Reducción de Errores de Odometría en un Robot Móvil utilizando Algoritmos de Scan Matching basados en Sensores de Rango

Eduardo Aparicio Cárdenes

Escuela de Ingeniería Informática
Universidad de Las Palmas de G.C.

Proyecto fin de carrera

Título: Reducción de Errores de Odometría en un Robot Móvil utilizando Algoritmos de Scan Matching basados en Sensores de Rango

Apellidos y nombre del alumno: Aparicio Cárdenes, Eduardo

Fecha: Julio del 2011

Cotutor: Antonio Carlos Domínguez Brito

Cotutor: Antonio Falcón Martel

Agradecimientos.

Este proyecto no podría haberse realizado sin la dedicación especial de mi familia que han realizado la labor de soporte para su consecución. Manteniendo una postura paciente y motivadora llevando a buen puerto este fantástico proyecto.

En segundo lugar quería agradecer a mi cotutor Antonio Domínguez Brito, las horas empleadas en apoyar mi labor y hacer hueco en momentos difíciles para llevar una labor más allá de la obligación profesional y llenando de entusiamo el desarrollo, incluso en ocasiones superior al mío.

Por otro lado, quería dar las gracias al cotutor Antonio Falcon Martel que su carácter abierto siempre me hizo interesarme por la disciplina que imparte y consecuentemente por este proyecto fin de carrera.

Seguidamente, me siento plenamente agradecido a Luis Alvarez que me enseño una lección vital para mi fututo "Los obstáculos están siempre presentes en nuestra vida, nosotros elegimos a que altura ponerlos", siempre con actitud agradable pero firme me hizo sudar para lograr centrar mi futuro. Por este motivo, aprovecho la ocasión para felicitarle y que no cambie nunca su manera de proceder.

Finalmente, quiero dedicarme un especialmente a mi mismo que a pesar de haber habido momentos muy duros, he sabido seguir adelante y luchar por finalizar esta etapa de mi formación.

Índice general

Índice de figuras

Capítulo 1

Introducción

El MbICP es una variante del algoritmo ICP ("Iterative Closest Point")[2]. Este método tiene como objetivo corregir el error de odometría producido por errores externos que no se ven reflejados en los codificadores del Robot como son: deslizamientos en las ruedas, errores de redondeo, etc. Para ello hace uso de las mediciones de un sensor de rango (laser) cuyas medidas tienen una alta precisión dentro de su rango de funcionamiento dando un nivel de ruido despreciable, que favorece la acotación del problema.

Seguidamente, pasaremos a describir sin entrar en detalle el algoritmo ICP, el algoritmo MbICP y su funcionamiento, para tener una visión global del sistema y dar pie a describir los objetivos académicos y específicos que se han alcanzado en la realización de este proyecto. Para cerrar la sección introductoria con la estructura del documento con el fin de organizar y presentar rápidamente qué encontrarán en los próximos capítulos de este trabajo.

1.1. Un poco de historia, ICP.

El algoritmo ICP [2] ("Iterative Closest Point") se utiliza para encontrar el mejor "encaje" entre dos conjuntos de puntos en el espacio (éste es el problema del encaje de barrido ó scan-matching). Este algoritmo es un método orientado a minimizar la diferencia existente entre dos conjuntos de puntos. Algúnas de las implementaciones son: la reconstrucción 3-D de superficies para diferentes conjuntos de muestreos dados, la localización robots en el espacio, la planificación y trazado de caminos, el correlacionado de modelos de huesos, etc.

Una característica de este método es que siempre converge monótonamente al mínimo local más cercano de distancia cuadrática. En suma, posee experimentalmente un rápido ritmo de convergencia a las pocas iteraciones. Por todo ello, este método se utiliza comúnmente en tiempo real, permitiendo revisar de forma iterativa la transformación (translación y rotación), necesaria para reducir al mínimo la distancia entre los puntos de dos exploraciones o muestreos con sensores de rango.

Entradas del algoritmo:

- Dos barridos directos del sensor de rango, en dos instantes de tiempo distintos.
- Una estimación inicial de la transformación, es decir, estimar el desplazamiento del robot/sensor (translación y rotación).

- Un criterio para detener las iteraciones del algoritmo.

Salida resultante:

- Transformación final, es decir, desplazamiento estimado del robot (translación y rotación).

En esencia, el algoritmo consta de los siguientes pasos:

1. Asociar los puntos por el criterio del vecino más cercano.
2. Estimar los parámetros de transformación mediante una función cuadrática de coste medio.
3. Transformar los puntos usando los parámetros estimados.
4. Iterar (volver a aplicar los pasos 1, 2 y 3) hasta que se cumpla el criterio de parada.

Este algoritmo hace uso de la distancia Euclídea para establecer las correspondencias y aplicar mínimos cuadrados en el **paso 2**.

Sin embargo, como punto negativo, el uso de la distancia Euclídea limita el conjunto de soluciones porque no se toma en cuenta la rotación en la distancia a la hora de evaluar la correspondencia ente los puntos.

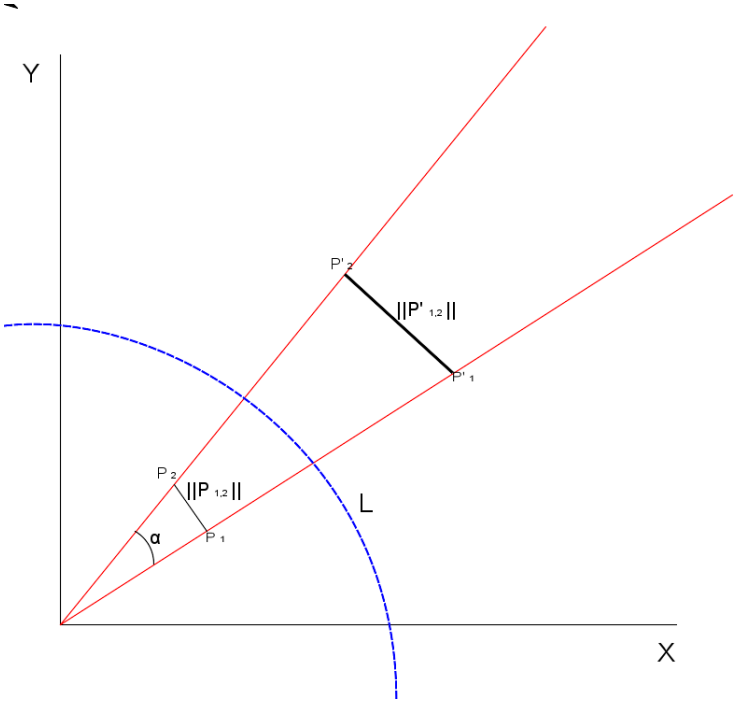


Figura 1.1: Umbral donde L cobra importancia

Por ello, aquellos puntos mas alejados del sensor pueden no tener una adecuada correspondencia en el espacio de soluciones a la hora de calcular las correspondencias debido a que no se considera la rotación del sensor, como se ilustra en la figura 1.1, en la evaluación de los puntos y consecuentemente, no puede expresarse claramente dichas asociaciones en el movimiento. Este problema es el que trara de atenuar el algoritmo MbICP.

1.2. ¿En qué consiste el algoritmo MbICP?

El algoritmo MbICP esta basado en la filosofía del algoritmo ICP, cuya diferencia esencial radica en un nuevo concepto de distancia llamada distancia MbICP. Introduciéndolo en los robots móviles para aproximar la posición del vehículo en el espacio, cálculo de obstáculos, búsqueda de objetos, etc. La idea básica consiste en emplear las capturas del soporte sensorial del robot para calcular un mapeo entre el nuevo y anterior barrido del sensor de rango, permitiendo así hallar el desplazamiento del robot en el espacio.

En este caso el soporte es un sensor de rango y se considera su error de medida despreciable en superficies planas. Esto nos permite partir de la idea que entre dos capturas del sensor de rango tenemos información suficiente para estimar la posición del nuevo sistema de referencia[1] en cada instante y deducir así cual es el error producido por los codificadores (odometría del robot) y eliminarlo.

Por tanto, este algoritmo obtiene en cada dos lecturas brutas consecutivas, un mapeo sensorial mediante el encaje de cojuntos de puntos del sensor de rango, buscando la configuración que minimice el error producido por la odometría . Con ello estimaremos el valor real del desplazamiento del robot, basándonos en la premisa de que los objetos a priori son estáticos.

Una de las principales diferencias entre los algoritmos existentes es el uso o no de entidades de alto nivel como rectas y planos. Por un lado, en un ambiente estructurado uno puede asumir la existencia en el entorno de formas poligonales[4][5][6]. Estos métodos son rápidos y trabajan muy bien dentro entornos cerrados. Sin embargo, limitan su ámbito de aplicación a la extracción de características geométricas, que no están siempre disponibles en entornos no estructurados.

Por tanto para mejorar los inconvenientes del algoritmo ICP que limita el conjunto de soluciones antes comentado, puesto que la distancia Euclídea no tiene en cuenta la rotación del sensor. El algoritmo MbICP propone el uso de una distancia diferente que introduce un término regular a la distancia Euclídea, el término $\theta^2 L^2$, como veremos.

Consecuentemente, se trata esta de una nueva medida de distancia en el espacio de configuración del sensor que toma en cuenta al mismo tiempo la translación y la rotación compensando simultáneamente. En consecuencia, [1] muestra que los resultados mejoran respecto a los métodos anteriores en términos de robustez, precisión, convergencia y carga computacional.

1.3. Funcionamiento.

Las precondiciones que debe cumplir el algoritmo para poder ejecutar el proceso son:

- Z_{ref} es el sistema de referencia obtenido del codificador de posición en el instante i .
- Z_{new} es el sistema de referencia obtenido del codificador de posición en el instante $i+1$.

[1] Los robots móviles, en ciertos casos, usan estas técnicas para estimar el desplazamiento.

- q_0 es la estimación del desplazamiento entre las dos lecturas ("scans") consecutivas del sensor en dos instantes de tiempo consecutivos i e $i+1$, relación inicial entre Z_{new} y Z_{ref}.
- $p_i \epsilon Z_{ref} \mid i = \{ \ 1 \quad \dots \quad n \ \}$ donde n es el número de muestras capturado por el sensor de rango en Z_{ref}.
- $r_i \epsilon Z_{new} \mid i = \{ \ 1 \quad \dots \quad m \ \}$ donde m es el número de muestras capturado por el sensor de rango en Z_{new}.

El funcionamiento de este algoritmo consta de tres fases destacadas:

1. Para cada punto p_i en Z_{ref} calculamos los puntos más próximos en Z_{new} (Transformado el sistema de referencia Z_{ref} usando la estimación q_k que inicialmente es q_0) cuya distancia es menor que una distancia dada d_{min}:

$$c_i = \underset{r_j \epsilon Z_{new}}{argmin} \quad \{d(p_i, q_k(r_j)) \ and \ d(p_i, q_k(r_j)) < d_{min}\} \tag{1.1}$$

   donde $d(p_1, p_2)$ es la medida de distancia. El resultado es un vector de l correspondencias

$$C = \{(p_i, c_i) \mid i = 1 \dots l\}$$

2. A continuación calculamos la estimación de la transformación q_{min} que minimiza el error cuadrático medio entre los pares de correspondencias del paso previo:

$$E_{dist}(q) = \sum_{i=1}^{l} d(p_i, q(c_i))^2 \tag{1.2}$$

   donde q es la incógnita de la ecuación que se pretende minimizar haciendo uso de mínimos cuadrados como explicaremos más adelante. Los puntos p_i y $c_i$son las correspondencias encontradas en el paso previo que verifica la ecuación (1.1), que son un subconjunto de las lecturas de rango en Z_{ref} y Z_{new} (c_i son las lecturas en Z_{new} no han sido transformadas por la transformación $q_k(r_j)$ aplicada en el paso 1.).

3. Ahora comprobamos que $q_{sol} = q_{min} \oplus q_k$. Si converge la estimación q_{sol} siendo inferior a un cierto umbral hemos encontrado la solución, en otras caso, es necesario iterar nuevamente con $q_{k+1} = q_{sol}$.

Se puede observar la similitud con el nucleo del algoritmo ICP. En la sección (2.2)veremos como el elemento diferencial es la distancia utilizada por el algoritmo.

1.4. Objetivos.

Los objetivos generales de un Proyecto Fin de Carrera (PFC) son principalmente que el alumno ponga en práctica los conocimientos adquiridos durante la carrera, aplicándolos a un trabajo real y completo. Además aparecen los objetivos específicos del proyecto en cuestión que dirigen el desarrollo del mismo. Así pues, en esta sección se presentan los objetivos divididos en dos apartados, objetivos académicos y objetivos específicos del PFC.

1.4.1. Objetivos Académicos.

Como objetivos académicos en la realización de este proyecto se plantean los siguientes:

- Seleccionar y aplicar una metodología de desarrollo de software.
- Aplicar los conocimientos de programación orientada a objetos.
- Aplicar los conocimientos de programación prodecimental, cálculo numérico intensivo y analisis de grandes volúmenes de datos.
- Aplicar conocimientos de asignaturas relacionadas con los sistemas robóticos móviles.
- Manejo de software de control de versiones.
- Manejo de herramientas de producción documental.

1.4.2. Objetivos específicos.

El objetivo inicial de este proyecto es el estudio, análisis e implementación del algoritmo MbICP para la reducción de errores de odometría mediante sensores de rango. Tomando como referencia el artículo [1] se tiene como objetivo inicial de la primera fase del desarrollo, el estudio del comportamiento del algoritmo sobre Matlab [26]. En base a los resultados existe una segunda fase, su implementación sobre la plataforma CoolBOT [24] para su puesta en funcionamiento en el entorno de ejecución a tiempo real.

Comenzaremos por describir en detalle **el primer objetivo**, que consistía en el desarrollo de un entorno sencillo con un sistema de gráficos simples a fin de visualizar los datos de la simulación y poder verificar la eficacia del método propuesto por el artículo [1]. No obstante, rápidamente se observo que esta propuesta impedía su correcto estudio y posterior análisis.

Por este motivo, se amplió el objetivo inicial de un pequeño programa de pruebas a **un simulador en Matlab capaz de estudiar y analizar grandes volúmenes de información extraídos de ficheros tras la ejecución de los sistemas robóticos en entornos reales, junto un interfaz gráfico capaz de transitar adecuadamente el estudio, depuración, análisis e interpretación de los datos sensibles y de los resultados del algoritmo.** Este objetivo pretendía elaborar resultados que justifiquen la adicción de este algoritmo dentro de la plataforma CoolBOT como sistema corrector de errores odométricos.

Una vez cumplido el primer objetivo y elaborado un informe sobre la correcta y eficaz mejora que supone el uso del algoritmo MbICP. **El segundo objetivo pretende integrar el algoritmo MbICP dentro de entorno de producción del sistema CoolBOT**, para incrementar la eficacia y robustez del mismo. Para su correcta realización es necesario:

- Contruir un componente para el algoritmo MbICP.
- Construir su vista o interfaz de control para dicho componente.
- Construir paquetes para comunicar la vista con el componente.
- Crear una integración para el sistema CoolBOT.
- Evaluar y verificar el correcto funcionamiento del Componente, la vista y los paquetes de datos que requiera para su comunicación.

1.5. Estructura del documento.

Este documento se organiza en 7 capitulos y 5 apéndices:

- **Capítulo 1:** Esta introducción
- **Capítulo 2:** En este capítulo se estudia el algoritmo MbICP, para ello comenzaremos describiendo la representación matemática elegida y seguidamente introduciremos el algoritmo MbICP describiendo su formulación y funcionamiento paso a paso. En este capítulo se hace especial enfasís en el entramado matemático que conforma el MbICP.
- **Capítulo 3:** Este capítulo describe como se ha llevado a cabo el desarrollo del proyecto. Nos centramos en el paradigma software elegido a la hora de llevar a cabo el desarrollo, las herramientas software que lo soportaron y en los pasos que se han seguido para tener el mejor plan de trabajo.
- **Capítulo 4:** Dentro de este capítulo describiremos el diseño elegido para el algoritmo MbICP. Haciendo mención especial a la gestión de persistencia para usar ficheros de datos externos.
- **Capítulo 5:** Se especifican los aspectos ligados al diseño de los principales elementos que intervienen en la elaboración del prototipo en Matlab.
- **Capítulo 6:** En este capítulo se describe el diseño en la plataforma CoolBOT. Comenzando por una breve descripción de como usar esta plataforma y continuaremos con la descripción del componente que contiene el algoritmo MbICP, así como los paquetes y la vista encargados de la comunicación y la visualización de los datos obtenidos en la simulación.
- **Capítulo 7:** Llegados a este punto del documento cerraremos la descripción del proyecto con los resultados y conclusiones extraídos de su desarrollo.
- **Apéndice A:** El manual de usuario de como usar el prototipo en Matlab se hace importante, para futuras personas que deseen usar esta aplicación de forma satisfactoria.
- **Apéndice B:** Este manual describe como poder usar e integrar los módulos que componen el desarrollo en la plataforma CoolBOT. Esta implementación se divide en el componente "MbICPCorrector", la vista "MbICP-Gtk" y el conjunto de paquete "MbICPPackets". Además se ha añadido la integración "MbICPIntegration" por si se deseará usar directamente.
- **Apéndice C:** En el capítulo 6 se habla del gestor de persistencia y la capacidad de soportar ficheros con distinto formato. En este anexo se han incorporado los formatos actualmente implementados para las aplicaciones Matlab y CoolBOT.
- **Apéndice D:** Este capítulo posee una descripción del algoritmo MbICP implementado sobre la plataforma Matlab.
- **Apéndice E:** En este capítulo detallamos la aplicación de evaluación implementada para realizar la comprobación del correcto funcionamiento del sistema.

Capítulo 2

Estudio del problema.

Durante todo este capítulo y en ciertas ocasiones a lo largo del documento vamos a hacer uso de una formulación matemática. Esta se ha extraído del libro Robot Analysis and Control[3] por su claridad y sencillez a la hora de expresar los cambios de coordenadas, transformaciones espaciales, etc. Vamos a introducir dicha nomeclatura a continuación.

2.1. Posición y orientación de un cuerpo rígido.

Nosotros podemos modelar un sistema robótico como un conjunto de cuerpo rígidos relacionados entre sí. La localización de cada cuerpo rígido esta completamente descrita por su posición y orientación.

La posición puede ser representada por las coordenadas de un punto fijo arbitrario, respecto al cuerpo rigido. Sea O_{xy} un sistema de coordenadas (SC) fijo y el punto O' un punto arbitrario del cuerpo rígido, como se muestra en la figura 2.1. Entonces la posición del cuerpo rígido se representa con relación al SC O_{xy} mediante

$$O_0 = \begin{pmatrix} x_o \\ y_o \end{pmatrix} \tag{2.1}$$

donde O_0 es un vector columna de $2x1$.

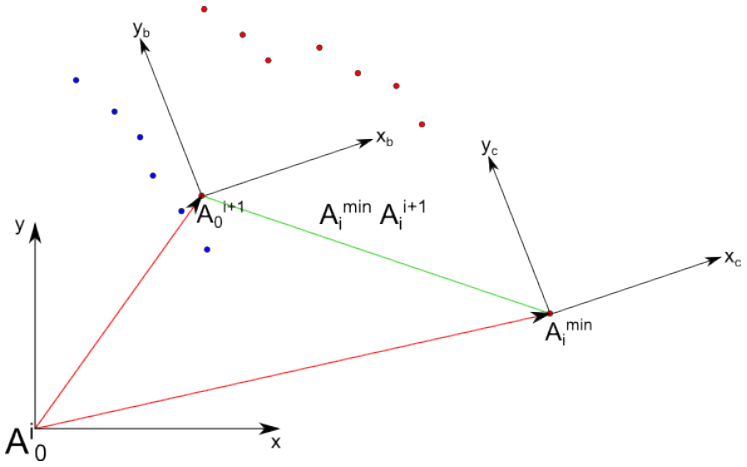


Figura 2.1: Posición y orientación de un cuerpo rígido

Para representar la orientación de un cuerpo rígido, dos ejes de coordenadas x_b, y_b han sido añadidos al cuerpo rígido mostrado en la figura. Estos ejes forman otro sistema de coordenadas, $O'_{x_b y_b}$ que se mueve solidariamente con O_{xy}. La orientación del cuerpo rígido entonces es representada por las direcciones de estos ejes de coordenadas. Sea **n** y **t** vectores unitarios apuntando en la dirección de los ejes de coordenadas, x_b y y_b, respectivamente. Los componentes de cada vector unitario son los cosenos directores de cada eje de coordenadas proyectado sobre el SC O_{xy}. Por conveniencia, nosotros vamos a combinar los dos vectores juntos y escribirlos usando una matriz de rotación llamada R:

$$R = [n, t] = \begin{bmatrix} n_x & t_x \\ n_y & t_y \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \tag{2.2}$$

La matriz R describe completamente la orientación del cuerpo rígido con referencia al eje de coordenadas $O_{x,y}$. Hay que tener en cuenta que las columnas de la matriz R son ortogonales entre sí.

$$n^T t = 0 \tag{2.3}$$

y poseen un módulo unitario

$$|n| = 1 \quad |t| = 1 \tag{2.4}$$

(donde $|a|$define la norma Euclídea del vector a). Una matriz en la que todos los vectores columna son ortonomales entre sí y cuya norma es la unidad, se la conoce como una matriz ortonormal (Se da que si R es ortonormal se cumple que $R^{-1} = R^T$).

