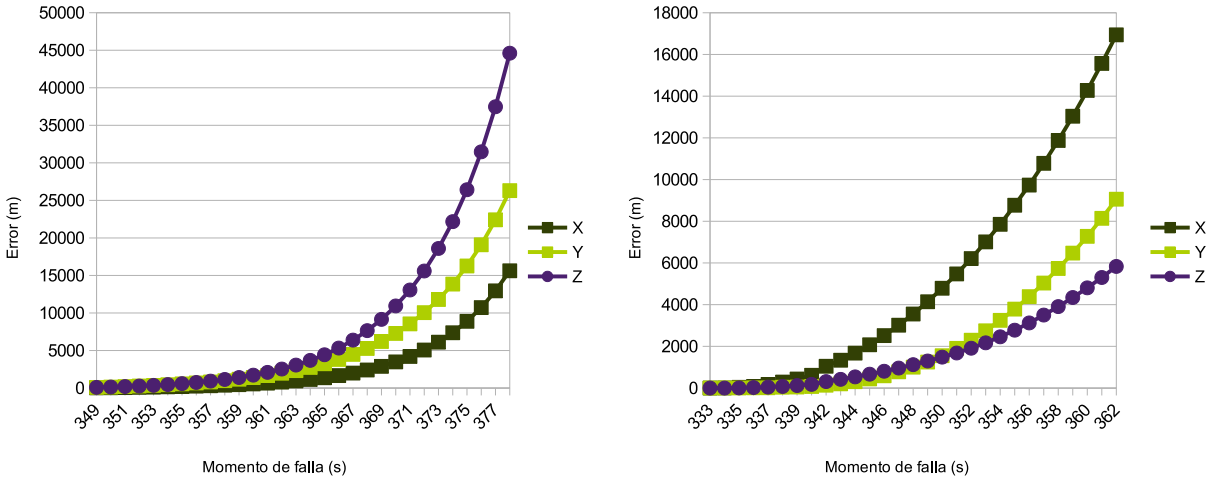


(a) EKF (b) MLP

Figura 4.8: Comportamiento de algoritmos en recta durante fallo de 15s

de error.

El siguiente gráfico (Figura 4.9) se produce con los datos recolectados en línea recta con velocidad constante por 30 segundos.



(a) EKF (b) MLP

Figura 4.9: Comportamiento de algoritmos en recta durante fallo de 30s

En ambas ocasiones mostradas por la Figura 4.9 los errores de todos los ejes muestran un crecimiento exponencial con respecto al tiempo. El error acumulado para el MLP que presenta el mejor de los casos 16 940.48 metros para el eje X. El EKF tiene un error de 44 609.3 metros, pero en este caso, para el eje Z.

Anteriormente se mostraron los datos en cada uno de los ejes para tratar de entender si alguno de ellos se veía afectado de manera diferente, pero el vehículo se mueve en un espacio tri-dimensional y como tal el error real es la magnitud del error que es descrito por el vector cuyos componentes son los errores en cada eje para un momento dado. Es por ello que

se genera la Tabla 4.4 en donde se muestra la magnitud máxima del error en cada uno de los experimentos (5, 15 y 30 segundos) sin importar en qué instante se realizó. Además se muestra el error máximo que observa el sistema cuando ambos sensores se encuentran en funcionamiento.

Tabla 4.4: Magnitud de error máximo para ambos algoritmos

Estado	**EKF (m)**	**MLP (m)**	**Mejora (%)**
Funcionamiento	51 541.47	39 403.81	23.55
Falla 5s	1 345.93	236.66	82.42
Falla 15s	10 490.66	4 179.63	60.16
Falla 30s	144 419.86	20 076.15	86.10

El primer dato interesante que arroja la Tabla 4.4 es que el sistema no se recupera instantáneamente. Los datos utilizados para la estadística incluyen absolutamente todos los datos en los cuáles no se presentó una falla de GPS, eso incluye inicialización y la posición luego de una predicción. En el caso de EKF se puede observar cómo se recupera rápidamente de una falla, mientras que el MLP tiene un incremento en el dato siguiente al máximo error en ausencia del GPS.