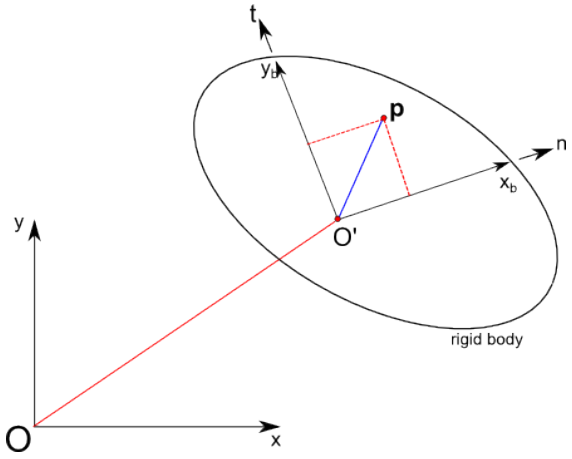


Figura 2.2: Transformación de coordenadas

2.1.1. Transformación de coordenadas.

Sea p un punto arbitraio en el espacio, como se muestra en la figura 2.4. Representamos la coordenada del punto p con referencia al eje O_{xy}, mediante:

$$\mathbf{p} = \begin{pmatrix} x \\ y \end{pmatrix} \tag{2.5}$$

La posición del punto p puede ser representada mediante las coordenadas obtenidas del cuerpo rígido, $O'_{x_b y_b}$, mediante:

$$\mathbf{p^b} = \begin{pmatrix} u \\ v \end{pmatrix} \tag{2.6}$$

El superíndice ^{b} indica que el vector esta definido con referencia a las coornadas del cuerpo rígido. Ahora vamos a encontrar la relacion entre los dos sistemas de coordenadas. Esta relacion define la transformación de coordenadas entre los ejes fijos y el eje de coordenadas del cuerpo rígido. La posición y orientación del cuerpo rígido, el cual es representado por el vector O_0 de dimensión $2x1$ y la matriz de $2x2$ llamada R en la sección previa, son usados para obtener la transformación de coordenadas. Como se muestra en la figura 2.4, el punto P puede ser expresado a través de O'

$$\overrightarrow{O\,p} = \overrightarrow{O\,O'} + \overrightarrow{O'p} \tag{2.7}$$

donde $\overrightarrow{O\,p} = p$ y $\overrightarrow{O\,O'} = O_0$. por tanto, podemos lo reescribir como

$$\overrightarrow{p} = \overrightarrow{O_0} + \overrightarrow{n} \cdot u + \overrightarrow{t} \cdot v = O_0 + u \begin{pmatrix} n_x \\ n_y \end{pmatrix} + v \begin{pmatrix} t_x \\ t_y \end{pmatrix} = O_0 + \overbrace{\begin{pmatrix} n_x & t_x \\ n_y & t_y \end{pmatrix}}^{R^b} \begin{pmatrix} n \\ v \end{pmatrix}$$

por tanto, tenemos que $\overrightarrow{p}$ es

$$\mathbf{p} = O_0 + R^b \mathbf{p}^b \tag{2.8}$$

La ecuación (2.8) proporciona la deseada transformación de coordenadas del cuerpo de coordenadas x^b al sistema de coordenadas x. Notar que esta transformación de coordenadas es dada en término de x_0 y R, los cuáles representan la posición y orientación del cuerpo rígido, o los cuerpos de coordenadas relativos al sistema transformado.

Ahora vamos a premultiplicar ambos lados de la ecuación (2.7) por la transpuesta R^T de la matriz R

$$R^T p = R^T O_0 + R^T R p^b \tag{2.9}$$

de (2.3) y (2.4), la matriz producto $R^T R$ por el lado derecho será

$$R^T R = \begin{bmatrix} n^T n & n^T t \\ t^T n & t^T t \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{2.10}$$

Por tanto, la ecuación (2.9) se reduce a

$$p^b = -R^T O_0 + R^T p \tag{2.11}$$

La ecuación (2.11) representa la transformación de coordenadas desde el eje de coordenadas transformado al origen de coordenadas, que es, la inversa de la transformación original (2.8). Por lo tanto, la tranformación inversa se obtiene simplemente transponiendo la matriz R.

Ejemplo sistema de coordenadas 2D

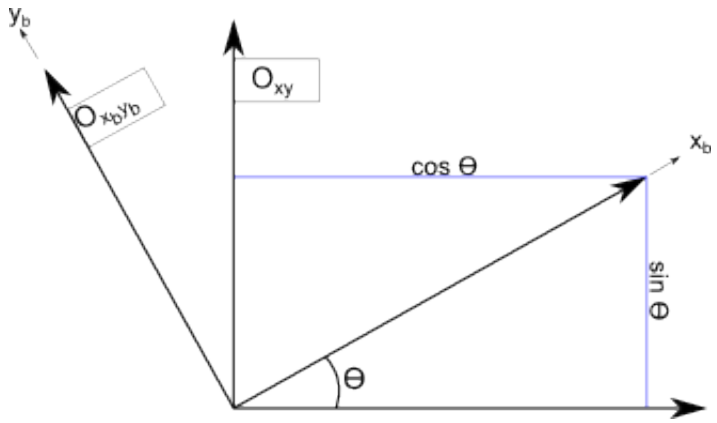


Figura 2.3: Ejemplo sistema coordenadas 2D.

Para ilustrar esta formulación vamos a describir con este ejemplo las tranformaciones de coordenadas para un sistema de coordenadas 2D. Como muestra la figura 2.3, dado un sistema de coordenadas $O'_{x_b y_b}$ que coíncide con el origen de coordenadas O_{xy}. El angulo entre los ejes x y

x_b están marcados por el ángulo $\theta = \angle xOx_b$. Buscamos el vector O_0 y la matriz R que representa la posición y orientación de $O'_{x_by_b}$ relativo a O_{xy}, y entonces obtenemos la transformación de coordenadas de $O'_{x_by_b}$ a O_{xy}.

En caso de que el origen de coordenadas de ambos sitemas coíncida, la posición O_0 es 0. Para obtener la matriz R, vamos a buscar dos vectores unitarios n y t, que componen R. Como se muestra en la figura 2.3, las componentes de cada vector están marcadas por los cosenos directores respecto a O_{xy}. Por tanto,

$$n = \begin{pmatrix} \cos\theta \\ \sin\theta \end{pmatrix} \quad t = \begin{pmatrix} -\sin\theta \\ \cos\theta \end{pmatrix}$$

tambien lo podemos expresar como

$$R = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \tag{2.12}$$

La transformación de coordenadas se obtiene sustituyento la matriz R y $O_0 = 0$ en la ecuación (2.8). La componentes de la transformación se expresan mediante

$$\begin{aligned} x &= u \cdot \cos\theta - v \cdot \sin\theta \\ y &= u \cdot \sin\theta + v \cdot \cos\theta \end{aligned} \tag{2.13}$$

Vamos a verificar los resultados anteriores. La figura 2.4 muestra dos ejes de coordenadas de un sistema bidimensional. El punto P'

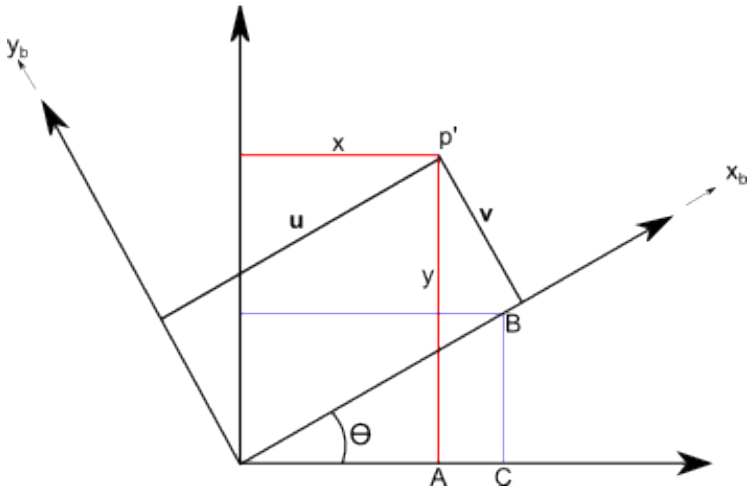


Figura 2.4: Transformación de coordenadas 2D

2.1.2. Transformaciones homogéneas.

En esta sección, desarrollaremos un método útil para representar las transformaciones de coordenadas de una forma compacta.

Recordemos la transformación de coordenadas dada por la ecuación (2.8):

$$p = O_0 + R^b p^b \tag{2.14}$$

El primer término de la parte derecha representa la transformación de translación, mientras el segundo término representa la transformación rotacional. El objetivo de esta sección es derivar una representación única de las transformaciones de coordenadas en las cuales ambas transformaciones de translación y rotación vengan dadas por una única matriz. Con este fin, vamos a definir los vectores $3x1$ en coordenadas homogéneas:

$$p = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad p^b = \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \tag{2.15}$$

y la matriz de $3x3$:

$$A^b = \left[\begin{array}{ccc|c} & R^b & & O_0 \\ & & & \\ \hline & 0 & & 1 \end{array}\right] \tag{2.16}$$

donde R equivale a

$$A^b = \left[\begin{array}{cc|c} \multicolumn{2}{c|}{\begin{pmatrix} n & t \end{pmatrix}} & O_0 \\ & & \\ \hline \multicolumn{2}{c|}{0} & 1 \end{array}\right] = \left[\begin{array}{cc|c} n_x & t_x & O_{0_x} \\ n_y & t_y & O_{0_y} \\ \hline 0 & 0 & 1 \end{array}\right] \tag{2.17}$$

Los vectores originales p y p^b son aumentados añadiendo un "1" como tercer elemento para que el resultado sea un vector de $3x1$. Tambien, la matriz de rotación R is extendida a una matriz de $3x3$ combinándola con el vector $3x1$ de posición O_0, con tres 0s y un 1 en la tercera fila. La ecuación (2.14) puede ser reescrita como

$$p = A^b p^b \tag{2.18}$$

que es,

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \left[\begin{array}{ccc|c} & R^b & & O_0 \\ & & & \\ \hline & 0 & & 1 \end{array}\right] \begin{bmatrix} x^b \\ y^b \\ 1 \end{bmatrix} \tag{2.19}$$

Se puede ver que la matriz de $3x3$, A^b representa la posición y orientación del sistema O_{x_b,y_b}. Los dos términos del lado derecho de la ecuación (2.14) son reducidos a un simple término en la ecuación (2.18). La transformación de coordenadas dada por la ecuación (2.18) se conoce como *transformación de coordenadas homogéneas*.

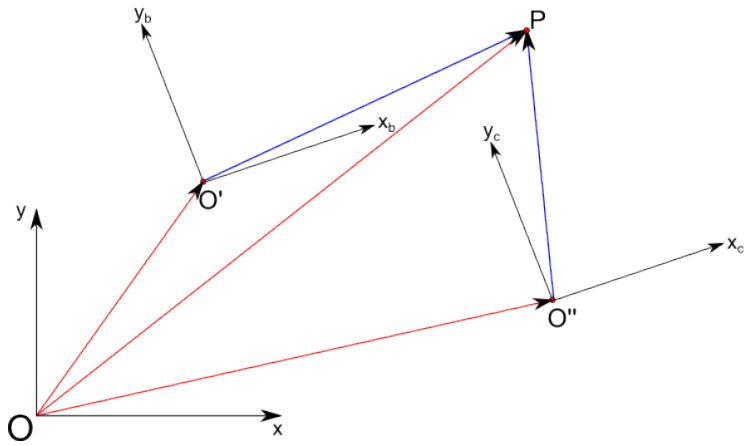


Figura 2.5: Transformaciones de coordenadas consecutivas.

La compactación de la transformación de coordenadas homogéneas és, particularmente ventajosa cuando se representan una serie de transformaciones consecutivas. Sea $O^n_{x_c,y_c}$ otro sistema de coordenadas, como se muestra en la figura 2.5, y p^c el punto expresado con referencia a $O^n_{x_c y_c}$. Entonces

$$p^b = O'_0 + R' p^c \tag{2.20}$$

donde O'_0 y R' son vectores de $2x1$ y matrices de $2x2$ asociadas con la transformación de coordenadas de p^c a p^b. Subtituyendo (2.20) en (2.14), obtenemos

$$p = O_0 + R p'_0 + R R' p^c \tag{2.21}$$

Hay ahora tres términos en el lado derecho de la ecuación (2.21). Como la transformación se repite, el número de términos en el lado derecho se incrementa. En general, n transformaciones de coordenadas consecutivas, dan lugar a un polinomio de grado n-ésimo que consiste en (n +1) términos no homogéneos. Las transformaciones homogéneas que usan las matrices de $3x3$, por otro lado, proporcionan una forma compacta de representar cualquier consecusión de transformaciones con un solo término. Considerando n transformaciones consecutivas desde el sistema n hasta el sistema 0. Sea A^{i-1}_i la matriz $3x3$ asociada con la transformación homogenea del sistema i al sistema $i-1$; entonces una posición del vector p^n en el sistema n es transformado a SC_0en el sistema 0 mediante

$$p^0 = A^1_0 \, A^2_1 \cdots A^n_{n-1} p^n \tag{2.22}$$

Por lo tanto las consecutivas transformaciones se compactan, descritas en un solo término.

Las matrices $3x3$ tienen otras dos propiedades equivalentes a las descritas anteriormente para las matrices de rotación. Una matriz $3x3$ representa la posición y orientación de un sistema de coordenadas. También representa la traslación y rotación del sistema de coordenadas. Por lo tanto, la propiedades de las matrices de rotación se mantienen para las matrices $4x4$, en las que ambas translaciones y rotaciones están involucradas.

2.2. Algoritmo MbICP.

Analizado el algoritmo ICP de donde proviene el algoritmo MbICP cuyo único cambio como verémos es la sustitución de la distancia Euclídea por un nuevo concepto de distancia. Este concepto aporta una medida de la distancia donde a partir de un cierto umbral que pondera los puntos más proximos angularmente con un mayor peso frente al resto de puntos.

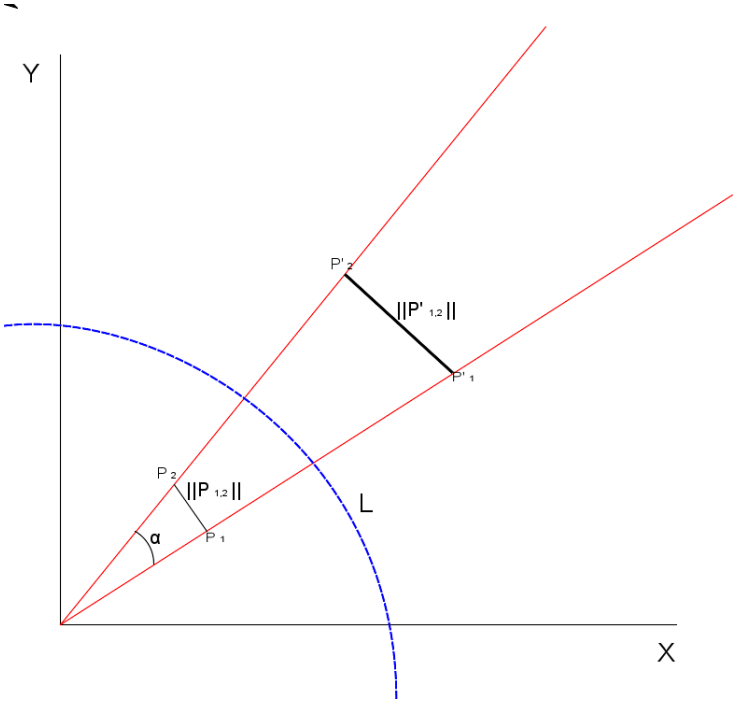


Figura 2.6: Umbral donde L cobra importancia

A continuación presentamos la formulación matemática del algoritmo así como alguna aclaraciones en su desarrollo. Con esto formalizamos las herramientas matemáticas que ha dado origen al algoritmo.

2.2.1. Distancia punto a punto.

Como se pudo observar en la sección1.2 es necesario reformular el concepto de distancia Euclídea para tener en cuenta la rotación del sensor. Sea, la transformación de un cuerpo rígido en el plano definida por un vector $q = (x, y, \theta)$ representando la posición y orientación entre $(-\Pi < \theta < \Pi)$ del sensor de rango en el plano. En base a esto se define la norma de q como:

$$\|q\| = \sqrt{x^2 + y^2 + L^2\theta^2} \tag{2.23}$$

Donde L es un número real positivo homogéneo para una longitud que expresa un radio de rotación angulary θ representa el ángulo de giro del sensor en cada instante. Dados dos puntos $p_1 = (p_{1x}, p_{1y})$ y $p_2 = (p_{2x}, p_{2y})$ en $\mathbb{R}^2$, definiremos una distancia entre p_1 y p_2 como la norma mínima para la transformación de SC que transforma de un punto a otro.

$$d_p(p_1, p_2) = min\{\|q\| \, tal \, que \, q(p_1) = p_2\} \tag{2.24}$$

Donde desarrollamos el término $\|q\|$

$$\|q\|^2 = \|p_2 - p_1\|^2 + L^2\theta^2 = (p_{2x} - p_{1x})^2 + (p_{2y} - p_{1y})^2 + L^2\theta^2$$

Por otro lado, se verifica que:

$$\begin{pmatrix} p_{2x} \\ p_{2y} \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} + R \cdot p_1 = \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} p_{1x} \\ p_{1y} \end{pmatrix}$$

Y ordenando un poco los términos

$$\begin{pmatrix} p_{2x} \\ p_{2y} \end{pmatrix} = q(p_1) = \begin{pmatrix} x + p_{1x}\cos\theta - p_{1y}\sin\theta \\ y + p_{1x} sin\theta + p_{1y}\cos\theta \end{pmatrix} \tag{2.25}$$

Fácilmente se puede comprobar que d_p es una distancia real que satisface para p_1 y o_2:

1. $d_p(p_1, p_2) = d_p(p_2, p_1)$ (distancia simétrica)
2. $d_p(p_1, p_2) = 0 \rightarrow p_1 = p_2$ (igualdad para distancia nula)
3. $d_p(p1, p3) \leq d_p(p_1, p_2) + d_p(p_2, p_3)$ (desigualdad triagular)

Vamos a linealizar esta expresión utilizando el teorema de Taylor en $\theta = 0$ hasta el 2^{o} término.

$$f(t) = f(t_0) + \sum_{i=1}^{\infty} \frac{f^{(i)}(t_0)}{i!}(t - t_0)^i \tag{2.26}$$

$$\cos\theta = \cos\theta_0 + \frac{\cancel{-\sin\theta_0}}{1}(\theta - \theta_0) = 1$$

$$\sin\theta = \cancel{\sin\theta_0} + \frac{\cos\theta_0}{1}(\theta - \theta_0)^1 = 0 + \theta$$

Así la ecuación 2.25 puede ser aproximada despues de la linealización como:

$$\begin{aligned} x + p_{1x} - \theta p_{1y} &= p_{2x} \\ y + \theta p_{1x} + p_{1y} &= p_{2y} \end{aligned} \tag{2.27}$$

El conjunto de soluciones para este sistema es infinito, (dado que tenemos tres incognitas y solo dos ecuaciones) y puede expresarse mediante:

$$\begin{aligned} x &= p_{2x} - p_{1x} + \theta p_{1y} \\ y &= p_{2y} - p_{1y} - \theta p_{1x} \end{aligned} \tag{2.28}$$

Donde θ es un parámetro para el conjunto de soluciones. Volviendo a usar la norma de la distancia (2.24), necesitamos encontrar la solución que minimice la norma de $q = (x, y, \theta)$. Para un θ dado, esta norma viene dada por la siguiente ecuación tras sustituir las expresiones de x e y de la ecuación (2.28) en la expresión de la ecuación de la norma (2.23):

$$\|q\|^2 = (\delta_x + \theta p_{1y})^2 + (\delta_y - \theta p_{1x})^2 + L^2\theta^2$$

donde $\delta_x = p_{2x} - p_{1x}$ y $\delta_y = p_{2y} - p_{1y}$. Expandiendo la expresión obtenemos un polinomio de segundo grado en θ:

$$\|q\|^2 = a\theta^2 + b\theta + c$$

donde $a = p_{1y}^2 + p_{1x}^2 + L^2, b = 2(\delta_x p_{1y} - \delta_y p_{1x})$ y $c = \delta_x^2 + \delta_y^2$, así pues estamos ante una ecuación de segundo grado que contiene dos soluciones que dependen del valor de a , como este es $a > 0$ implica que esta expresión tiene una única solución mínima para $\theta = -\frac{b}{2a}$ y el valor de este mínimo viene dado por

$$\|q\|^2 = \frac{-b^2 + 4ac}{4a} = \delta_x^2 + \delta_y^2 - \frac{(\delta_x p_{1y} - \delta_y p_{1x})^2}{p_{1y}^2 + p_{1x}^2 + L^2}$$

Finalmente la aproximación la distancia entre p_1 y p_2 es:

$$d_p^{ap}(p_1, p_2) \quad = \quad \sqrt{\delta_x^2 + \delta_y^2 - \frac{(\delta_x p_{1y} - \delta_y p_{1x})^2}{p_{1y}^2 + p_{1x}^2 + L^2}} \simeq d(\ p_1, \quad p_2\ ) = \|\ q\ \| \tag{2.29}$$

si sustituimos $\delta_x = p_{2x} - p_{1x}$ y $\delta_y = p_{2y} - p_{1y}$

$$d_p^{ap}(p_1, p_2) = \sqrt{\|P_2 - P_1\|^2 - \frac{\left((P_2 - P_1)\begin{bmatrix} P_{1y} \\ -P_{1x} \end{bmatrix}\right)^2}{\|P_1\|^2 + L^2}} = \sqrt{\|\triangle p_{12}\|^2 - \frac{\left((\triangle p_{12}^T)\begin{bmatrix} P_{1y} \\ -P_{1x} \end{bmatrix}\right)^2}{\|P_1\|^2 + L^2}} \tag{2.30}$$

donde $p_2 - p_1 = \triangle p_{12}$

2.2.1.1. Estudio de la distancia MbICP.

El algoritmo MbICP añade a la distancia Euclídea un nuevo término. En este apartado describiremos el porque se le ha añadido el termino $L^2\theta^2$ a la distancia Euclídea y que objeto tiene hacerlo. Para ello comenzaremos analizando el término final resultante de la linealización de la ecuación 2.23

$$\frac{(\delta_x p_{1y} - \delta_y p_{1x})^2}{p_{1y}^2 + p_{1x}^2 + L^2} \tag{2.31}$$

Este término define la solución de θ después de ser linealizada, bueno si retiramos temporalmente L^2de la ecuación podemos observar que el término inferior coincide con la norma cuadrada de P_1. A su vez la parte superior la podemos expresar como el producto escalar de $\triangle p_{12} = (p_2 - p_1)$ por el punto p_1', donde p_1' es $\begin{bmatrix} p_{1y} \\ -p_{1x} \end{bmatrix}$ que es un vector normal a p_1, de forma que obtenemos la siguiente expresión.

$$\frac{\left(\triangle p_{12}^T \cdot p_1'\right)^2}{\|p_1\|^2} = \frac{\left(\triangle p_{12}^T \cdot p_1'\right)}{\|p_1\|} \frac{\left(\triangle p_{12}^T \cdot p_1'\right)}{\|p_1\|} \tag{2.32}$$

Donde podemos intuir a simple vista la relación que guarda con el producto escalar donde $\triangle p_{12} \cdot p_1'$ es el producto escalar,

$$\left(\triangle p_{12}^T \cdot p_1'\right) = \begin{bmatrix} \delta_x & \delta_y \end{bmatrix} \cdot \begin{bmatrix} p_{1y} \\ -p_{1x} \end{bmatrix} = \delta_x p_{1y} - \delta_y p_{1x} \tag{2.33}$$

Si igualamos los términos obtenemos que

$$\left(\frac{\left(\triangle p_{12} \cdot p_1'\right)}{\|p_1\|}\right)^2 = \left(\frac{\|\triangle p_{12}\| \cancel{\|p_1'\|} \cos\alpha}{\cancel{\|p_1\|}}\right)^2 = \left(\|\triangle p_{12}\| \cos\alpha\right)^2 \tag{2.34}$$

Donde, α expresa la relación de ángulo entre p_2 y p_1', conociendo que el producto escalar es la proyección del punto p_2 sobre la normal a p_1 definida por p_1'. Por tanto, este término tendrá su máximo valor cuando el punto p_2 se encuentre sobre la normal a p_1 puesto que el ángulo que formarán p_2 y p_1' sera 0 . Analizando estos resultados podemos ver que tiene sentido pensar que los valores ubicados en la normal originan un valor inferior de distancia frente al resto, dado que el término resta de la distancia euclídea $\|p_2 - p_1\|$ como se describe en la ecuación (2.30)

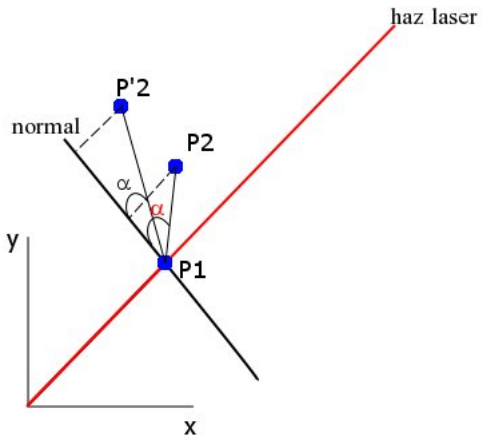


Figura 2.7: Relación entre los puntos del Haz y su normal

Obsérvese que si p_2 está sobre la normal a p_1' el ángulo $\alpha = 0$. Por tanto, el término de la ecuación 2.31 restará su máximo valor de la distancia, es decir, que dicho punto tendrá el mínimo valor posible para la distancia MbICP.

No obstante, cuando p_2 está alineado con p_1 y el origen, la distancia coíncidirá con la distancia Euclídea al ser $\alpha = \frac{\Pi}{2}$, es decir, el término de la ecuación 2.31 se hará 0.

Por ello, se puede observar en la figura 2.8 las iso-curvas de distancia que caracterizan la distancia MbICP donde el término de la ecuación 2.31 transforma la circunferencia de la distancia Euclídeas en un elipsoide. Las iso-curvas de distancia miden la distibución de los puntos más próximos al vértice v_i

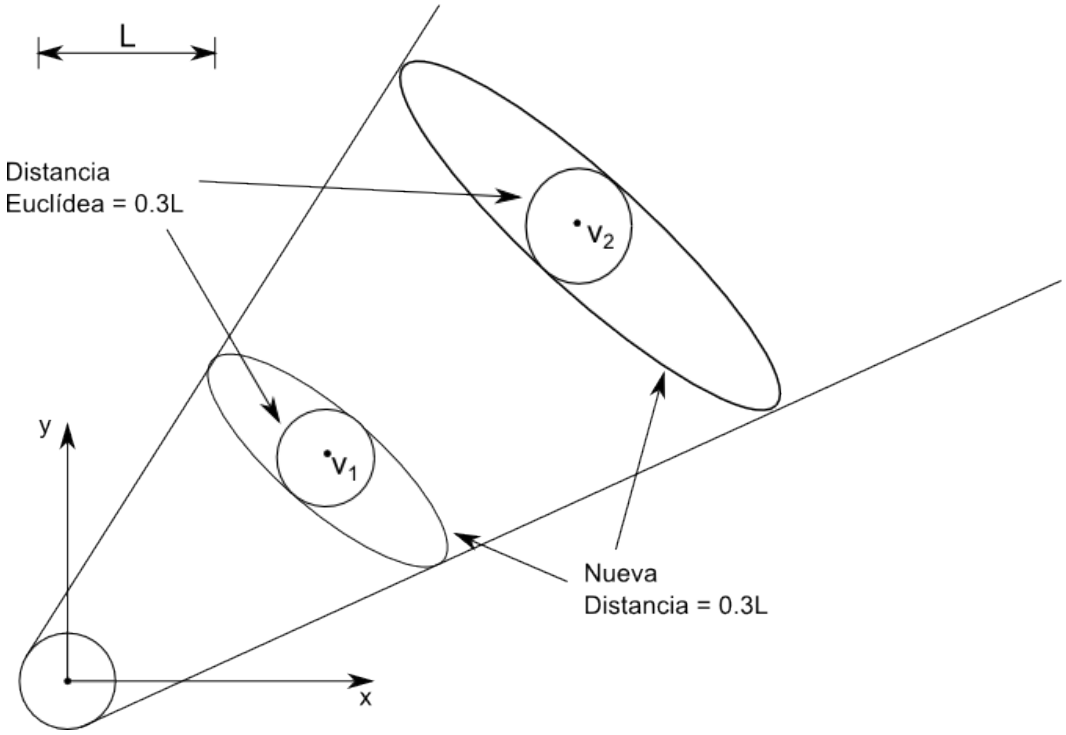


Figura 2.8: Curvas de distancia isométrica de d_p^{ap}para los puntos v_1y v_2

No obstante, este término ampliando la distancia euclídea es el encargado de favorecer aquellos puntos que están más próximos a la normal al vector p_1 frente de aquellos que no lo están como se muestra en la figura 2.7.

Por este motivo, aquellos puntos más próximos a la normal deben tener más peso que aquellos que se encuentran más alejados. El nuevo punto de acuerdo a los axiomas mencionados estará próximo a la normal que al haz que capturo el punto p_2. El motivo es simple, el robot se desplaza por el espacio a pequeños incrementos de posición frente al tiempo que se realiza cada muestreo sensorial, consiguiendo así una continuidad de valores, premisa necesaria para considerar válida la linealización de la ecuación 2.27.

Esto permite que utilizando este nuevo término, tener en cuenta la rotación del robot conforme se desplaza por el espacio, mejorando la seleción del conjunto de puntos más próximos entre sí,

como ilustra la figura 2.9.

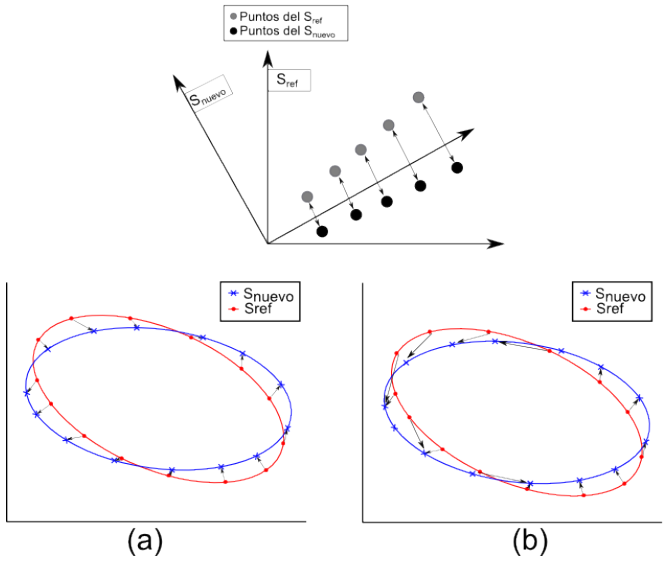


Figura 2.9: (Arriba) La distancia entre los mismos puntos llega a ser mayor en términos de distancia Euclídea con la aparición de un desplazamiento tras la rotación, el cual dificulta la asociación. (Abajo) Una elipse rotada. (a) Las asociaciones con la distancia euclídea no explican con claridad el movimiento de rotación, lo que afectaría a la convergencia. (b) Con la nueva distancia el movimiento de rotación es capturado.

¿Puede ser negativo el radicando de la expresión de la menor distancia MbICP de la ecuación de la ecuación 2.29? Bueno para responder a esta pregunta únicamente partiremos de la premisa

$$\frac{\left(\delta_x p_{1y} - \delta_y p_{1x}\right)^2}{p_{1y}^2 + p_{1x}^2 + L^2} > \delta_x^2 + \delta_y^2 \tag{2.35}$$

Si esta premisa se cumple significa que existe algún valor para el que el término añadido hace negativa la raíz, si llegamos a una inconsistencia significa que la distancia es válida.

$$\frac{\left(\triangle p_{12}^\top * p_1'\right)^2}{\|p_1\|^2 + L^2} > \|p_2 - p_1\|^2$$

$$\left(\triangle p_{12}^\top * p_1'\right)^2 > \|p_2 - p_1\|^2 \left(\|p_1\|^2 + L^2\right)$$

$$\left(\cancel{\|\triangle p_{12}\|} * \|p_1\|^2 * \cos\alpha\right)^2 > \cancel{\|\triangle p_{12}\|^2} \left(\|p_1\|^2 + L^2\right)$$

$$(\cos\alpha)^2 > \frac{\left(\|p_1\|^2 + L^2\right)}{\|p_1\|^2} \rightarrow (\cos\alpha)^2 > 1 + \frac{L^2}{\|p_1\|^2}$$

Podemos ver que el coseno cuadrado oscila entre 0 y 1 siempre. Esto da lugar a una inconsistencia que valida el hecho de que el interior de la raíz nunca podrá hacerse negativo.

2.2.1.2. Calibración del parámetro L.

Esta constante se calibra experimentalmente y posee unidades de medida. Por tanto, cuando evaluamos el comportamiento del algoritmo MbICP para distintos valores de L podemos percatarnos que los valores pequeños provocan efectos indeseados y los valores excesivamente elevados restan importancia al termino $L^2\theta^2$ provocando que este pierda su contribución.

$$d_p^{np}(p_1, p_2) = \sqrt{\|\triangle_p\|^2 - \frac{\|\triangle p\|^2 \|P_1\|^2 \cos^2\alpha}{\|p_1\|^2 + L^2}} \tag{2.36}$$

desarrollando el interior de la ecuación 2.36

$$\frac{\|\triangle p_{12}\|^2 \left(\|p_1\|^2 + L^2\right) - \|\triangle p_{12}\|^2 \|P_1\|^2 \cos^2\alpha}{\|p_1\|^2 + L^2} = \frac{\|\triangle p_{12}\|^2 L^2 + \|\triangle p_{12}\|^2 \|P_1\|^2 \left(1 - \cos^2\alpha\right)}{\|p_1\|^2 + L^2} =$$

$$\frac{\|\triangle p_{12}\|^2 L^2 + \|\triangle p_{12}\|^2 \|P_1\|^2 \sin^2\alpha}{\|p_1\|^2 + L^2} = \frac{\|\triangle p_{12}\|^2 \|P_1\|^2 \sin^2\alpha}{\|p_1\|^2 + L^2} + \frac{\|\triangle p_{12}\|^2 L^2}{\|p_1\|^2 + L^2}$$

por tanto, obtenemos que

$$d_p^{np}(p_1, p_2) = \sqrt{\|\triangle p_{12}\|^2 \frac{L^2 + \|P_1\|^2 \sin^2\alpha}{\|p_1\|^2 + L^2}} \tag{2.37}$$

por consiguiente vemos en la expresión 2.37 la influencia de L sobre la ecuación 2.29 es:

- Si $L \gg \|P_1\| \Rightarrow d(p_1, p_2) \simeq \sqrt{\|\triangle p_{12}\|^2} = \|\triangle p_{12}\|$
- Si $L \ll \|P_1\| \Rightarrow d(p_1, p_2) \simeq \|\triangle p_{12}\| \sqrt{\frac{L^2 + \|P_1\|^2 \sin^2\alpha}{\|p_1\|^2 + L^2}}$

Por tanto, veamos que sucede cuando tenemos un valor de L bajo por ejemplo $3\,mm$ frente a puntos de 2 ordenes superiores en magnitud.

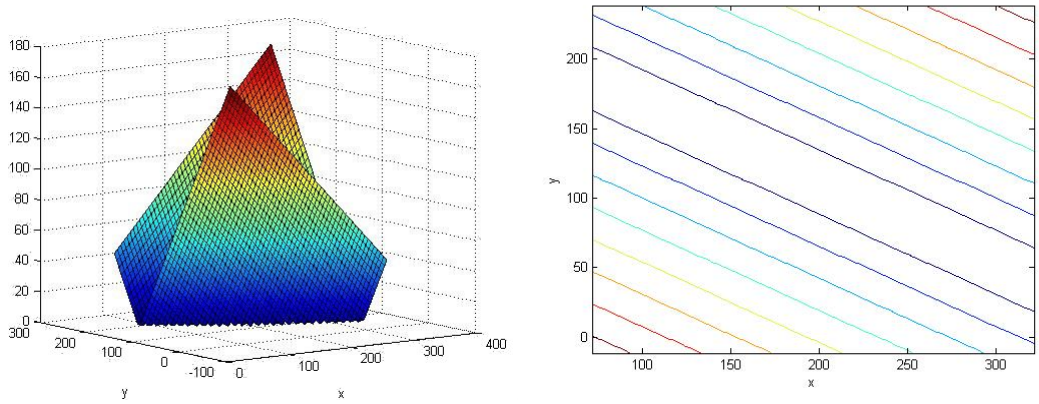


Figura 2.10: Distancia $L = 3mm$ donde $p_1 = \begin{pmatrix} 110mm \\ 200mm \end{pmatrix}$

En la figura 2.10 izquierda podemos observar que aquellos puntos que se encuentren en la normal tendrán una distancia menor frente aquellos puntos que forman un ángulo con ella, aunque este sea mínimo. Además podemos ver que el contorno de la distancia en la figura 2.10 derecha que es imposible distinguir las elípses que forman los puntos para una misma distancia.

Por consiguiente, todos los valores dentro de la normal son considerados prácticamente iguales haciendo difícil evaluar cual de ellos está más próximo al punto p_1. Observándose en algunos casos que puntos que en teoría se encuentra más alejados tienen una distancia inferior puesto que su proyección sobre la normal da un valor más grande restando más a la distancia euclídea.

Sin embargo, cuando hacemos esta $L \to \infty$ la función pasa a ser la distancia euclídea, provocando que se evalúe en función de esta.

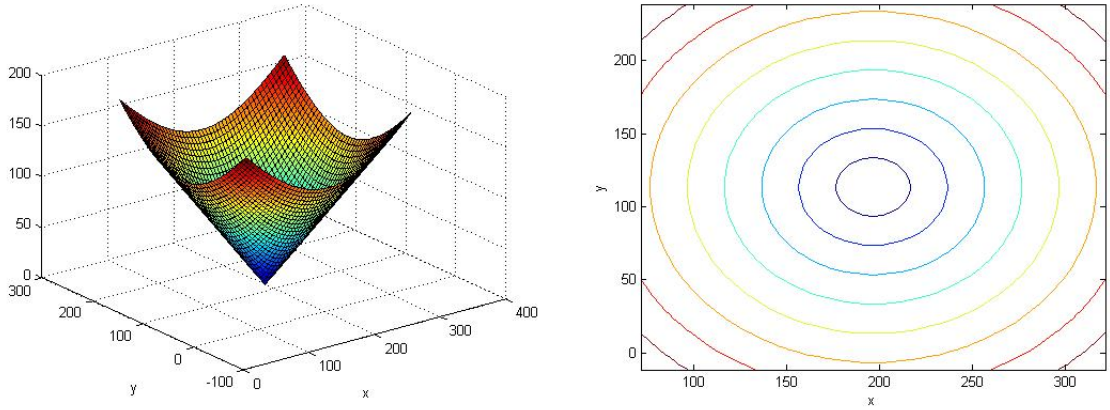


Figura 2.11: Distancia cuando $L \to \infty$ donde $p_1 = \begin{pmatrix} 110mm \\ 200mm \end{pmatrix}$

Donde en la figura 2.11 ambas figuras se puede observar las características de la distancia euclídea con isolíneas regulares en su contorno y valores asindóticamente decrecientes en torno al punto sobre el que se evalúa la distancia. Eliminando con ello la contribución a la hora de capturar las correspondencias del comportamiento del sensor de rango en el robot.