En los experimentos se puede observar cómo el único algoritmos que mantiene el error por debajo de los 500 metros es el MLP con 236.66 metros de error para una falla de 5 segundos. Este error es un 82.42% menor que el que presenta el EKF para el mismo experimento (1 345.93m).

En los siguientes dos casos ya la magnitud del error supera el kilómetro de diferencia, lo cual hace que las aplicaciones en las que se pueda utilizar el algoritmo se reduzcan. El EKF alcanza los 144.41Km de error mientras que MLP 20.07Km. Estos números siguen indicando que el error máximo es un 86.10% menor al utilizar MLP que utilizando EFK.

Los datos de la Tabla 4.4 solo muestran las magnitudes máximas pero no siempre son las más relevantes. Es por ello que se construye la Tabla 4.5. En ella también se muestran el promedio de los errores para el sistema en los experimentos y durante su funcionamiento usual.

Tabla 4.5: Magnitud de error promedio para ambos algoritmos

Estado	**EKF (m)**	**MLP (m)**	**Mejora (%)**
Funcionamiento	405.63	294.82	27.32
Falla 5s	400.68	73.32	81.70
Falla 15s	1 902.51	872.12	54.16
Falla 30s	15 357.40	6 280.47	59.10

En la Tabla 4.5 se muestra cómo el error luego de un fallo de GPS todavía se filtra en el sistema, es por eso que el promedio de los tres experimentos bajo funcionamiento tiene un valor mayor (294.82m) que durante una falla de GPS para 5 segundos (73.32m).

En esta tabla se muestra que la precisión menor a 500m ahora abarca EFK con error de 5 segundos y MLP con error a 5 segundos. Para ese mismo periodo se ve que la mejora de MLP, en la predicción de la posición, es de un 81.70%. En las otras dos mediciones la mejora es de un 54.16% y 59.10% (15s y 30s, respectivamente).

Los datos mostrados en las tablas 4.4 y 4.5 coinciden con las conclusiones de los autores de [5], donde se afirma que el INS tiene una desviación que impide que sea utilizado por largos periodos de tiempo. Los resultados mostrados en [17] también muestran cómo la pérdida de calidad de señal por pocos segundos (zona boscosa) puede llevar el sistema a un error de 15.27 metros.

4.3 Consumo de algoritmos

El siguiente y más importante aspecto a calificar es la capacidad del sistema para responder y ejecutar el programa y las predicciones realizadas. Es por esa razón que se toma el tiempo en cada una de las ejecuciones de los predictores en estado estacionario. Los valores se exponen en la Tabla 4.6.

Tabla 4.6: Tiempo de ejecución del algoritmo en diversos estados

Estado	**EKF** (μs)		**MLP** (μs)	
	Promedio	Máximo	Promedio	Máximo
Funcionamiento	110.36	250.00	821.11	5 445.00
Falla 5s	115.60	118.00	240.80	250.00
Falla 10s	110.40	117.00	247.80	346.00
Falla 15s	110.33	119.00	235.29	238.00
Falla 20s	114.45	220.00	236.25	239.00
Falla 25s	109.80	122.00	246.88	407.00
Falla 30s	117.71	210.00	235.62	331.00

Los datos de la Tabla 4.6 muestran que el sistema es capaz de dar la respuesta en un tiempo más rápido del esperado. En la investigación presente se está utilizando una frecuencia de lectura de 1Hz, esto quiere decir que la red neuronal implementada puede ser ejecutada 18 veces antes de pegar el límite de la capacidad serial del CPU para ejecutarla (tiempo máximo 5.45ms).

En el caso de EKF el tiempo máximo de ejecución mostrado en la Tabla 4.6 es de 250μs el cual le permite realizar hasta 4 000 iteraciones del filtro entre lecturas del CPU o 400 en caso de aumentar la frecuencia de lectura a 10Hz que soporta el sensor.

En esta investigación se ejecuta todo el sistema de manera serial, pero el procesador es de cuatro núcleos, lo que implicaría que se pueden ejecutar 4 trabajos en paralelo. La capacidad del CPU es suficiente para reducir el tiempo mediante la paralelización de cada uno de los ejes.

En la Tabla 4.6 también se muestra cómo en MLP tiene un tiempo 13.38 veces menor para la predicción que para el entrenamiento. De este resultado se puede señalar que en caso de aplicaciones de tiempo crítico se puede realizar el aprendizaje fuera de línea con suficientes datos y utilizar únicamente la predicción cuando sea necesario, reduciendo en 13.38 veces el tiempo de respuesta del algoritmo.

Una vez entrenado el MLP, su tiempo de ejecución es cercano a dos veces el tiempo de ejecución de EFK. Por ejemplo, en el caso de falla de 30 segundos EFK dura en promedio 117.71μs mientras MLP dura en promedio 235.62μs el cual es 2.00 veces mayor.

La memoria es otro recurso reducido en sistemas empotrados. Es por eso que se toman mediciones del uso de memoria para ambos algoritmos. En estado inactivo el sistema consume 40.22 MB de memoria RAM. En la Tabla 4.7 se muestra la diferencia de ambos algoritmos durante funcionamiento normal y durante falla.

Tabla 4.7: Uso de memoria para ambos algoritmos

Estado	**EKF (MB)**	**MLP (MB)**
Funcionamiento	2.30	2.86
Falla 30s	2.31	3.28

En la Tabla 4.7 se puede observar cómo el EKF se mantiene con una diferencia de 10KB entre un fallo y su operación normal (de 2.30MB a 2.31MB), no así para el MLP. Para este último algoritmo el cambio fue de 420KB (2.86MB a 3.28MB).

Por último se trata de realizar mediciones de uso de CPU. Para ello se utiliza la herramienta "top" de linux. Los resultados se muestran en la Tabla 4.8.

Tabla 4.8: Uso de CPU promedio del sistema durante un minuto

Estado	**EKF (%)**	**MLP (%)**
Reposo	0.02	0.02
Funcionamiento	0.00	0.11
Falla 30s	0.00	0.06

Observando la Tabla 4.8 no se puede determinar el efecto del EKF en el uso del CPU, el sistema en reposo llega a tener un valor promedio más alto que el sistema en funcionamiento. En cuanto al MLP se observa cómo el sistema en entrenamiento llega a subir el promedio del uso de CPU a 0.11%, mientras que en estado de falla llega a 0.06%. Esto confirma los resultados obtenidos por la Tabla 4.6 donde se mostraba que el sistema tenía un mayor tiempo de ejecución cuando se encontraba en reposo.

Capítulo 5

Conclusiones y recomendaciones

En el presente capítulo se describen las conclusiones y recomendaciones que se generan del presente trabajo.

5.1 Conclusiones

Se determina que para la plataforma Raspberry Pi 2 los sistemas operativos disponibles (Ubuntu Mate y Raspbian) incrementan el tiempo de ejecución de un programa con EKF en al menos 5.25 veces, por lo que es necesario crear una imagen de tamaño mínimo para caracterizar algoritmos de la predicción de la posición.

Al reducir la precisión de los sensores y la frecuencia a la que ejecuta instrucciones el CPU se logra reducir el costo de los mismos en un 96.44%.

Se reduce el consumo de potencia del sistema en un 37.34% al utilizar una imagen de tamaño mínimo en lugar de sistema operativo comercial.

Se seleccionó el filtro extendido de Kalman (EKF por sus siglas en inglés) y el perceptrón de múltiple capa (MLP) como algoritmos para predecir la posición debido a su uso en investigaciones como [17], [18] y [9] donde se afirma que son capaces de realizar un predicción por al menos 5 segundos. Además, dichos filtros son representativos del costo computacional de un sistema de predicción.

Mediante el uso de la herramienta Yocto se puede crear un programa que es capaz de predecir la posición en una plataforma empotrada como la Raspberry Pi 2. También mediante la creación de interfaces de software se puede utilizar el mismo programa con un algoritmo diferente, lo cual permite caracterizarlos en uso de memoria, CPU y tiempo de ejecución.