Finalmente, cuando tratamos con valores de L en el intervalo $[100mm, 500mm]$ dan un buen comportamiento viéndose como los valores conforme se alejan de la normal poseen una mayor distancia, incluso dentro de la normal los puntos más alejados al vértice poseen un valor superior, como se puede ver en la figura 2.12 izquierda donde las líneas de isodistancia son en este no elípticas.

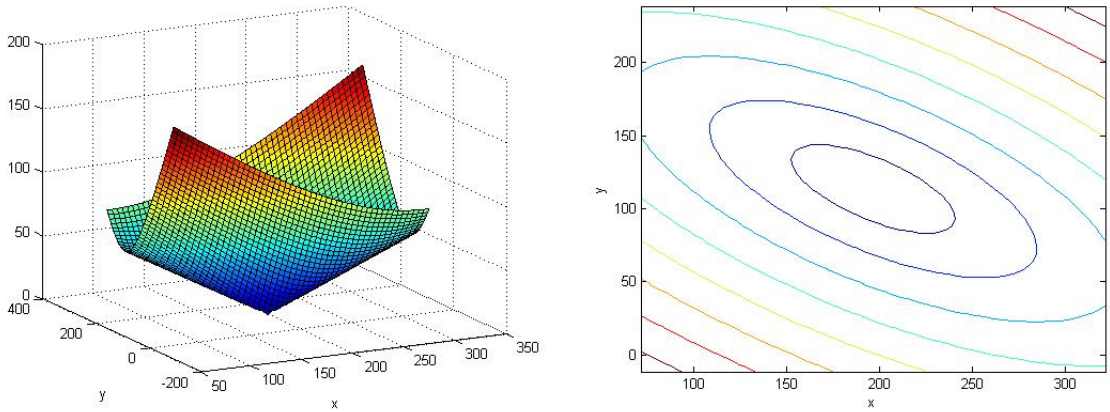


Figura 2.12: Distancia $L = 300mm = 0{,}3m$ donde $p_1 = \begin{pmatrix} 110mm \\ 200mm \end{pmatrix}$

A la luz de estos resultados podemos observar la importancia del parámetro L y su dependencia de las unidades de magnitud con las que estemos trabajando. Por ello, este valor dependerá del espacio en el que se este moviendo el robot, así como sus dimensiones y la proximidad a la que se encuentren los obstáculos.

2.2.2. Minimización por mínimos cuadrados.

El siguiente paso a la hora de computar el valor de q requiere que minimicemos por mínimos cuadrados [8] la expresión (1.2) pero en términos de la nueva distancia antes formulada. La expresión (1.2) con la distancia (2.25) nos lleva a:

$$E_{dist}(q) = \sum_{i=1}^{n} \left(\delta_x^2 + \delta_y^2 - \frac{(\delta_x p_{1y} - \delta_y p_{1x})^2}{p_{1y}^2 + p_{1x}^2 + L^2} \right) \tag{2.38}$$

Donde $d(p_i, q(c_i))^2 \equiv d_p^{a,p}(p_1, p_2)^2$ esto implica que $p_1 \to p_i$ y $p_2 \to q(c_i)$

$$\delta_x = p_{2x} - p_{1x} \equiv \delta_x = q(c_i)_x - p_{ix} = c_{ix} - c_{iy}\theta + x - p_{ix}$$

$$\delta_y = p_{2y} - p_{1y} \equiv \delta_y = q(c_i)_y - p_{iy} = c_{ix}\theta + c_{iy} + y - p_{iy}$$

Hay que tener en cuenta que $q(c_i)$ es la transformación (2.25) linealizada por Taylor tendiendo como resultado

$$x + c_{ix} - \theta c_{iy} = q(c_i)_x$$

$$y + \theta c_{ix} + c_{iy} = q(c_i)_y$$

La ecuación cuadrática (2.38) tiene la forma:

$$E_{dist}(q) = q^T A q + 2b^T q + c$$

Donde c es un número constante, A es una matriz simétrica

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{12} & a_{22} & a_{23} \\ a_{13} & a_{23} & a_{33} \end{pmatrix}$$

$a_{11} = \sum_{i=1}^{n} 1 - \frac{p_{iy}^2}{k_i}$
$a_{12} = \sum_{i=1}^{n} \frac{p_{ix} p_{iy}}{k_i}$
$a_{13} = \sum_{i=1}^{n} -c_{iy} + \frac{p_{iy}}{k_i}(c_{ix} p_{ix} + c_{iy} p_{iy})$

$a_{22} = \sum_{i=1}^{n} 1 - \frac{p_{ix}^2}{k_i}$
$a_{23} = \sum_{i=1}^{n} c_{ix} - \frac{p_{ix}}{k_i}\left(c_{ix}p_{ix} + c_{iy}p_{iy}\right)$
$a_{33} = \sum_{i=1}^{n} c_{ix}^2 + c_{iy}^2 - \frac{1}{k_i}\left(c_{ix}p_{ix} + c_{iy}p_{iy}\right)^2$
y

$$b = \begin{pmatrix} \sum_{i=1}^{n} c_{ix} - p_{ix} - \frac{p_{iy}}{k_i}\left(c_{ix}p_{iy} - c_{iy}p_{ix}\right) \\ \sum_{i=1}^{n} c_{iy} - p_{iy} + \frac{p_{ix}}{k_i}\left(c_{ix}p_{iy} - c_{iy}p_{ix}\right) \\ \sum_{i=1}^{n} \left[\frac{1}{k_i}\left(c_{ix}p_{ix} + c_{iy}p_{iy}\right) - 1\right]\left(c_{ix}p_{iy} - c_{iy}p_{ix}\right) \end{pmatrix}$$

Donde $k_i = p_{ix}^2 + p_{iy}^2 + L^2$. El valor de q que minimiza $E_{dist}(q)$ es

$$q_{min} = -A^{-1}b$$

En resumen, se ha particularizado la resolución del sistema de ecuaciones de mínimos cuadrados para n ecuaciones con 3 incógnitas. Para ello únicamente es necesario calcular el sumatorio para los n puntos correlacionados en el paso previo del algoritmo en (1.1) y resolver el producto matricial de q_{min}.

2.2.3. Aplicar q_k.

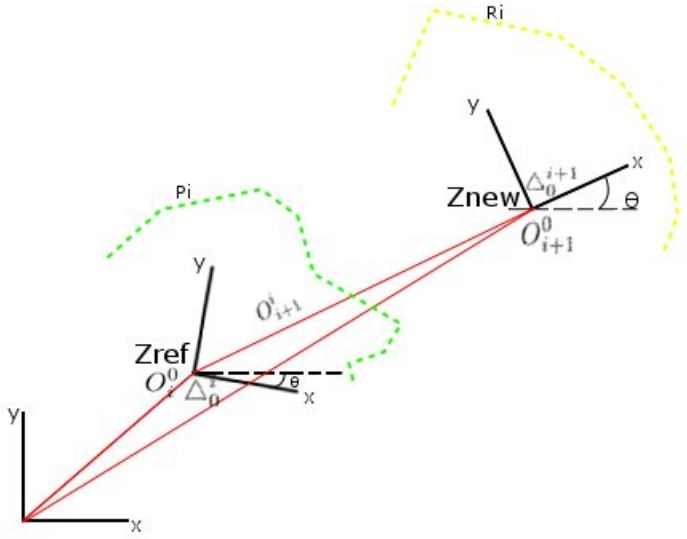


Figura 2.13: Relación geométrica entre el sistemas de coordenadas

A la hora de calcular los puntos más próximos entre sí es necesario ser conscientes de que el sistema de referencia ha cambiado desde el barrido i-ésimo con respecto al barrido $i+1$ porque el robot se encuentra en movimiento. Por tanto, antes de poner en funcionamiento el algoritmo MbICP es necesario aplicar la formulación matemática que a continuación enunciamos para trasladar los puntos del sistema de referencia $i+1$ al i

$$p^0 = O_0^{i+1} + R_0^{i+1} p^{i+1} = A_0^{i+1} p^{i+1} \tag{2.39}$$

$$p^i = O_i^0 + R_i^0 p^0 \tag{2.40}$$

Dadas una expresión en términos en el sistema de referencia origen, expresarlo en el sistema de coordenadas i sería

$$p^0 = O_0^i + R_0^i p^i \rightarrow p^0 - O_0^i = R_0^i p^i$$

$$p^i = \left(R_0^i\right)^{-1} \cdot \left(p^0 - O_0^i\right) = \left(R_0^i\right)^{-1} \cdot \left(p^0 - O_0^i\right) = R_i^0 \cdot \left(p^0 - O_0^i\right)$$

como$\left(R_0^i\right)^{-1} = \left(R_0^i\right)^T = R_i^0$ entonces

$$p = R_i^0 p^0 - R_i^0 O_0^i \equiv O_i^0 + R_i^0 * p^0 \rightarrow O_i^0 = -R_i^0 \cdot O_0^i$$

Una vez tenemos las expresiones que relacionan al SC $i+1 \rightarrow 0$ y al SC $0 \rightarrow i$ y viceversa, podemos ver que sustituyendo la ecuación 2.39 que transforma un punto del sistema de referencia $i+1$ a 0 en la ecuación 2.40que transforma del sistema 0 a i obtenemos una transformación de puntos $i+1$ a i y como consecuencia tenemos la siguiente ecuación:

$$p^i = R_i^0 \left(O_0^{i+1} + R_0^{i+1} * p^{i+1}\right) - R_i^0 O_0^i$$

$$p^i = R_i^0 \left(O_0^{i+1} - O_0^i\right) - R_i^0 R_0^{i+1} p^{i+1} \tag{2.41}$$

Con esto podemos definir $\left(O_i^{i+1}, \ R_i^{i+1} \right) \equiv \triangle_i^{i+1} \equiv q_k$ un SC que viene definido por la ecuación (2.41) que transforma todos los puntos del barrido del sensor desde el SC $i+1$ al SC i para poder operar correctamente con ellos dentro del algoritmo, como se puede ver en la siguiente figura el algoritmo tiene como precondición de todos los puntos de referencia estén expresados sobre el sistema de referencia i.

2.2.4. Cálculo de la solución (q_{sol}).

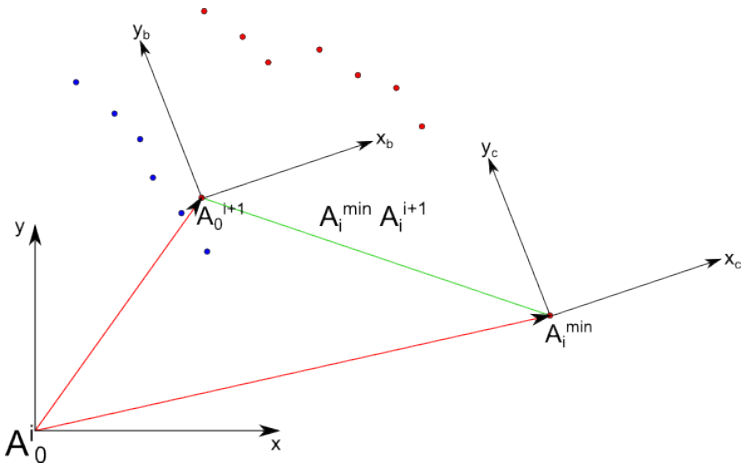


Figura 2.14: Relación entre A_k^{min}, A_i^{min} y A_i^k

A lo largo de cada ejecución obtenemos una aproximación de la correción que mejora la odometría proveniente del sensor odométrico (SC A_0^{i+1}). Esta solución tiene una estimación inicial de correción que es la siguiente:

$$q_0 \equiv^0 A_i^{i+1} = \left(A_0^i\right)^{-1} \cdot A_0^{i+1} \tag{2.42}$$

Conforme evoluciona el algoritmo cada iteración proporcina una estimación solución mejorada donde q_{min} es la correción sobre la odometría en cada instante.

1. $q_0 \Rightarrow q_1 \equiv^1 A_i^{i+1} =^1 \overbrace{A_i^{min}}^{q_{min}^1} .^0 A_i^{i+1}$
2. $q_1 \Rightarrow q_2 \equiv^2 A_i^{i+1} =^2 \underbrace{A_i^{min}}_{q_{min}^2} .^1 A_i^{i+1}$
3. $q_2 \Rightarrow q_3 \equiv^3 A_i^{i+1} =^3 \underbrace{A_i^{min}}_{q_{min}^3} .^3 A_i^{i+1}$

$\vdots$

k. $q_{k-1} \Rightarrow q_k \equiv^k A_i^{i+1} =^k \underbrace{A_i^{min}}_{q_{min}^k} .^{k-1} A_i^{i+1}$

Por tanto, despues de la k-ésima iteración se satisface que $q_{min}^k < q_{error}$, es decir, q_{min} es tan pequeño que no mejora la solución obtenida en la iteración $k-1$. Por ello concluímos que la solución es:

$$q_{sol} = q_k \equiv^k A_i^{i+1} =^k A_i^{min} .^{k-1} A_i^{i+1} \tag{2.43}$$

$$q_{sol} = q_k \equiv^k A_i^{min} .^{k-1} A_i^{min} .^{k-1} A_i^{min} \ldots . .^1 A_i^{min} .^0 A_i^{i+1}$$

2.2.5. Correción de odometría acumulada.

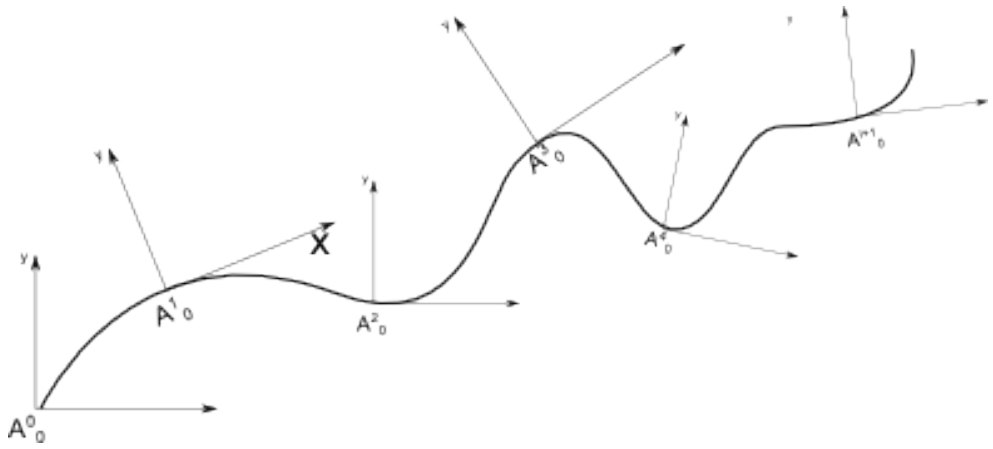


Figura 2.15: Evolución de la odometría durante la ejecución del robot

Dados los sistemas de coordenadas SC_1, SC_2, SC_3 y SC_4 verificados para las matrices homogéneas A_0^1, A_0^2, A_0^3 y A_0^4 respectivamente. Y sea p^1, p^2, p^3 y p^4 puntos en dichos SC respectivamente se cumple que:

$$^{sol}A_0^4 =^{sol} A_0^1 \cdot^{sol} A_1^2 \cdot^{sol} A_2^3 \cdot^{sol} A_3^4$$

y de forma genérica para un instante de tiempo i-ésimo de la ejecución del robot obtenemos que la solución es:

$$^{sol}A_0^i =^{sol} A_0^1 \cdot^{sol} A_1^2 \cdot^{sol} A_2^3 \cdot^{sol} A_3^4 \cdots^{sol} A_{i-1}^i$$

2.2.6. Acumulación del odómetro .

A la hora de desplazarse el robot vamos a contabilizar el número de metros recorridos durante la ejecución (el odómetro) se ha desplazado. En nuestro, caso particular se ha empleado para calcular cuanto se reduce el desplazamiento total con el uso del algoritmo MbICP dentro del planificador frente a la odometría directa procedente del robot.

Para hacernos una idea de que representa, esta distancia es la suma de todos los tramos parciales, donde cada tramo se mide como la distancia entre dos instantes de odometría consecutivos.

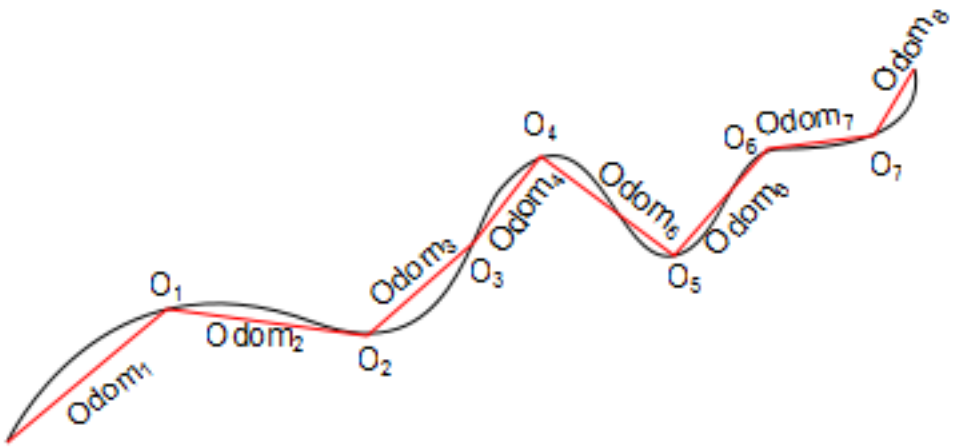


Figura 2.16: Segmentos acumulados en el cuentakilómetros

Donde

$$Odom_1 = \left\|\triangle_0^1\right\|, Odom_2 = \left\|\triangle_1^2\right\|, Odom_3 = \left\|\triangle_2^3\right\|, \cdots, Odom_n = \left\|\triangle_{n-1}^n\right\|$$

Cada uno de estos valores representa un tramo entre el SC i e $i+1$ para saber que valor que tiene el odómetro en un instante $i+1$, únicamente es necesario:

$$Odom_{i+1} = \sum_{n=1}^{i+1} Odom_n$$

o en nuestro caso añadirle al acumulado anterior el nuevo tramo recorrido

$$Odom_{i+1} = Odom_i + \left\|\triangle_i^{i+1}\right\|$$

Capítulo 3

Estudio de la plataforma CoolBOT.

A continuación pasamos a realizar un estudio del marco de programación CoolBOT. Sólo nos centraremos en las partes de dicha plataforma que son necesarias para el presente trabajo. Para un conocimiento más profundo de esta plataforma, se recomienda leer [Domínguez-Brito et al., 2003].

3.1. Orígenes de CoolBOT.

CoolBOT fue principalmente originado debido a consideraciones muy prácticas. A principios del año 2000, se observó la falta de una metodología sistemática de desarrollo de sistemas robóticos [9]. Tampoco se había definido una arquitectura software válida para estos sistemas.

Durante el desarrollo de varios sistemas robóticos por parte del grupo GIAS1 (Grupo de Inteligencia Artificial y Sistemas) se llegó a la conclusión que era necesario diseñar e implementar algún tipo de infraestructura software común que disminuyera los costes de desarrollo e integración. Esta infraestructura tenía que ser lo suficientemente genérica como para soportar cualquier arquitectura o esquema de control ideado para los proyectos en los que el grupo se encontraba involucrado en aquel momento. Al mismo tiempo, debía permitir integrar software de manera fácil.

Partiendo de estos requisitos, se diseño e implementó un marco software basado en componentes denominado CAV [10]. CAV fue una herramienta que permitió modelar software de control como redes de agentes software interconectados, y proporcionaba mecanismos de intercomunicación entre agentes, ya fueran locales o remotos. CAV carecía de muchos recursos y primitivas que se consideraron también necesarias, por ejemplo, un conjunto más rico de mecanismos de intercomunicación, y un suporte para programación multihilo (multithreading) más sistemática y menos propensa a errores. Pero sobre todo carecía de mecanismos que facilitaran la integración de software, que continuaba siendo un importante problema a resolver en los diferentes proyectos.