Es necesario definir un circuito de pruebas que incluya fallos en el GPS en tres momentos diferentes. Estos momentos son: detenido, en línea recta y en curva. Dado a que un vehículo se mueve en un espacio tridimensional es importante no olvidar el movimiento en curva sobre el eje Z (pendiente). El circuito mostrado en la Figura 3.6 contiene los tres momentos mencionados.

Se define como medio para la definición del error la comparación directa de las lecturas del GPS contra las lecturas efectuadas mediante el filtro.

El filtro extendido de Kalman, usando la aplicación implementada, es capaz de predecir una ubicación tridimensional cuya magnitud no supere los 1 345.93 metros de error en una falla de cinco segundos y el promedio durante ese periodo es de 400.68 metros. Durante esa falla el algoritmo le tomará un máximo de 118μs en calcular la nueva posición y en su funcionamiento normal el sistema no incrementará los 250μs. Dado el tiempo tan corto de ejecución, la aplicación completa no registra un uso distinguible del CPU en un promedio de 1 minuto. Todo esto se consigue utilizando un máximo de 2.31MB de memoria RAM

Con el perceptrón múltiple capa implementado dentro de la aplicación empotrada aquí descrita se consiguió que la magnitud del error tridimensional entre la posición predicha y la real tuviera un promedio de 73.32 metros para una falla de cinco segundos. Para este caso mencionado al sistema le toma un tiempo máximo de 250μs y un promedio de 240.80μs en predecir la nueva posición del sistema. En cuanto al tiempo de entrenamiento, el algoritmo toma un tiempo de ejecución máximo de 5.45ms y en promedio 0.82ms. El uso de CPU de dicho algoritmo para la falla mencionada anteriormente es de 0.06% y 0.11% en estado de entrenamiento. El gasto de memoria RAM para este caso es de 3.28MB.

La plataforma seleccionada, Raspberry Pi 2, es capaz de realizar 100 entrenamientos de las tres redes neuronales, con 34 neuronas cada una, en un tiempo aproximado de 5.45ms para la aplicación implementada.

Ambos algoritmos presentan sus ventajas para la predicción de la posición en sistemas empotrados. El EKF tiene un rendimiento 108.3% mayor (108.3% más rápido) que el MLP para una falla de cinco segundos. En la precisión de la predicción, para el mismo tiempo de falla, el MLP tiene un error 81.70% menor que el EKF. En el funcionamiento normal el EFK tiene un tiempo de ejecución 95.41% menor que el MLP. En el peor de los casos el EKF también tiene un uso de memoria 29.57% menor.

5.2 Recomendaciones

Se debe investigar la relación de la gravedad y la inclinación del vehículo con respecto al error de la posición generado por el Sistema de Posicionamiento Inercial (INS). Esto explica que los mayores errores presentados por el sistema se encuentre durante las activaciones de fallas en pendiente.

Si se quiere incrementar la exactitud con la que se mida el error de la posición obtenida de los algoritmos se deben trazar lineas con equipos sensibles sobre el circuito de manera que se pueda determinar la distancia perpendicular a los puntos conocidos y no a las mediciones del mismo equipo.

Con los algoritmos caracterizados se pueden variar el modelo de predicción de la posición para incluir filtros de Kalman o de otro tipo a la salida de la lectura de los sensores. De esta forma se puede eliminar ruido de las mismas. Inclusive se puede utilizar una red neuronal

entrenada para convertir la entrada del IMU a posición.

Bibliografía

[1] Instituto Nacional de Estadísca y Geografíade México, "Sistema de Posicionamiento Global (GPS)." [Online]. Available: http://www.inegi.org.mx/geo/contenidos/geodesia/gps.aspx?dv=c1

[2] R. Thrapp, C. Westbrook, and D. Subramanian, "Robust localization algorithms for an autonomous campus tour guide," in *Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on*, vol. 2, 2001, pp. 2065–2071 vol.2.

[3] B. H. Groh, S. J. Reinfelder, M. N. Streicher, A. Taraben, and B. M. Eskofier, "Movement prediction in rowing using a Dynamic Time Warping based stroke detection," in *Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP), 2014 IEEE Ninth International Conference on*, April 2014, pp. 1–6.

[4] R. Jurdak, P. Corke, A. Cotillon, D. Dharman, C. Crossman, and G. Salagnac, "Energy-efficient Localization: GPS Duty Cycling with Radio Ranging," *ACM Trans. Sen. Netw.*, vol. 9, no. 2, pp. 23:1–23:33, Apr. 2013. [Online]. Available: http://doi.acm.org/10.1145/2422966.2422980

[5] C. Cappelle, D. Pomorski, and Y. Yang, "GPS/INS Data Fusion for Land Vehicle Localization," in *Computational Engineering in Systems Applications, IMACS Multiconference on*, vol. 1, Oct 2006, pp. 21–27.

[6] G. Araújo Borges, A. P. Lanari Bo, and J. Yoshiyuki Ishihara, "An IMU/Magnetometer/GPS-based Localization System Using Correlated Kalman Filtering," *Departamento de Engenharia Elétrica, Universidade de Brasılia*, Jan 2008.

[7] P. Aggarwal, Z. Syed, and N. El-Sheimy, "Hybrid Extended Particle Filter (HEPF) for integrated civilian navigation system," in *Position, Location and Navigation Symposium, 2008 IEEE/ION*, May 2008, pp. 984–992.

[8] N. El-Sheimy, K. W. Chiang, and A. Noureldin, "The Utilization of Artificial Neural Networks for Multisensor System Integration in Navigation and Positioning Instruments," *IEEE Transactions on Instrumentation and Measurement*, vol. 55, no. 5, pp. 1606–1615, Oct 2006.

[9] M. Malleswaran, S. A. Deborah, S. Manjula, and V. Vaidehi, "Integration of INS and GPS using radial basis function neural networks for vehicular navigation," in *Control*

Automation Robotics Vision (ICARCV), 2010 11th International Conference on, Dec 2010, pp. 2427–2430.

[10] M. Malleswaran, V. Vaidehi, and S. A. Deborah, "CNN based GPS/INS data integration using new dynamic learning algorithm," in *Recent Trends in Information Technology (ICRTIT), 2011 International Conference on*, June 2011, pp. 211–216.

[11] G. Mao, S. Drake, and B. D. O. Anderson, "Design of an Extended Kalman Filter for UAV Localization," in *Information, Decision and Control, 2007. IDC '07*, Feb 2007, pp. 224–229.

[12] M. H. Refan and A. Dameshghi, "Comparing error predictions of GPS position components using, ARMANN, RNN, and ENN in order to use in DGPS," in *Telecommunications Forum (TELFOR), 2012 20th*, Nov 2012, pp. 815–818.

[13] Swift Navegation, "Swift Navegation Store." [Online]. Available: https://www.swiftnav.com/store

[14] VectorNav Technologies, "VectorNav Technologies Store." [Online]. Available: http://www.vectornav.com/purchase/buy-vn-100

[15] Amazon.com, "Intel Core 2 Duo E8400 3.0GHz Processor EU80570PJ0806M OEM TRAY: Computers Accessories." [Online]. Available: https://www.amazon.com/dp/B0019NKGR4

[16] A. Industries, "Adafruit raspberry pi." [Online]. Available: https://www.adafruit.com/category/105

[17] A. N. Ndjeng, A. Lambert, D. Gruyer, and S. Glaser, "Experimental comparison of Kalman Filters for vehicle localization," in *Intelligent Vehicles Symposium, 2009 IEEE*, June 2009, pp. 441–446.

[18] R. Toledo-Moreo, M. A. Zamora-Izquierdo, B. Ubeda-Minarro, and A. F. Gomez-Skarmeta, "High-Integrity IMM-EKF-Based Road Vehicle Navigation With Low-Cost GPS/SBAS/INS," *IEEE Transactions on Intelligent Transportation Systems*, vol. 8, no. 3, pp. 491–511, Sept 2007.