Trabajos y experiencias posteriores utilizando CAV llevaron al diseño y desarrollo de CoolBOT [11], pasando por diferentes fase que pueden seguirse en [12] y [13]. CoolBOT [Domínguez-Brito et al., 2004a] es un marco de programación C++ orientado a componentes donde el software que controla un sistema se ve como una red dinámica de unidades de ejecución interconectadas por medio de caminos de datos. Cada una de estas unidades de ejecución es un componente software, modelado como un autómata de puertos [14] [15] [12], que proporciona una funcionalidad dada, oculta tras un interfaz externo de puertos de entrada y salida que especifica claramente los datos que el componente consume, y cuáles produce. Cualquier componente, una vez es definido, cons-

truido y probado, puede instanciarse e integrarse tantas veces como se necesite en otros sistemas. CoolBOT proporciona la infraestructura necesaria para soportar este concepto de componente software, así como para que se intercomuniquen entre ellos mediante conexiones de puertos que puedan establecerse y desestablecerse dinámicamente.

3.2. Características de la plataforma CoolBOT.

La plataforma o framework CoolBOT plantea una infraestructura software que permita programar sistemas robóticos mediante ensamblaje e integración de componentes a modo de puzzle software. Provee de un potente entorno donde es posible sintetizar diferentes arquitecturas usando el mismo lenguaje de especificación. A continuación se enumeran los principios básicos de la plataforma:

- **Orientado a Componentes.** CoolBOT se concibe como una programación orientada a componentes, que se vale de un lenguaje de especificación de manipulación componentes como bloques de construcción con el fin de definir funcionalmente un sistema robótico completo mediante la integración de componentes.
- **Uniformidad entre Componentes.** Una aproximación basada en componentes claramente demanda cierto nivel de uniformidad entre componentes. Dentro de CoolBOT esta uniformidad se manifiesto en dos importantes aspectos:

  1. Autómata de Puertos. Se define un interfaz uniforme para todos los componentes, basado en el concepto de Autómatas de Puertos [14] [15] [10] que establece una clara distinción entre la funcionalidad interna de una entidad activa (el autómata) y su interfaz externo, los puertos de entrada y salida.
  2. Puertos por defecto. Todo componente debe ser observable y controlable en cualquier instante desde el exterior del propio componente. Para ello se establece una interfaz y estructura de control uniforme a todos los componentes para que todo componente facilita su observabilidad y su controlabilidad. Esta interfaz y estructura de control son los denominados puertos por defecto.

- **Robustez y Controlabilidad.** Un sistema robótico orientado a componentes será robusto y controlable si sus componentes son también robustos y controlables. Un componente se considerará robusto cuando:

  1. Sea capaz de observar su propio rendimiento, adaptándose a condiciones de operación cambiantes, e implementando sus propios mecanismos de adaptación y recuperación para tratar todo error que pueda ser detectado internamente (dentro el componente, robustez local).
  2. Cualquier error detectado por un componente que no pueda ser tratado y/o resuelto por sus propios medios, debería ser notificado utilizando algún mecanismo estándar a través de su interfaz externo (robustez externa), llevando al componente a un estado de inactividad total (idle) en el que se espera por una intervención externa, que consistirá, o bien, en reiniciar el componente, o en abortarlo. Las comunicaciones enviadas y recibidas por un componente cuando esta tratando/resolviendo excepciones deberían ser comunes a todos los componentes.

Adicionalmente, se considerará a un componente controlable cuando el pueda ser llevado bajo supervisión externa a través de su interfaz - por medio de un supervisor o un controlador - a lo largo de una trayectoria de control establecida. En orden a conseguir dicha controlabilidad externa, los componentes serán modelados como autómatas cuyos estados serán forzados por un supervisor externo, y compartiendo todos ellos la misma estructura en su autómata de control.

- **Modularidad y Jeraquía.** La arquitectura de un sistema robótico se definirá en CoolBOT utilizando componentes como unidades funcionales elementales. Como en casi cualquier marco basado en componentes, habrá unidades atómicas y compuestas. Un componente atómico será indivisible, es decir, uno que está formado/compuesto por otros componentes. Un componente compuesto será un componente que incluye en su definición a otros componentes, atómicos o no, y provee un supervisor para su observación y control. Con esta visión, un sistema completo no es nada más que un componente compuesto único, que a su vez incluye otros componentes, que análogamente incluyen otros, y así hasta que esta cadena de descomposiciones finaliza en algún componente atómico. Así pues, un sistema completo puede verse como una jerarquía de componentes desde un punto de vista de coordinación y control.

- **Distribuido.** La distribución de componentes sobre un entorno de computación distribuida es una necesidad fundamental en muchos sistemas de control. CoolBOT debería gestionar las comunicaciones entre componentes situados en la misma y/o en diferentes máquinas de forma que para un usuario de CoolBOT parecería que se realizaran exactamente de la misma forma.

- **Reutilización.** Los componentes son unidades que mantienen sus interioridades ocultas detrás de un interfaz uniforme. Una vez ellos han sido definidos, implementados y probados, podrían utilizarse como componentes integrantes de otros componentes o sistemas mayores. Los modernos sistemas robóticos están llegando a convertirse en sistemas realmente complejos, y muy pocos grupos de investigación tienen los recursos humanos necesarios para construir sistemas desde cero. Los diseño orientado a componentes representan una forma apropiada de aliviar esta situación. En nuestra opinión, la investigación en robótica podría beneficiarse enormemente de la posibilidad de intercambiar componentes entre laboratorios como un medio de validación cruzada de los resultados de investigación.

- **Completitud y Expresividad.** El modelo de computación subyacente en CoolBOT debería ser válido para construir arquitecturas muy diferentes para sistemas robóticos y ser lo suficientemente expresivo como para tratar la concurrencia, el paralelismo, la distribución y compartición de recursos, la respuesta en tiempo real, la existencia de múltiples y simultáneos bucles de control y de múltiples objetivos a satisfacer, de una forma estable y sistemática.

3.3. Modelado de componentes

Cada componente CoolBOT es modelado como un autómata con puertos de entrada y puertos de salida. Este modelo permite diferenciar claramente la funcionalidad interna del componente de la interfaz externa como puede verse en las figuras 3.1 y 3.2.

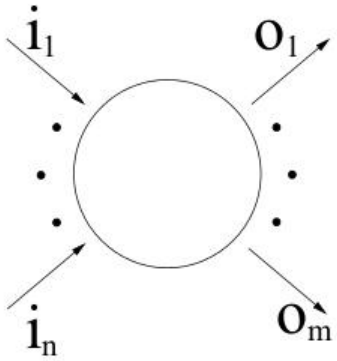


Figura 3.1: Vista externa de componente.

La figura 3.1 muestra la descripción externa de un componente, donde el componente en sí mismo es representado como un círculo, sus puertos de entrada como flechas entrantes al componente y sus puertos de salida como flechas salientes.

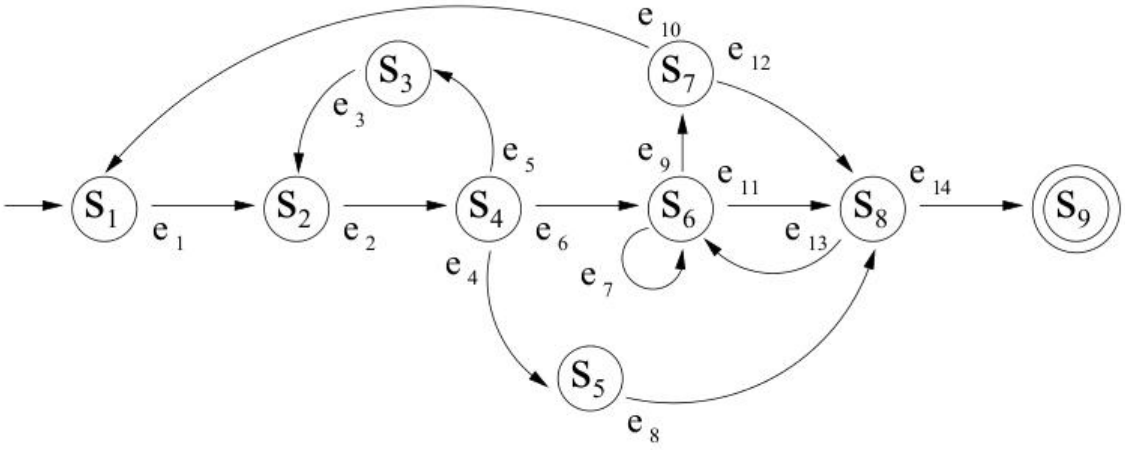


Figura 3.2: Vista interna de componente

En la figura 3.2 observamos una representación interna de un componente, esto es un autómata que modela su comportamiento. En la representación usada del autómata, cada estado es representado por un círculo (el estado final con doble línea), y las transiciones entre estados por flechas. Cada estado se encuentra etiquetado (Sn), así como las flechas de transiciones (ei) indicando bajo qué condición interna al componente, o dato de entrada concreto, o ambos, se produce una transición entre estados.

3.4. Variables observables y controlables.

Con el objetivo de proporcionar robustez y controlabilidad, CoolBOT proporciona dos conjuntos de variables: observables y controlables. Esto permite diseñar componentes que sean observables para así determinar y seguir su correcto funcionamiento, así como otorgarles cierto nivel de control sobre su modo de operación.

- Variables Observables: representan aspectos del componente de interés desde fuera del mismo.

- Variables Controlables: representan aspectos del componente que pueden ser controlados externamente.

CoolBOT garantiza la observabilidad y controlabilidad de cualquier componente, para lo cual introduce dos tipos de puertos por defecto en todo componente: un puerto de monitorización y un puerto de control. La visión externa que tenemos de un componente en la figura 3.1 ahora se ve ampliada a la mostrada en la figura 3.3.

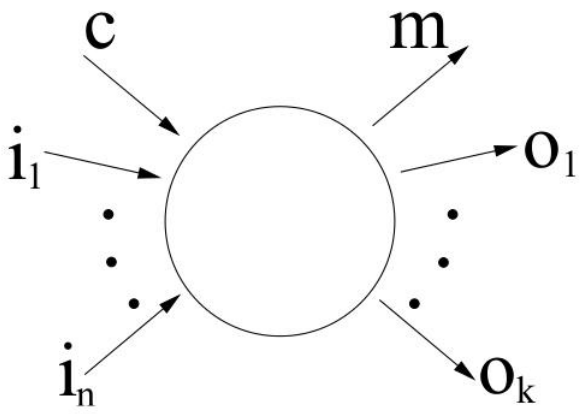


Figura 3.3: Vista externa de componente con puerto de control y monitorización.

- El puerto de monitorización es un puerto público que permite publicar las variables observables.

- El puerto de control es un puerto público que permite modificar/actualizar las variables de control.

Así pues cualquier componente puede ser controlado y monitorizado por un supervisor externo, tal y como se ilustra en la figura 3.4, donde se observa la utilidad de ambos puertos.

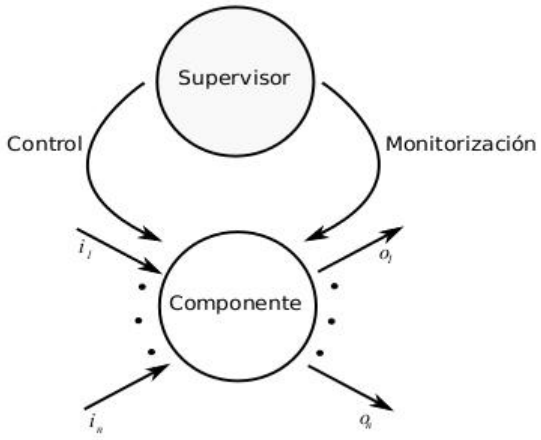


Figura 3.4: Bucle común de control.

Además CoolBOT proporciona a cada componente de una serie de variables observables y controlables por defecto. Estas variables se describen en la tablas 3.1 y 3.2.

Variables de monitorización por defecto	
Nombre	**Descripción**
state (s)	Estado del autómata donde se encuentra el componente.
priority (p)	Prioridad actual de ejecución del componente.
config (c)	Solicita un cambio de configuración supervisado, confirma comandos de configuración.
result (r)	Resultado de ejecución.
error description (ed)	Descripción de error indicando una excepción local irrecuperable.

Cuadro 3.1: Variables de monitorización por defecto.

Variables de control por defecto	
Nombre	**Descripción**
new state (ns)	Estado del autómata al que se desea que el componente transite.
new priority (np)	Prioridad de ejecución a la que se desea que el componente se ejecute.
new exception (nex)	Excepción inducida externamente.
new config (nc)	La configuración del componente puede ser modificada y actualizada durante la ejecución a través de esta variable de control.

Cuadro 3.2: Variables de control por defecto.

3.5. Autómata por defecto.

Como ya se mencionó con anterioridad, en CoolBOT todos los componentes se modelan a partir de un autómata por defecto. Podemos ver una representación del autómata por defecto en la figura 3.5, donde los diferentes estados se representan por un círculo (estado final con doble línea) y las transiciones entre los mismos por arcos etiquetados para indicar el evento que dispara la transición.

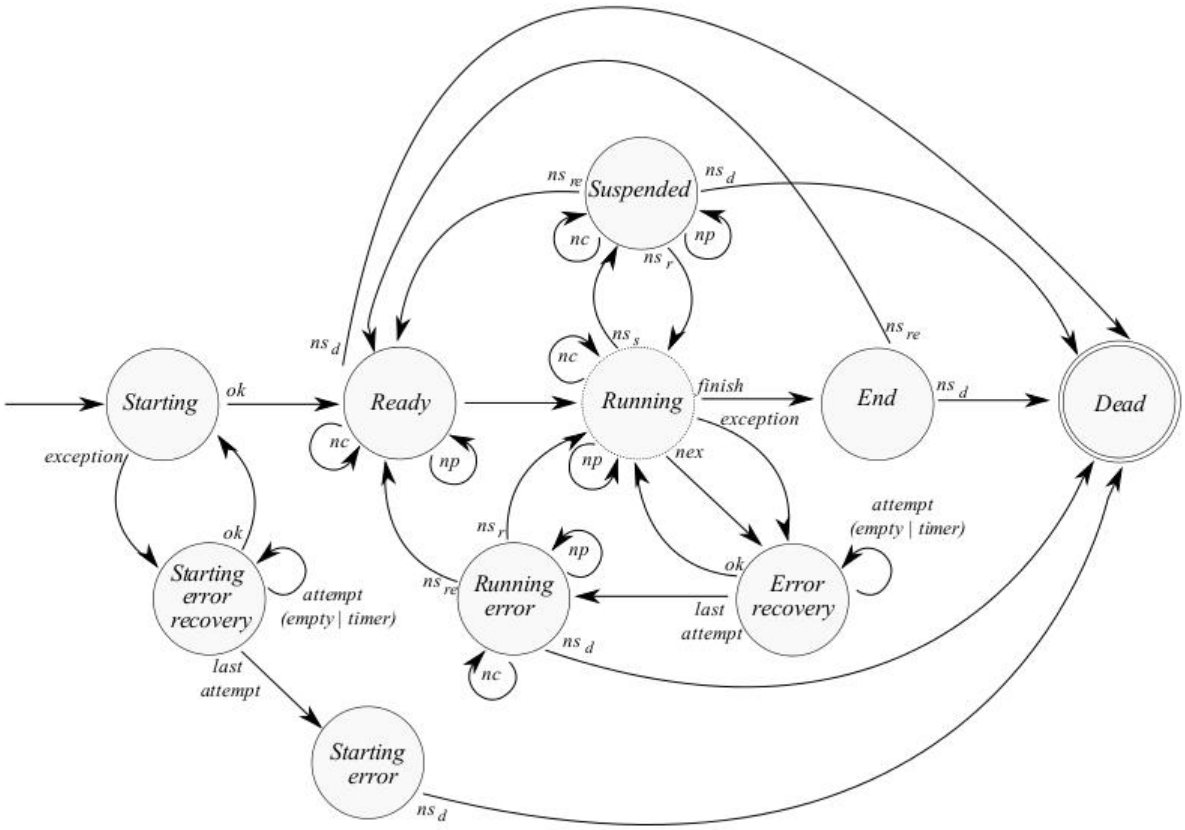


Figura 3.5: Autómata por defecto.

Algunos de los eventos que provocan transiciones son internos al componente (*exception, nex, attempt, ok, last attempt, finish*), el resto son eventos provocados por cambios en las variables de control por defecto (*nsr , nsre , nss , nsd , np, nc y nex*), donde el subíndice indica a que estado del autómata ha sido comandado el componente: r (*running state*), re (*ready state*), s (*suspended state*), d (*dead state*). Los restantes eventos (*np, nc y nex*) indican que un supervisor externo ha cambiado la prioridad a la que el componente ha de ejecutarse (np), su configuración interna (nc), o bien, que ha inyectado una excepción (nex) para su comprobación.

Como se puede observar en la figura 3.5, el estado running se encuentra indicado mediante un círculo con línea discontinua. En realidad el estado running es un pseudoestado que representa

la porción del autómata donde se implementa la funcionalidad concreta del componente, por ello es llamado autómata de usuario. Obviamente el autómata de usuario varía entre componentes dependiendo de la funcionalidad que se requiera en cada caso, este autómata es definido durante la fase de creación del componente.

El resto de los estados del autómata por defecto organizan la vida de un componente en distintas fases:

- **starting**: Adquiere los recursos necesarios para la ejecución del componente.
- **ready**: el componente esta listo para la ejecución y se encuentra a la espera de que se le o comande transitar hacía el autómata de usuario (nsr).
- **running**: se ejecuta del autómata de usuario.
- **suspended**: el componente se encuentra suspendido a la espera de que se le comande transistar a otro estado.
- **end**: el componente ha acabado su tarea y finaliza su ejecución publicando el resultado (si lo hubiera) a través del puerto de monitorización.
- **dead**: finalización del componente.

Además existen dos estados concebidos para el tratamiento de fallos durante la ejecución del componente. Starting *error recovery* y *starting error* manejan errores durante la adquisisión de recursos, mientras que *error recovery* y *running error*, manejan los errores durante la ejecución.

3.6. Componentes multihilo.

Los componentes CoolBOT son entidades independientes que se ejecutan concurrentemente para realizar y llevar a cabo sus propios objetivos y tareas. Cada componente se mapea en hilos, ya sean Win32 o POSIX, seg´n nos encontremos sobre Windows o GNU/Linux respectivamente.

Durante la ejecución de un componente CoolBOT este se encuentra en un bucle constante procesando paquetes de puertos que acarrean distintas acciones dependiendo del tipo o contenido del paquete que se reciba y del estado actual del autómata que modela al componente. En general, son las llegadas de paquetes las que ocasionan las transiciones entre estados del autómata y en función de la frecuencia de llegada de los mismos, puede darse el caso de un bloqueo del componente a la espera de paquetes de puertos, esto es, los componentes se comportan como máquinas de flujo de datos, que procesan la información cuando disponen de ella en sus entradas y en otro caso esperan la llegada de datos a través de sus puertos de entrada.

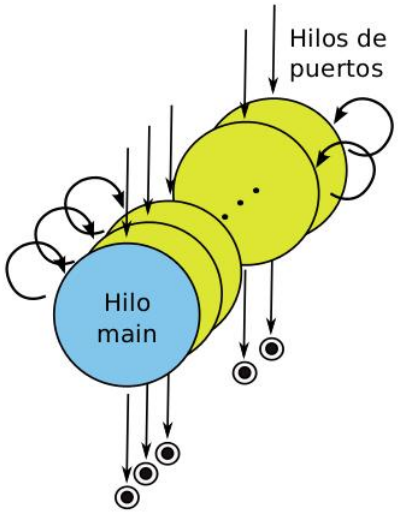


Figura 3.6: Componente multihilo.

Este bucle de procesamiento que constituye el núcleo de un componente puede descomponerse, o no, en unidades más ligeras de ejecución: hilos. De forma general, todo componente necesita para su ejecución de al menos un hilo, éste es el llamado *hilo main*. Sin embargo, con el objetivo de lograr que un componente sea más reactivo, es posible distribuir la atención de los puertos de entrada en múltiples hilos, como se ilustra en la figura 3.6. Estos son los llamados *hilos de puertos*. Estos hilos atienden conjuntos disjuntos de puertos de entrada del componente, y siguen el mismo paradigma de máquina de flujo de datos para el procesamiento de paquetes que lleguen a través de dichos conjuntos de puertos. En los casos de componentes donde aparezcan este tipo de hilos, es el *hilo main* el encargado de ejecutar el autómata del componente así como de controlar y monitorizar dichos hilos. Por otro lado, el hilo main es también el encargado de mantener la consistencia de las estructuras de datos internas del componente y sincronizar el acceso a dichos datos sin que se produzcan interbloqueos.

3.7. Intercomunicación entre componentes CoolBOT.

De forma similar a la comunicación entre procesos (IPC: Inter Process Communications)[16] aportada por los actuales sistemas operativos, CoolBOT utiliza un sistema de comunicación entre componentes (ICC: Inter Component Comunications). Este modelo estandariza las comunicaciones, permitiendo el trabajo cooperativo entre componentes a la par que manteniéndolos desacoplados, lo que favorece la reutilización y desarrollo independiente de componentes.

El modelo de intercomunicación utilizado en CoolBOT se basa en los puertos de entrada y de salida de los componentes, realizando conexiones entre los mismos. Los datos se transmiten a través de estas conexiones en forma de paquetes de datos de distintos tipos denominados paquetes de puertos (port packets). Como norma general, los puertos de entrada y de salida sólo aceptan un conjunto limitado de esos tipos de paquetes de datos.

3.8. Tipos de puertos y conexiones.

CoolBOT proporciona distintos tipos de puertos de entrada y salida, que determinan distintos protocolos de comunicaciones al establecer conexiones entre ellos. Esto permite que, combinándolos adecuadamente como conexiones entre puertos, se haga uso de diferentes protocolos de interacción entre componentes. Cabe destacar que las conexiones entre puertos sólo son posibles si los tipos de paquetes que aceptan ambos puertos coínciden, pero además ambos puertos deben constituir a un par compatible. La tabla 3.3 muestra las conexiones posibles entre puertos de entrada y salida, así como una descripción del protocolo que se obtiene para cada conexionado.

Puerto de Salida	Puerto de Entrada	Descripción
OTick (t)	ITick (t)	Conexiones tipo Tick : Implementa un protocolo para señalizar eventos entre componentes
OGeneric (g)	ILast (l)	Conexiones tipo Last, Fifo y Unbounded Fifo: Hay una cola (fifo) de paquetes en el puerto de entrada .
	IFifo (f)	
	IUFifo (uf)	
OPoster (p)	IPoster (p)	Conexiones tipo Poster : Hay una copia principal de paquetes en el puerto de salida, los puertos de entrada mantienen copias locales
OShared (s)	IShared (s)	Conexiones tipo Shared: Los componentes comparten una memoria residente en el puerto de salida. Implementa un protocolo de memoria compartida
OMultiPacket (mp)	IMultiPacket (mp)	Conexiones tipo Multi Packet: Acepta múltiples tipos de paquetes a través de la misma conexión entre puertos
OLazyMultiPacket (lmp)		
OPrioriy (pr)	IPriorities (pr)	Conexiones tipo Priority : Implementa un protocolo de envío con prioridad

Cuadro 3.3: Conexiones de Puertos.

Para manejar distintos tipos de paquetes los puertos MultiPacket se subdividen en Slots, cada uno de los cuales está dedicado a un tipo concreto de paquete de puerto de los que el puerto MultiPacket acepte (figura 3.7).

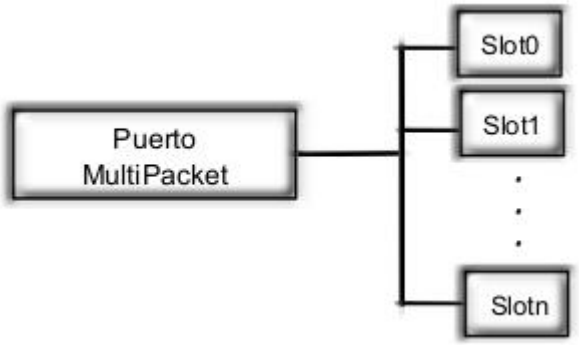


Figura 3.7: Puerto MultiPacket.

Un puerto MultiPacket permite por tanto que cada Slot se conecte a puertos SinglePacket (todos los tipos restantes de puertos) de acuerdo a los criterios de compatibilidad ilustrados en la figura 3.8, estableciendo conexiones MultiPacket simples.

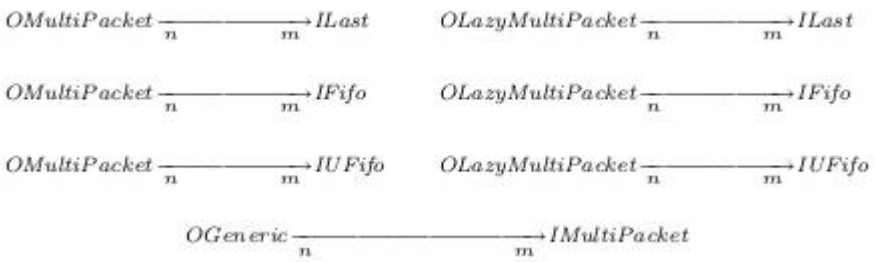


Figura 3.8: Conexiones MultiPacket simples ($\forall n, m \epsilon N; n, m \geq 1$).

3.9. Componentes compuestos.

Los componentes CoolBOT pueden ser descompuestos en dos tipos, componentes atómicos y componentes compuestos. Los primeros son componentes simples ideados para abstraer el hardware subyacente, implementar algoritmos genéricos o encapsular librerías o bibliotecas. Sin embargo, existe la posibilidad de crear componentes más complejos a partir de los atómicos. Los componentes compuestos tienen como atributos instancias de componentes simples y/o otros componentes compuestos, de forma que se establece una jerarquía aprovechando la modularidad que ofrece el concepto de componente usado por CoolBOT.

La idea principal es que los componentes compuestos implementan su funcionalidad apoyándose en las de componentes más simples. Cada componente compuesto supersiva, usando los puertos a de control y monitorización, las acciones de los componentes que integra. Esta jerarquía de componentes se ilustra en la figura 3.9.

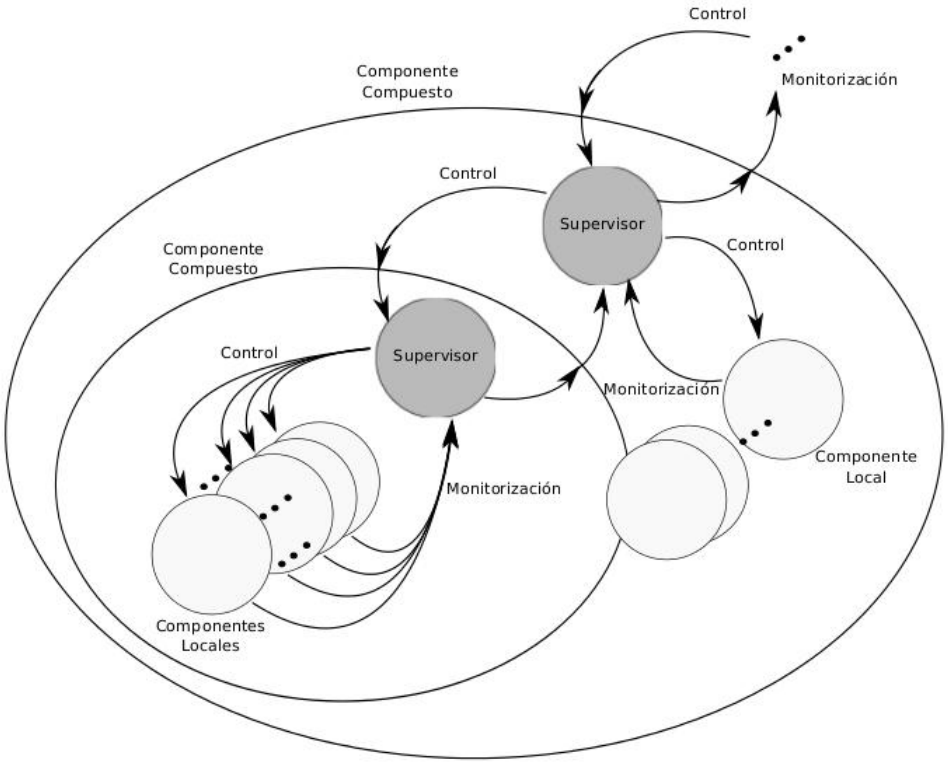


Figura 3.9: Jerarquía de componentes.

3.10. Como crear proyectos en CoolBOT

Lo primero a valorar cuando queremos comenzar a realizar la creación de alguna implementación en CoolBOT es conocer en qué compenentes queremos organizarla. Una vez hemos decidido esto necesitamos conocer que vistas necesitaremos y que información necesitamos mostrar por pantalla. Consecuentemente, al decidir que datos usaran las vistas de los componentes podemos proceder a generar los paquetes que transladarán los datos por las diferentes interfaces.

3.10.1. Componentes.

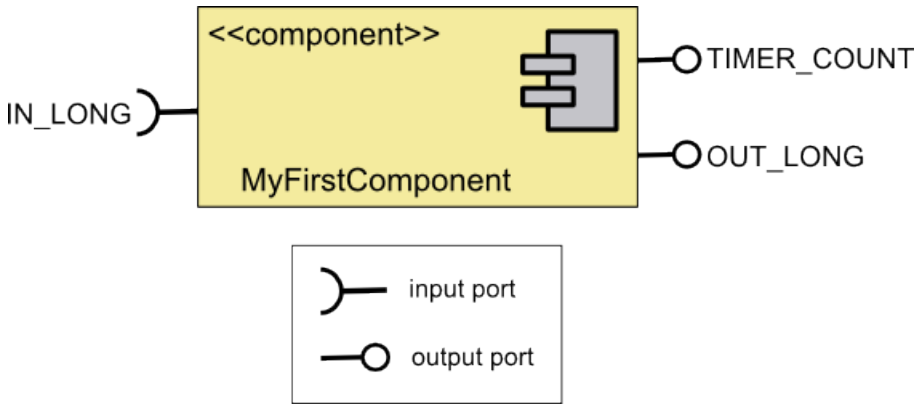


Figura 3.10: Componente en CoolBOT

Crear un componente tiene como objeto resolver un problema mediante una serie de procesos generalmente de una cierta complejidad computacional. Este componente se caracteriza por tener una serie de entradas y salidas donde se comunica con el exterior mediante paquetes ya sea implementados por el usuario o aquellos predefinidos de los que dispone CoolBOT. Esto permite aislar los procesos internos fácilmente, el comando que se necesita para crear un componente es:

```
coolbot-ske --create-component name-component component-dir
```

Mediante este simple comando el sistema CoolBOT crea un esqueleto básico de ficheros y directorios que sirven de soporte para desarrollar nuestro componente.

Primeramente, hay que tener en cuenta que el primer lugar donde debemos dirigirmos una vez realizado el comando es al directorio "src" donde se ubica el fichero en el que desarrollaremos como queremos que sea el esqueleto de nuestro componente. En él definiremos las entradas, las salidas, los estados del automata y con que entrada/salida transitaremos por los estados. Fijados estos parámetros prodecemos a generar el esqueleto en C++ mediante el comando:

```
coolbot-c  [name-component].coolbot-component
```

Una vez generados los ficheros C++ de cabecera ".h" y de código ".cpp" procedemos a incluir los ficheros que usemos y desarrollar el código normalmente, hasta ver el diseño deseado implementado y funcionando.

3.10.2. Vistas.

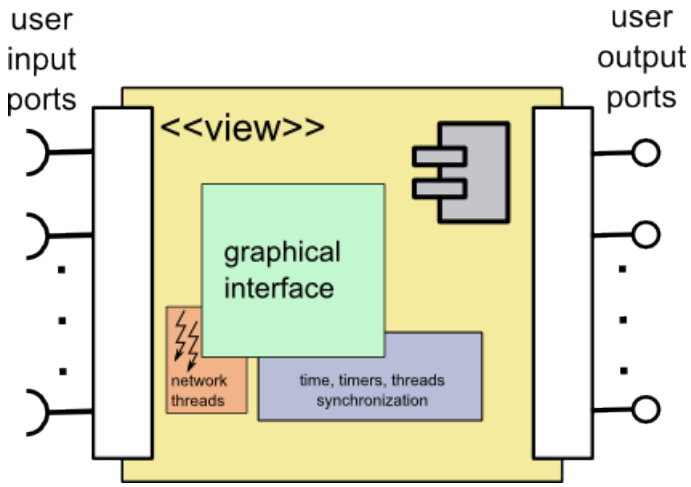


Figura 3.11: Vista en CoolBOT

A la hora de crear una vista en CoolBOT la dinámica se asemeja a un componente, mediante el siguiente comando:

```
coolbot-ske --create-view name-view view-dir
```

Mediante es simple comando CoolBOT crea el esqueleto de la vista, incluyendo para ello las librerías Gtk [23], el objetivo de este comando es crear un directorio donde se encuentra una estructura de ficheros para el desarrollo de la vista. Para definir que constantes, Entradas/Salidas, etc. tenemos que dirigirnos al subdirectorio "src/" que contiene un único fichero, cuyo nombre es "[name-view].coolbot-view", encargado de definir a alto nivel cuál será la composición de la vista.

Una vez definido el fichero "[name-view].coolbot-view" prodecemos a generar el código C++ mediante el siguiente comando:

```
coolbot-c  [name-view].coolbot-view
```

Este comando generará dos ficheros C++, "[name-view].h" y "[name-view].cpp", que ubicará la clase traduciendo las declaraciones ubicadas en el fichero "name-view.coolbot-view" a código que pueda entender el compilador de C++.

3.10.3. Integraciones.

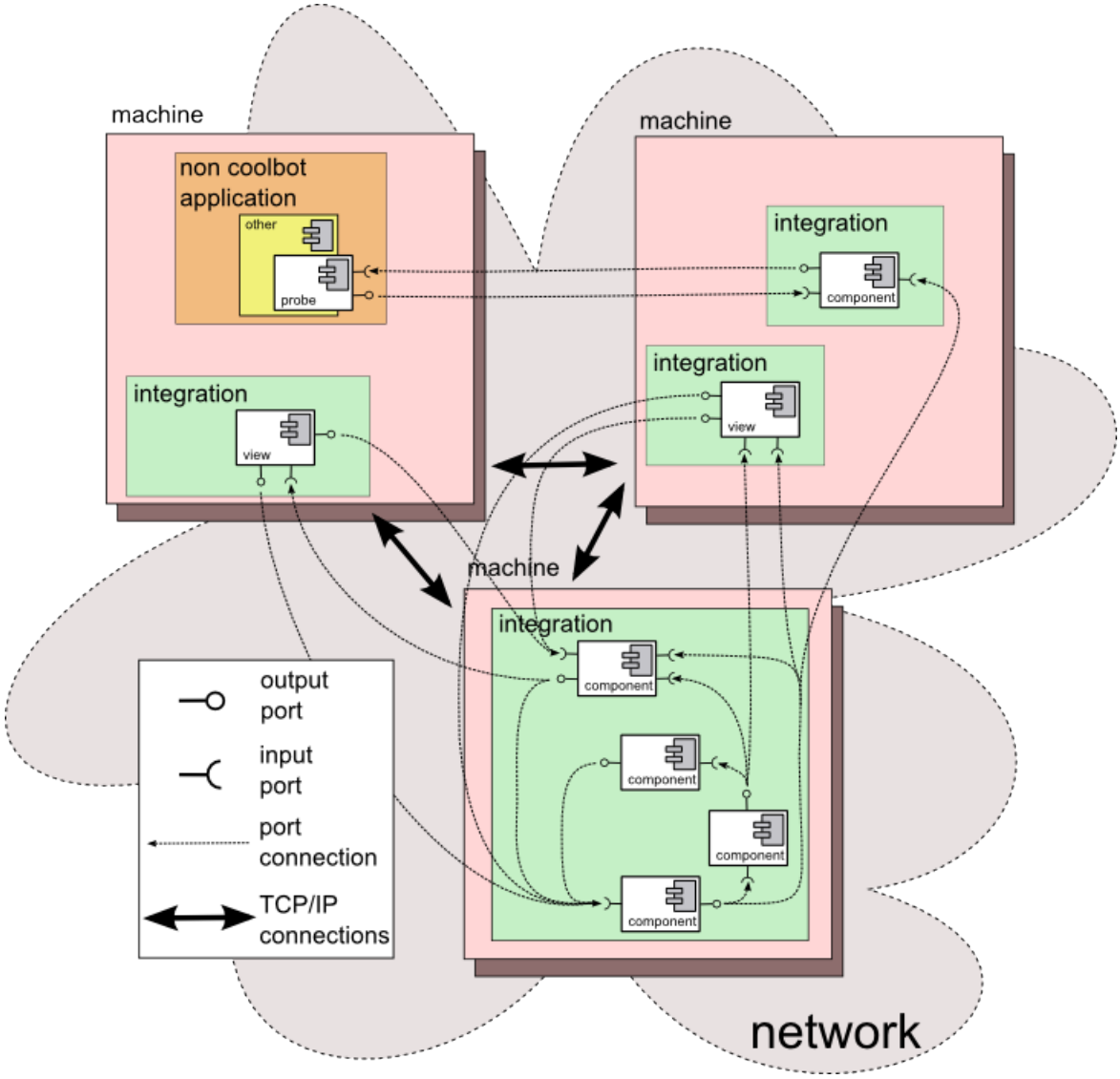


Figura 3.12: Integración en CoolBOT

A la hora de llevar a cabo una integración la mecánica es similar mediante el mismo comando pero con una opción diferente llevamos a cabo un nuevo esqueleto

```
coolbot-ske --create-integration name-integration integration-dir
```

Una vez llevado a cabo la integración la forma en la que rellenamos el fichero que se encuentra en "src" es algo distinto. En primer lugar definimos la maquina donde se llevaran a cabo el acceso

(Ejemplo, vistas en ordenador y componentes en el robot), seguidamente definimos el nombre que tendrá cada componente y vista para identificarlos en el siguiente paso que es definir las conexiones que relacionarán los componentes y vistas con otros componentes y vistas, de forma que podemos entablar relaciones múltiples que esten relacionadas por los puertos a los que se conecten siendo esto una sencilla y eficiente forma de hacerlo.

Finalizado el proceso de interconexión solo queda llamar al comando:

```
coolbot-c [name-integration].coolbot-integration
```

Para con ellos tener generado el código C++ (su fichero "[name-integration].cpp") que ejecutará nuestra aplicación. No obstante, para llevar a cabo su correcta ejecución es necesario tener presente que cada componente y vista tienen en su llamada al constructor la posibilidad de declarar parámetros y por tanto, en algunos casos será necesario añadirles los valores necesarios para que se ejecuten adecuadamente.

3.10.4. Paquetes de puertos.

Crear paquete de datos tiene como objeto abstraer al programador de las capas de conexionado y comunicación existentes para la transmisión adecuada de los datos entre componentes y vistas adecuadamente. Esto permite al usuario se limite a definir que información quiere empaquetar y elimine la interacción directa con aspectos tales como Sockets, Protocolos (TCP/IP, UDP, etc.), Serialización y Deserialización de datos, etc. A la par que se protege un poco la integridad de las comunicaciones entre los distintos módulos en términos de flujo de datos. El comando que se necesita para crear un conjunto paquetes es:

```
coolbot-ske --create-packets name-packets packets-dir
```

Mediante este simple comando el sistema CoolBOT crea un esqueleto básico de ficheros y directorios que sirven de soporte para desarrollar nuestros paquetes.