[19] L. Chen and J. Fang, "A Hybrid Prediction Method for Bridging GPS Outages in High-Precision POS Application," *IEEE Transactions on Instrumentation and Measurement*, vol. 63, no. 6, pp. 1656–1665, June 2014.

[20] Y. Arias-Esquivel, C. Salazar-García, and J. González-Gómez, "Real-time vibration analysis for structure fault detection," in *2016 IEEE 36th Central American and Panama Convention (CONCAPAN XXXVI)*, Nov 2016, pp. 1–4.

[21] G. Welch and G. Bishop, "An Introduction to the Kalman Filter," *Department of Computer Science University of North Carolina*, Jul 2006. [Online]. Available: http://www.cs.unc.edu/~welch/media/pdf/kalman_intro.pdf

[22] R. Faragher, "Understanding the Basis of the Kalman Filter Via a Simple and Intuitive Derivation," *International Journal of Computer Science and Network Security*, vol. 7, no. 3, pp. 128–132, Mar 2010. [Online]. Available: http://paper.ijcsns.org/07_book/201003/20100329.pdf

[23] J. Yick, B. Mukherjee, and D. Ghosal, "Analysis of a prediction-based mobility adaptive tracking algorithm," in *Broadband Networks, 2005. BroadNets 2005. 2nd International Conference on*, Oct 2005, pp. 753–760 Vol. 1.

[24] E. A. Wan and R. van der Merwe, "The Unscented Kalman Filter for Nonlinear Estimation," hardvard CS281 Course. [Online]. Available: https://www.seas.harvard.edu/courses/cs281/papers/unscented.pdf

[25] A. Doucet and A. M. Johansen, "A Tutorial on Particle Filtering and Smoothing: Fifteen years later," Dic 2008. [Online]. Available: http://www.cs.ubc.ca/~arnaud/doucet_johansen_tutorialPF.pdf

[26] D. Svozil, V. Kvasnicka, and J. Pospichalb, "Introduction to multi-layer feed-forward neural networks," *Chemometrics and Intelligent Laboratory Systems*, vol. 39, pp. 43–62, Nov 1997. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.133.124&rep=rep1&type=pdf

[27] C. J. Burges, "A Tutorial on Support Vector Machines for Pattern Recognition," *Chemometrics and intelligent laboratory systems, University of British Columbia*, vol. 39, pp. 43–62, 1997. [Online]. Available: http://research.microsoft.com/pubs/67119/svmtutorial.pdf

[28] FISICALAB, "Ecuaciones Movimiento Rectilíneo Uniformemente Acelerado (M.R.U.A.)." [Online]. Available: https://www.fisicalab.com/apartado/mrua-ecuaciones#contenidos

[29] Y. Project, "Hardware specific bsp overlay for the RaspberryPi device," 2010-2017, version Jethro. [Online]. Available: https://www.yoctoproject.org

[30] Meson, "Cross compilation." [Online]. Available: http://mesonbuild.com/Cross-compilation.html

[31] K. D. Cooper, D. Subramanian, and L. Torczon, "Compilation problems for embedded systems." [Online]. Available: https://www.cs.rice.edu/~keith/EMBED/

[32] R. Purdie, C. Larson, and P. Blundell, "Bitbake user manual." [Online]. Available: http://www.yoctoproject.org/docs/2.3/bitbake-user-manual/bitbake-user-manual.html

[33] "Openembedded." [Online]. Available: http://www.openembedded.org/wiki/Main_Page

[34] S. Rifenbark, "Yocto project mega-manual." [Online]. Available: http://www.yoctoproject.org/docs/2.3/mega-manual/mega-manual.html

[35] CMake, "CMake," 2004-2016, version 3.1.0. [Online]. Available: http://www.cmake.org/

[36] Freedesktop, "pkg-config," 2003-2016, version 0.29.1. [Online]. Available: https://www.freedesktop.org/wiki/Software/pkg-config/

[37] F. B. Laboratories and S. I. Feldman, "Make A Program for Maintaining Computer Programs," vol. 9, pp. 255–265, 1979.