Primeramente, hay que tener en cuenta que el primer lugar donde debemos dirigirmos una vez realizado el comando es al directorio "src" donde se ubica el fichero en que describiremos como queremos que sea el esqueleto de nuestros paquetes. En el definiremos los distintos paquetes según necesitemos (cabe aclarar que crear un conjunto de paquetes tal cual se define en CoolBOT permite agrupar diversos paquetes bajo un mismo espacio de nombres) cada uno contendrá los tipos de datos que necesitaremos emplear para llevar a cabo la ejecución adecuadamente. Fijados estos parámetros prodecemos a generar el esqueleto en C++ mediante el comando:

```
coolbot-c  [name-component].coolbot.packet
```

Una vez generado los ficheros cabecera "[name-component].h" y código "[name-component].cpp" usando *coolbot-c* procedemos a incluir los ficheros que usemos y desarrollar el código normalmente, hasta ver el diseño deseado implementado y funcionando, Principalmente tres zonas son esenciales para un correcto funcionamiento las funciones:

- **"debug"** encargada de mostrar la información del contenido de las variables del paquete.
- **"assign"** y **"constructor de copia"** se encargarán de pasar adecuadamente los datos y sus respectivas asignaciones.

- Los **"set's" y "get's"** . En la guia de estilo de CoolBOT[7 en la página 159] se sugiere que cada variable tenga su propia pareja de métodos "set" y "get" que se encargan de gestionar la información almacenada en la variable *.coolbot-enviroment*.

3.10.5. CMakeList y .coolbot-envoiroment.

CoolBOT emplea la aplicación pkg-config (más información web: 41 en la página 161) para la busqueda e inclusión de los diferentes módulos que han generado una librería y un fichero pkg-config para cada módulo (En el caso de ser una vista, un componente o un conjunto de paquetes). Por tanto, en el "CMakeList" solo es necesario definir qué módulos externos queremos incluir al nuestro ya sean componentes, vistas o paquetes cuando los integramos. Para llevar a cabo correctamente este proceso el pkg-config junto con el cmake posee un conjunto de herramientas que solo necesitas replicar con el nombre correspondiente como podemos ver a continuación:

```
#Código para la busqueda de la librería y sus flags de compilación
GET_FLAGS_PKGCONFIG("NOMBRE")
SET(NOMBRE_CFLAGS "${PKG_CFLAGS}")
GET_LIBS_PKGCONFIG("NOMBRE")
SET(NOMBRE_LIBS "${PKG_LIBS}")
#Código de inclusón de ruta en los ficheros bin de NOMBRE a compilar
SET(FLAGS_BIN_COMPILER "${FLAGS_BIN_COMPILER} ${NOMBRE_CFLAGS}")
SET(FLAGS_BIN_LINKER   "${FLAGS_BIN_LINKER} ${NOMBRE_LIBS}")
#Código de inclusión de las librerías de NOMBRE
SET(FLAGS_LIB_COMPILER "${FLAGS_LIB_COMPILER} ${NOMBRE_CFLAGS}")
SET(FLAGS_LIB_LINKER   "${FLAGS_LIB_LINKER} ${NOMBRE_LIBS}")
```

Por otro lado, la herramienta "pkg-config" necesita encontrar la ruta de cada NOMBRE en el path del sistema. Para ello, CoolBOT posee de un fichero configurable llamado ".coolbotenvoiroment" creado al instalarlo, preparado para definir las rutas "lib" y "pkg-config" de cada componente, vista y conjunto de paquetes, mediante las siguientes variables de entorno:

- Path de liberias indicado por COOLBOT_LIB_PATH
- Ficheros pkg-config indicado por COOLBOT_PKG_PATH

Para actualizar el los recorridos del entorno solo tenemos que reiniciar el shell mediante el comando:

```
source ~/.bashrc
```

Capítulo 4

Metodología, recursos y plan de trabajo.

4.1. Elección del plan de trabajo.

Hemos comentado que el objetivo de este proyecto fin de carrera es implementar y tener funcionando el algoritmo MbICP dentro de la plataforma CoolBOT, que esta implementada sobre lenguaje C++, pero hasta el momento no hemos hablado del proceso que nos llevó al plan de trabajo que ha sido ejecutado. Para ello introduciremos el motivo por el que se optó por un prototipo en Matlab.

La propuesta de este proyecto fin de carrera (PFC: Reducción de Errores de Odometría en un Robot Móvil utilizando Algoritmos de Scan Matching basados en Sensores de Rango) ha sido selecionada por su proximidad al ámbito de la robótica, cuyo nivel de complejidad es adecuado para comenzar la formación en esta disciplina.

Una vez elegido el proyecto, se llevó a cabo un primer estudio del análisis y la formulación del algoritmo MbICP a través de la bibliografía [1]. Observando tras su lectura que posee una elevada curva de aprendizaje y que su implementación en CoolBOT directa sería costosa y propensa a errores.

Por todo esto, la realización de un prototipo inicial en Matlab, (aplicación conocida a lo largo de la carrera y su lenguaje Q) permite ir implementado pequeños prototipos incrementales del algoritmo MbICP eliminando así las incertifdumbres y visualizar incrementalmente su comportamiento.

En caso de haber optado por implementarlo directamente sobre la plataforma CoolBOT, existiría una elevada curva de aprendizaje ocasionada por el algoritmo MbICP unido a la curva de aprendizaje propia de la plataforma CoolBOT.

4.2. Metodología.

El desarrollo de este proyecto fin de carrera se basa en dos aplicaciones, las primera para el estudio y análisis del algoritmo MbICP para el desarrollo de un prototipo en Matlab y la segunda con el claro objetivo de incorporar el algoritmo MbICP al robot real mediante la plataforma CoolBOT. Esto plantea dos Modelos de ciclo de vida del software bien distintos uno para cada aplicación.

4.2.1. Prototipo en Matlab.

Comenzaremos por describir el paradigma usado para el prototipo en Matlab. Este software en desarrollo necesita tener presente volver a definir fases anteriores debido a que el conocimiento inicial del problema esta limitado al artículo de referencia [1] y no se posee un claro análisis de los requisitos para su diseño e implementación. En consecuencia, previendo posibles actualizaciones y mejoras de las distintas fases de desarrollo del software se ha optado por el ***ciclo de vida iterativo e incremental*** [20] [21]. El desarrollo se organiza en una serie de mini-proyectos cortos, de duración fija (alrededor de cuatro semanas) llamados iteraciones; el resultado de cada uno es un sistema que puede ser probado, integrado y ejecutado. Cada iteración incluye sus propias actividades de análisis de requisitos, diseño, implementación y pruebas. El ciclo de vida iterativo se basa en la ampliación y refinamiento sucesivos del sistema mediante múltiples iteraciones, con retroalimentación cíclica y adaptación como elementos principales que se dirigen para converger hacia un sistema adecuado. El sistema crece incrementalmente a lo largo del tiempo, iteración tras iteración, y por ello, este enfoque también se conoce como desarrollo iterativo e incremental.

Note que el desarrollo incremental es 100 % compatible con el ***modelo en cascada*** [20]. Así, el modelo en cascada puede ser usado para administrar cada esfuerzo de desarrollo, como se muestra en la figura.

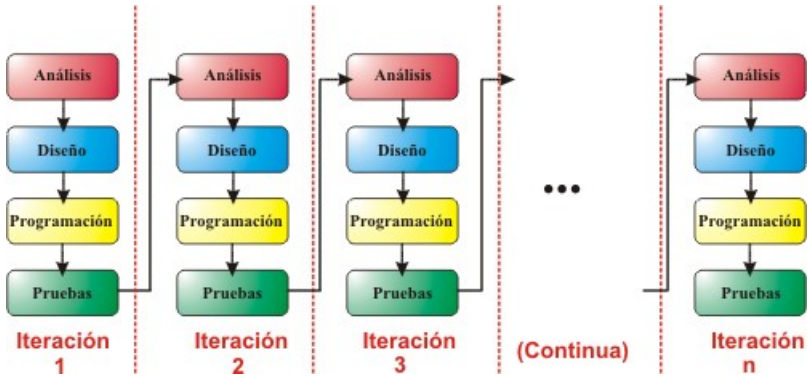


Figura 4.1: Ciclo de vida iterativo e incremental.

El resultado de cada iteración es un sistema ejecutable, pero incompleto; no está preparado para ser puesto en producción. El sistema podrá no estar listo para su puesta en producción hasta después de varias iteraciones (por ejemplo 10 ó 15).

La salida de una iteración no es un prototipo experimental o desechable. La salida es un subconjunto de la funcionalidad total con calidad de producción del sistema final.

Aunque, en general, cada iteración aborda nuevos requisitos y amplía el sistema incrementalmente, una iteración podría, ocasionalmente, volver sobre el software que ya existe y mejorarlo; por ejemplo, una iteración podría centrarse en mejorar el rendimiento de un subsistema, en lugar de extenderlo con nuevas características.

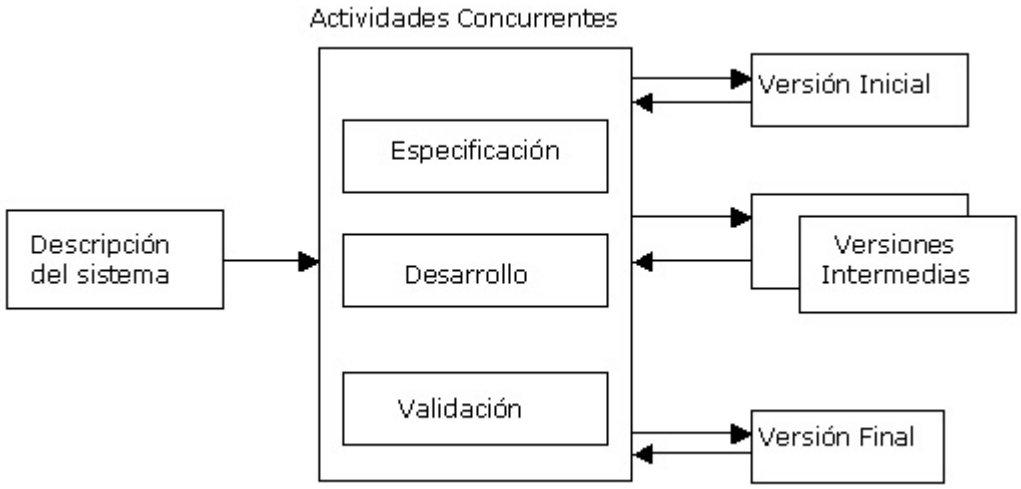


Figura 4.2: Relación de tareas en ejecución durante cada iteración del ciclo vida iterativo e incremental

El modelo de desarrollo incremental provee algunos beneficios significativos para los proyectos:

- Construir un sistema pequeño, siempre contiene menos riesgos que construir un sistema grande. Cada iteración mitiga los posibles riesgos tanto como sea posible (técnicos, requisitos, objetivos, usabilidad y demás).
- Al ir desarrollando parte de las funcionalidades, es más fácil determinar si los requerimientos planeados para los niveles subsiguientes son correctos.
- Progreso visible en las primeras etapas.
- Una temprana retroalimentación, compromiso de los usuarios y adaptación, que nos lleva a un sistema refinado que se ajusta más a las necesidades reales del personal involucrado.
- Reduciendo el tiempo de desarrollo de un sistema (en este caso en incremento del sistema) decrecen las probabilidades que esos requerimientos de usuarios puedan cambiar durante el desarrollo.
- Gestión de la complejidad; el equipo de desarrollo no se ve abrumado por la "parálisis por análisis o pasos muy largos y complejos".
- Si se produce un error importante, puede ser usado un incremento previo y solo se necesita ser descartada la última iteración.
- Los errores de desarrollo realizados en un incremento, pueden ser arreglados antes del comienzo del próximo incremento.

- El conocimiento adquirido en una iteración se puede utilizar metódicamente para mejorar el propio proceso de desarrollo, iteración a iteración.

El modelo incremental se ha elegido como la metodología a seguir durante el desarrollo del prototipo en Matlab porque al comienzo del proyecto eran desconocidos todos los requisitos del sistema, ajustándose mejor frente a los otros ciclos de vida a este requisito. Se planteá un análisis inicial del algoritmo MbICP para dar lugar a un diseño e implementación inicial coherente. Seguidamente, se van refinando succesivamente las iteraciones hasta obtener un prototipo que reuna todos los requisitos necesarios para validar el correcto funcionamiento del algoritmo MbICP y su respectiva implementación. Evaluándolo constantemente mediante una serie de conjuntos de prueba que nos dé estimaciones precisas de los siguientes cambios que hay que introducir en la nueva iteración.

4.2.2. Diseño y desarrollo del algoritmo MbICP en CoolBOT.

El paradigma que requería el desarrollo del algoritmo MbICP en CoolBOT es bien diferente porque partimos del conocimiento exhaustivo de los requisitos del sistema, diseño e implementación (aunque en otro lenguaje de progamación). En base a estos requisitos, el paradigma iterativo e incremental consumiría un tiempo superior de desarrollo haciéndolo más costoso y difícil de planificar.

El ***modelo en cascada*** es el más básico de todos los modelos, y sirve como bloque de construcción para los demás modelos de ciclo de vida. La visión del modelo cascada [20] [21] [22] del desarrollo de software es muy simple; dice que el desarrollo de software puede ser a través de una secuencia simple de fases. Cada fase tiene un conjunto de metas bien definidas, y las actividades dentro de una fase contribuye a la satisfacción de metas de esa fase o quizás a una subsecuencia de metas de la fase. Las flechas de la figura 4.3 muestran el flujo de información entre las fases. La flecha de avance muestra el flujo normal. Las flechas hacia atrás representan la retroalimentación.

Este ciclo de vida es el más conocido, esta basado en el ciclo convencional de una ingeniería, el paradigma del ciclo de vida abarca las siguientes actividades:

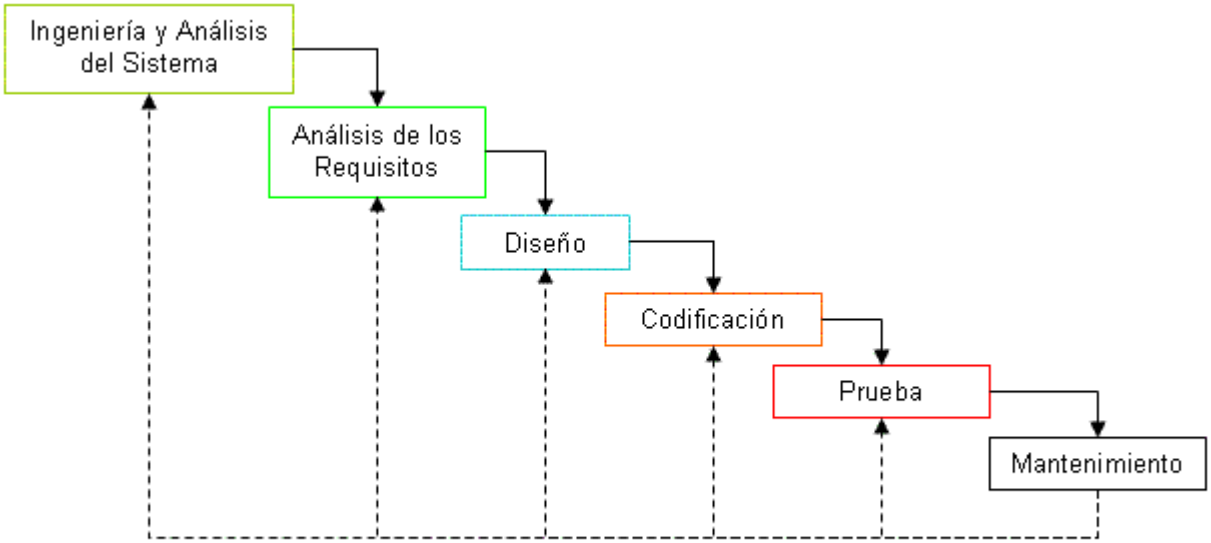


Figura 4.3: Ciclo de vida en cascada

- **Ingeniería y Análisis del Sistema:** Debido a que el software es siempre parte de un sistema mayor el trabajo comienza estableciendo los requisitos de todos los elementos del sistema y luego asignando algún subconjunto de estos requisitos al software.
- **Análisis de los requisitos del software:** el proceso de recopilación de los requisitos se centra e intensifica especialmente en el software. El ingeniero de software (Analistas) debe comprender el ámbito de la información del software, así como la función, el rendimiento y las interfaces requeridas.
- **Diseño:** el diseño del software se enfoca en cuatro atributos distintos del programa: la estructura de los datos, la arquitectura del software, el detalle procedimental y la caracterización de la interfaz. El proceso de diseño traduce los requisitos en una representación del software con la calidad requerida antes de que comience la codificación.
- **Codificación:** el diseño debe traducirse en una forma legible para la maquina. El paso de codificación realiza esta tarea. Si el diseño se realiza de una manera detallada la codificación puede realizarse mecánicamente.
- **Prueba:** una vez que se ha generado el código comienza la prueba del programa. La prueba se centra en la lógica interna del software, y en las funciones externas, realizando pruebas que aseguren que la entrada definida produce los resultados que realmente se requieren.

- **Mantenimiento:** el software sufrirá cambios después de que se entrega al cliente. Los cambios ocurrirán debido a que hayan encontrado errores, a que el software deba adaptarse a cambios del entorno externo (sistema operativo o dispositivos periféricos), o debido a que el cliente requiera ampliaciones funcionales o del rendimiento.

Principios a tener en cuenta en un desarrollo en cascada:

- Planear un proyecto antes de embarcarse en él.
- Definir el comportamiento externo deseado del sistema antes de diseñar su arquitectura interna.
- Documentar los resultados de cada actividad.
- Diseñar un sistema antes de codificarlo.
- Testear un sistema después de construirlo.

Cuyas ventajas son inmediatamente deducibles de los principios comentados:

- La planificación es sencilla.
- La calidad del producto resultante es alta.
- Permite trabajar con personal poco cualificado.

No obstante, este método posee una serie de desventajas para desarrollo como el prototipo en Matlab, precisamente por estas desventajas se lo descartó y optó por el ciclo de vida incremental:

- Lo peor es la necesidad de tener todos los requisitos al principio. Lo normal es que el cliente no tenga perfectamente definidas las especificaciones del sistema, o puede ser que surjan necesidades imprevistas.
- Si se han cometido errores en una fase es difícil volver atrás.
- No se tiene el producto hasta el final, esto quiere decir que:
  - Si se comete un error en la fase de análisis no lo descubrimos hasta la entrega, con el consiguiente gasto inútil de recursos.
  - El cliente no verá resultados hasta el final, con lo que puede impacientarse.
- No se tienen indicadores fiables del progreso del trabajo (síndrome del 90 %).[1]
- Es comparativamente más lento que los demás y el coste es mayor también.

Por tanto, se ha optado por el ciclo de vida en Cascada orientado especialmente para desarrollo de aplicaciones en tiempo real,concretamente en este caso aporta un sistema sólido, sencillo y eficaz para implementar nuestro componente en CoolBOT y su respectiva vista, como veremos.

[1] Consiste en creer que ya se ha completado el 90 % del trabajo, pero en realidad queda mucho más porque el 10 % del código da la mayor parte de los problemas [22]

4.3. Recursos necesarios.

4.3.1. Recursos Hardware.

Durante el desarrollo del proyecto han sido utilizados los siguientes componentes hardware:

- Ordenador portátil LG.
  - Procesador: Intel® Core™2 Duo Processor T9600.
  - Memoria RAM: 4096MB DDR3 800 Mhz + Intel® Turbo Memory 1.6 (2GB)
  - Disco duro: 320Gb SATA
- Robot Pioner 3.
- Sensor de rango, modelo SILK LMS200.
- Red inalámbrica.
- Disco duro externo Usb: 500Gb para copias de seguridad.
- Servidor git división RAC del SIANI.

4.3.2. Recursos Software.

Durante el desarrollo de este trabajo se han utilizado un gran número de herramientas software tanto para el sistema operativo GNU/Linux como para Windows Vista. Concretamente las distribuciones utilizadas han sido:

- Ubuntu 10.8 notebook edition.
- Microsoft Windows Vista.

A continuación se puede observar una lista con las aplicaciones utilizadas durante o en el desarrollo de este trabajo (las más significativas), clasificadas según el género al que pertenecen.

- Entornos de desarrollo integrado.
  - **Kdevelop 4.1.2 (GNU/Linux) [27].** Entorno de desarrollo integrado en C++ para el desarrollo de la fase final en CoolBOT. Además incluye soporte para CMake.
  - **Matlab 2008ra (GNU/Linux, Vista) [26]:** Software matemático que ofrece un entorno de desarrollo con un lenguaje de programación propio (lenguaje M). Entre sus prestaciones básicas se hallan: la manipulación de matrices, la representación de datos y funciones, la implementación de algoritmos, la creación de interfaces de usuario (GUI) y la comunicación con programas en otros lenguajes y con otros dispositivos hardware. El paquete Matlab dispone de dos herramientas adicionales que expanden sus prestaciones, a saber, Simulink (plataforma de simulación multidominio) y GUIDE (editor de interfaces de usuario - GUI).
- Librerías.