[38] F. S. Foundation, "Make manual," May 2016. [Online]. Available: https://www.gnu.org/software/make/manual/make.html

[39] V. Zalzal, "KFilter: Free C++ extended kalman filter library," 2006-2008, version 1.3. [Online]. Available: http://kalman.sourceforge.net/

[40] S. Nissen, A. Spilca, A. Zabot, A. P. Bardelli, A. Stergiakis, B. Kuyumcu, C. de Almeida Barreto, C. Ellison, D. Morelli, Degski(degski@gmail.com), E. Nemerson, Freegoldbar(freegoldbar@yahoo.com), G. Megidish, G. Menier, J. Zelenka, J. Bates, Joslwah, L. H. Negri, M. P. Maia, M. Vogt, S. F. G. Salvador, S. Levis, S. Miers, S. Hauberg, T. Leibovici, and V. D. M. and, "FANN: Fast Artificial Neural Network Library," 2003-2016, version 2.2.0. [Online]. Available: http://leenissen.dk/fann/wp/

[41] E. S. Raymond, "Gpsd." [Online]. Available: http://www.catb.org/gpsd/

[42] F. Warmerdam and G. Evenden, "proj.4." [Online]. Available: http://proj4.org/

[43] G. Henderson, "Wiring Pi GPIO Interface library for the Raspberry Pi." [Online]. Available: http://wiringpi.com/

[44] M. Electronics, "Raspberry pi 2 v1.2 model b project board by raspberry pi." [Online]. Available: http://www.mcmelectronics.com/product/83-17791

[45] Adafruit. (2017) Adafruit Ultimate GPS Breakout. [Online]. Available: https://www.adafruit.com/product/746

[46] ——. (2017) Adafruit 9-DOF IMU Breakout. [Online]. Available: https://www.adafruit.com/product/1714

[47] S. W. Smith, *The Scientist and Engineer's Guide to Digital Signal Processing*. San Diego, CA, USA: California Technical Publishing, 1997.

[48] F. S. Panchal and M. Panchal, "Review on methods of selecting number of hidden nodes in artificial neuralnetwork," *International Journal of Computer Science and Mobile Computing*, vol. 3, no. 11, p. 455–464, Nov 2014. [Online]. Available: http://www.ijcsmc.com/docs/papers/November2014/V3I11201499a19.pdf

[49] A. Abraham, "Meta-learning evolutionary artificial neural networks," *CoRR*, vol. cs.AI/0405024, 2004. [Online]. Available: http://arxiv.org/abs/cs.AI/0405024

[50] A. Gherzan, "Hardware specific bsp overlay for the RaspberryPi device," 2012-2016, version master. [Online]. Available: http://git.yoctoproject.org/cgit/cgit.cgi/meta-raspberrypi/about/

[51] Adafruit, "Adafruit_Sensor," 2013-2016, version 1.0.2. [Online]. Available: https://github.com/adafruit/Adafruit_Sensor

[52] ——, "Adafruit_LSM303DLHC," 2013-2016, version 1.0.3. [Online]. Available: https://github.com/adafruit/Adafruit_LSM303DLHC

[53] ——, "Adafruit_L3GD20_U," 2013-2016, version 1.0.1. [Online]. Available: https://github.com/adafruit/Adafruit_L3GD20_U

[54] ——, "Adafruit_9DOF," 2014-2016, version 1.0.0. [Online]. Available: https://github.com/adafruit/Adafruit_9DOF

[55] "Openembedded core layer." [Online]. Available: http://cgit.openembedded.org/openembedded-core/

[56] D. Wang, J. Liao, Z. Xiao, X. Li, and V. Havyarimana, "Online-SVR for vehicular position prediction during GPS outages using low-cost INS," in *Personal, Indoor, and Mobile Radio Communications (PIMRC), 2015 IEEE 26th Annual International Symposium on*, Aug 2015, pp. 1945–1950.

[57] Y. Arias Esquivel, "Sistema de tiempo real para el análisis de vibracion en puentes," Ph.D. dissertation, Tecnológico de Costa Rica, 2015. [Online]. Available: http://repositoriotec.tec.ac.cr/handle/2238/7127