- **Gtk + 2.0 (GNU/Linux) [23]:** Conjunto de bibliotecas multiplataforma para desarrollar interfaces gráficas de usuario (GUI), principalmente para los entornos gráficos GNOME, aunque portable tambien a Windows.
- **Eigen (GNU/Linux) [18]:** Librería matemática para el cálculo vectorial y matricial, portable a todas las plataformas que soporten C++. Hablaremos de ella con más detalle en la sección 7.2

- Servidor de versiones.
  - **cvs (GNU/Linux) [28]:** Servidor de versiones donde reside la versi´n de CoolBOT sin soporte o de red.
  - **Git (GNU/Linux)[29]:** Servidor de versiones donde se encuentra la nueva versión de CoolBOT con soporte de red así como todos los componentes CoolBOT y sus vistas, que vayamos implementando, además de todo el software desarrollado en el proyecto.
- Editores LaTeX.
  - **LyX 1.6 (Vista) [30]:** Se trata de un procesador de textos en el que el usuario no necesita pensar en el formato final de su trabajo, sino sólo en el contenido y su estructura (WYSIWYM)(Lo Que Ve Es Lo Que Quieres Decir, por sus siglas en Inglés), por lo que puede ser utilizado para editar documentos grandes (libros) o con formato riguroso (tesis, artículos para revistas científicas), con facilidad.
- Gráficos Vectoriales.
  - **Inkscape(GNU/Linux, Vista) [31]:** Programa de edición de gráficos vectoriales escalables o a (SVG).
- Editores gráficos.
  - **Gimp 2.0 (GNU/Linux, Vista) [32]:** Programa de edición de imágenes digitales en forma de mapa de bits, tanto dibujos como fotografías. Es un programa libre y gratuito. Forma parte del proyecto GNU y está disponible bajo la Licencia pública general de GNU.
- Compiladores.
  - **gcc (GNU/Linux,) [33]:** Compilador GNU del lenguaje C.
  - **g++ (GNU/Linux) [33]:** Compilador GNU del lenguaje C++.
  - **MiKTeX (Vista) [34]:** es una distribución TeX/LaTeX para Microsoft Windows que fue desarrollada por Christian Schenk. Las características más apreciables de MiKTeX son su habilidad de actualizarse por sí mismo descargando nuevas versiones de componentes y paquetes instalados previamente, y su fácil proceso de instalación.
  - **CMake (GNU/Linux) [35]:** CMake es una herramienta multiplataforma de generación o automatización de código. El nombre es una abreviatura para "cross platform make" (make multiplataforma); más allá del uso de "make" en el nombre, CMake es una suite separada y de más alto nivel que el sistema make común de Unix, siendo similar a las autotools de GNU [35].

- Depuradores.
  - **gdb (GNU/Linux) [36]:** Depurador GNU de código C/C++.
  - **cgdb (GNU/Linux) [37]:** Depurador GNU de código C/C++ con un front-end visualmente más amigable.
  - **ddd (GNU/Linux) [38]:** Data Display Debugger o DDD es una popular interfaz gráfica de usuario para depuradores en línea de comandos como GDB, DBX, JDB, WDB, XDB.
- Diagrama UML
  - **SmartDraw 2011 (Vista) [39]:** Programa para crear diagramas, esquemas de trabajo,dibujos técnicos, etc., con una gran potencia.
- Herramientas de Red.
  - **Wireshark (GNU/Linux) [40]:** Herramienta para la captura y análisis del tráfico por la red, utilizada para la depuración del envio de paquete entre los diferentes módulos de la integración en CoolBOT.
- Utilidades.
  - **Pkg-config (GNU/Linux) [25]:** Software que provee una interfaz unificada para almacenar el flujo de compilación y enlazarlo a las bibliotecas instaladas cuando se está compilando un programa a partir del código fuente. pkg-config fue diseñado originalmente para Linux pero ahora está disponible para BSDs, Microsoft Windows, Mac OS X y Solaris.

4.4. Plan de trabajo.

El proyecto se definió con un claro objetivo en su plan de trabajo: analizar el algoritmo MbICP, prototiparlo en Matlab, estudiar los resultados del prototipo e implementarlo en CoolBOT. Para ello, se planificarán una serie de tareas en secuencia que a continuación detallaremos.

Análisis del algoritmo MbICP

En la primera fase, se extrajo la formulación matemática del artículo [1] extrayendo el planteamiento al papel, deduciendo paso a paso las formulas que en él se postulan como solución al paradigma de "reducción de errores de odometría por medio de sensores de rango".

Para este proceso se usaron los conocimientos adquiridos en diferentes disciplinas a lo largo de la carrera como: Control de procesos por computador, Visión por Computador, Neurocomputación, Métodos matemáticos y Biocibernética que aportaron la base de formación para desenvolvernos en aspectos tales como: mínimos cuadrados, el desarrollo de Taylor, la correspondencia de puntos en el espacio, el análisis del sensor de rango representado matemáticamente, etc.

Como consecuencia, esta formación permitió desarrollar la formulación partiendo del planteamiendo inicial del problema y llegar así las expresiones que vimos en el análisis del algoritmo MbICP (Véase sección 5). No obstante, cabe destacar un amplio esfuerzo en su desarrollo abarcar

múltiples disciplinas en resolver un problema real partiendo de un planteamiento inicial, dedicando una semanas a su formulación y varios meses a su compresión, resistiendose aspectos el significado del término L en la ecuación, el operador $\oplus$ ó el manejo de mínimos cuadrados ajustando el método de forma especifica para optimizar su ejecución.

Prototipo en Matlab

Avanzado ya el análisis se procedió al desarrollo del prototipo en Matlab, cuya elaboración nos dió una perspectiva más clara del sistema de ecuciones, mejorando con ello nuestra comprensión profundizando en aspectos tales como: la calibración de L y la influencia de su valor en las ecuaciones, la correspondencia de puntos o el significado del operador $\oplus$.

En la confección del prototipo en Matlab se pusieron en práctica los conocimientos adquiridos en asignaturas relacionadas con Matlab como son: Control de procesos por computador, Reconocimiento de formas, Neurocomputación, Instrumentación, etc. Esto aportó una ventaja a la hora de implementar la interfaz gráfica cuya manejo tiene una pequeña curva de aprendizaje frente a otros sistemas de desarrollo de interfaces, eligido por su manejo matricial muy próximo a la formulación matemática expresada en el algoritmo MbICP, siendo un paso incremental bastante cómodo que permite evitar errores en su traducción a código. Además, su herramientas de visualización y su sencillez permiten un fácil análisis de los datos expresados en la fase siguiente del desarrollo.

El prototipo se implementó manteniendo presente el objetivo final, servir de plataforma para estudiar el algoritmo MbICP, para lograrlo planificamos haciendo uso los siguientes puntos en el plan de trabajo para implementar el prototipo en Matlab :

1. Implementar el algoritmo descrito mediante pseudocodigo en el análisis.
2. Confeccionar un gestor de persistencia para simular el algoritmo con datos reales.
3. Desarrollar un interfaz gráfico base para la visualización de datos y resultados obtenidos.
4. Añadir herramientas para la navegación de la simulación (Siguente, Atras, Reiniciar, Ir a la iteración n, etc.).
5. Completar la interfaz gráfica con herramientas para el estudio del comportamiento del algoritmo MbICP (Dibujar correspondencias, trayectoria recorrida, recta normal, superficie de distancia respecto a un punto, etc.)

No obstante, hay que tener presente que cada una de estas fase es una iteración independiente, de acuerdo al ciclo de vida incremental elegido, dividiéndose en fase de análisis, diseño, implementación y prueba como se puede ver en la figura 4.1. Por tanto, se evalua intensamente el prototipo durante todo el desarrollo para certificar la robustez y eficacia de la interfaz porque servirá de base para extraer conclusiones sobre el algoritmo MbICP discutiendo su capacidades y limitaciones antes de pasar a la implementación en CoolBOT, última fase del desarrollo.

Estudio del algoritmo MbICP sobre Matlab

Una vez hemos analizado como se expresa e implementado el prototipo en Matlab es hora de pasar a evaluar como se comporta el algoritmo MbICP, a pesar de que hay resultado muy alentadores en el artículo original [1] es necesario verificar por nuestros propios medios si hemos conseguido completar el desarrollo del algoritmo adecuadamente para alcanzar tales resultados.

Para ello se obtuvieron los datos de simulaciones reales obtenidos del Repositorio Radish [19] trasladándolo a Matlab por medio del gestor de persistencia, generando un fichero con datos reales para nuestro prototipo. Estos ficheros sirven de base para evaluar cual es el comportamiento del robot haciendo uso del algoritmo, evaluando como realiza las correspondecias, efectividad y número de iteraciones que emplea en converger a una solución.

En consecuencia, se paso largo tiempo evaluando y reparando errores de las fases anteriores tales como fallos de precisión, dibujo erroneo del mapa, etc. Tambien nos permitió estimar el número promedio de iteraciones, tiempo de ejecución y la soluciones que proporciona.

Implementación en CoolBOT

Una vez recorrido todo este camino de desarrollo y resueltos los errores en el desarrollo del algoritmo MbICP es momento de pasar a la plataforma final donde se ejecutará. Antes de comenzar esta fase es necesario recordar que se trata de un entorno real y que el tiempo es uno de los parámetros que prima a la hora de hacer uso de este componente dentro de la cadena procesos en ejecución para hacer operativo el robot.

Por este motivo, la implementación en CoolBOT se ha dividido en tres tareas según sea el tipo de módulo: componente, paquetes y vista. El componente es el núcleo de nuestra aplicación y cuyos conocimientos para llevarlo a cabo parten de las asignaturas de programación de la carrera, de las cuales las más destacadas son Técnolgía de la programación y Estructuras de datos, empleando para ello el lenguaje C++ visto en muchas de ellas.

Por otro lado, la vista componente claramente orientado a gráficos requirió el soporte de Gtk y C++ para mostrar eficazmente los resultados generados por el componente en su ejecución. Esto fue posible gracias a asignaturas tales como Gráficos por computador y fundamentos gráficos de la informática, cuyo manejo de las librerias de dibujo y eventos del sistema para su presentación fue clave a la hora de reducir el tiempo de desarrollo y facilitar una interfaz más elaborados para el usuario final.

Finalmente, los paquetes y la integración a pesar de herramientas incorporadas por la plataforma CoolBOT y que abstraen al programador mediante prototipado. Requirió conocimientos de asignaturas Redes de Computadores y Análisis y Gestion de Sistemas Distribuidos, entre otros. Puesto que era necesario depurar la información proveniente de los puerto del robot mediante software (“wiredshark”) para verificar la correcta entrega de paquetes en la fase de prueba y evaluación.

Por todo esto, el plan de trabajo recoge prácticamente todos y cada uno de los conocimientos expuestos en gran parte de las discipinas estudiadas a lo largo de la carrera, aunque en cierto caso de forma más profunda que en otros, como por ejemplo la ingeniería del software y las asignaturas de programación.

Capítulo 5

Algoritmo MbICP.

5.1. Diseño.

Como ya se ha visto, el algoritmo MbICP consiste en tres pasos fundamentales, que permiten calcular una solución de como ha sido modificado el sistema de referencia debido al movimiento del robot.

1. Hallar el conjunto de puntos que tienen una distancia mínima entre la captura y aquellos puntos almacenados.
2. Calcular q_{min} haciendo uso del conjunto de puntos del paso previo.
3. Comparar $q_{sol} = q_{min} \oplus q_k$ y comprobar que $q_{min} < umbral$.
   a) Si es mayor recalculamos $q_{k+1} = q_{sol}$ y aplicamos la transformación a todos los puntos y repetimos iteración volviendo al paso 1
   b) Si es menor hemos acabado de iterar.

Como resultado de ejecutar el algoritmo obtenemos el sistema de coordenadas que forma q_{min} cuyo valor es el más próximo a la posición actual del sistema robótico mejorando así la estimación odométrica proporcionada por el propio robot. Cada uno de los pasos los describimos en los puntos sucesivos de este documentos con más detalle.

5.1.1. Puntos correlativos a los puntos de referencia.

En esta funcionalidad se evaluarán los puntos de la nueva captura de datos del sensor con los puntos de una captura previa, hallando aquellos puntos que poseen una distancia mínima entre sí. Esta pareja de puntos resultante confeccionará una lista de puntos que se entregará como salida de esta funcionalidad.

$$C_i = \underset{r_j \epsilon Z_{new}}{argmin} \{ d(p_i, q_k(r_j)) \; and \, d(p_i, q_k(r_j)) < d_{min} \} \tag{5.1}$$

Esta funcionalidad tendrá como entrada:

- Barrido del sensor de rango en Z_{ref}, que llamaremos "P".
- Barrido del sensor de rango en Z_{new}, que llamaremos "R".
- Función de distancia que evaluará la correlación, que llamaremos "distance_measure".

Tendrá como salida:

- Matriz con la pareja de puntos (ecuación 5.1) que entre sí posean una distancia inferior al umbral.

Conocidas las entradas y la salida de la función pasamos a describir cual es su funcionamiento interno:

1. Para cada punto de P.
   a) **Iterar** por el vector R y **buscar la distancia mínima** entre P_i y R_j
   b) **Si** la distancia mínima es inferior que el umbral d_{min}, **entonces** almacenar la pareja de puntos en la matriz de salida
2. Retornar como salida las parejas de puntos que cumpla la restricción, $M_{C_iP_i}$.

5.1.2. Distancia MbICP.

Esta funcionalidad será la encargada de aplicar el concepto de distancia visto en el análisis de acuerdo a la formulación,

$$d_p^{a,p}\left(p_1,p_2\right) = \sqrt{\left(\delta_x^2+\delta_y^2-\frac{\left(\delta_x P_{1y}-\delta_y P_{1x}\right)^2}{P_{1y}^2+P_{1x}^2+-L^2}\right)}$$

su implementación es directa gracias a las herramientas matemáticas de Matlab.

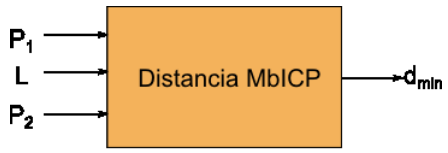


Figura 5.1: Distancia MbICP

Esta función tendrá como entrada:

- Punto origen, llamado P_1

- Punto destino, llamado P_2
- Perímetro de L.

Como salida se tendrá:

- Un valor real positivo que determinará la distancia que existe entre estos dos puntos.

Como se puede observar es innecesario describir el algoritmo, pues supone un mero conjunto de operaciones matemáticas. No obstante, cabe resaltar que se ha independizado la distancia porque a la hora de analizar el comportamiento del sistema en las pruebas es fácil definir un conjunto de test para comprobar la validez independientemente.

5.1.3. Resolvedor de mínimos cuadrados.

De acuerdo a la formulación desarrollada en la fase de análisis pasamos a la parte más compleja del algoritmo, pues dado un conjunto **n** de puntos, aplicaremos la teoría de mínimos cuadrados para obtener el vector

$$Q = \begin{pmatrix} X \\ Y \\ \Theta \end{pmatrix}$$

que se devolverá como solución una estimación del sistema de coordenadas que relaciona las parejas de puntos correspondientes al $M_{c_i p_i}$.

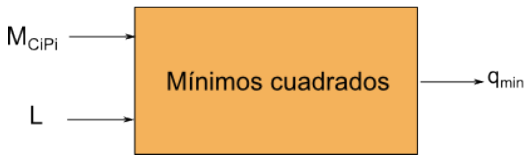


Figura 5.2: Resolvedor de mínimos cuadrados

Esta función tendrá como entradas:

- Matriz de correspondencias cuya proximidad es inferior a d_{min}, llamada $M_{C_i P_i}$
- Valor de la distancia L.

Tendrá como salida:

- Vector q_{min} que representa el valor q que minimiza $E_{dist}(q)$ como se ve en la ecuación 1.2

El funcionamiento del algoritmo es el que sigue:

1. Se calculan los coeficientes de la Matriz A y se monta la matriz.

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{12} & a_{22} & a_{23} \\ a_{13} & a_{23} & a_{33} \end{pmatrix}$$

$a_{11} = \sum_{i=1}^{n} 1 - \frac{p_{iy}^2}{k_i}$
$a_{12} = \sum_{i=1}^{n} \frac{p_{ix} p_{iy}}{k_i}$
$a_{13} = \sum_{i=1}^{n} -c_{iy} + \frac{p_{iy}}{k_i} (c_{ix} p_{ix} + c_{iy} p_{iy})$
$a_{22} = \sum_{i=1}^{n} 1 - \frac{p_{ix}^2}{k_i}$
$a_{23} = \sum_{i=1}^{n} c_{ix} - \frac{p_{ix}}{k_i} (c_{ix} p_{ix} + c_{iy} p_{iy})$
$a_{33} = \sum_{i=1}^{n} c_{ix}^2 + c_{iy}^2 - \frac{1}{k_i} (c_{ix} p_{ix} + c_{iy} p_{iy})^2$

2. Se calculan los coeficientes de la vector vertical B y se crea el vector.

$$b = \begin{pmatrix} \sum_{i=1}^{n} c_{ix} - p_{ix} - \frac{p_{iy}}{k_i} (c_{ix} p_{iy} - c_{iy} p_{ix}) \\ \sum_{i=1}^{n} c_{iy} - p_{iy} + \frac{p_{ix}}{k_i} (c_{ix} p_{iy} - c_{iy} p_{ix}) \\ \sum_{i=1}^{n} \left[\frac{1}{k_i} (c_{ix} p_{ix} + c_{iy} p_{iy}) - 1 \right] (c_{ix} p_{iy} - c_{iy} p_{ix}) \end{pmatrix}$$

donde $k_i = p_{ix}^2 + p_{iy}^2 + L^2$.

3. Se calcula la inversa de la Matriz A y se multiplica por B.

$$q_{min} = -A^{-1} b$$

4. Se genera el vector q_{min} de salida.

Curiosamente los pasos de esta función son rápidamente expresables en código, pues la plataforma Matlab esta expresamente especializada en calculo vectorial y matricial. Haciendo sencilla la translación de las formulas matemáticas del documento de referencia al código Matlab.

5.1.4. Ficheros de datos sensibles y datos relevantes.

A continuación describiremos la estandarización que han de cumplir los datos leídos de un fichero. En los ficheros de simulaciones reales extraídos del repositorio de Radish [19] podemos obtener datos adicionales a los requeridos para la simulación del algoritmo, como es el caso de lecturas de sonar, sensores de tacto, etc.

Por ello, vamos a fijar los datos que nos son relevantes para nuestro algoritmo:

- Estimaciones de posición del robot, en base a la tupla (x, y, Θ), que llamaremos M_Q
- Capturas del sensor laser, que llamaremos M_p.
- Ángulo e incremento de cada captura laser, que llamaremos V_{rango}.
- Estampas de tiempo de cada captura de posición-laser, que llamaremos V_t.

Esto permitirá por un lado dibujar las capturas en pantalla así como el recorrido estimado sensorialmente que va trazando el robot durante la simulación. Por otro lado, la información se empleará como datos de entrada en la ejecución del algoritmo a estudio, elaborando una corrección en el error percibido por los sensores de odometría del robot.

Por otro lado el resto de la información será descartada pues supone un coste adicional de recursos que no serán necesarios para las simulaciones ni para el funcionamiento del algoritmo. El interfaz requiere un poco más de información y por tanto, se rescatarán la estampa de tiempo y el rango de capturas del sensor.

Por último destacar que estas variables dentro del sistema son únicamente de lectura sus datos no podrán ser modificables en ningún momento. Serán utilizadas como valor invariable para la visualización de interfaz gráfico y la simulación.

5.1.5. Representación de datos.

Cabe destacar que el módulo cuya funcionalidad es la gestión de los datos de entrada usados por el interfaz gráfico requiere una representación concreta para la transmisión adecuada de datos entre los ficheros de entrada y el simulador del algoritmo MbICP en Matlab.

Para ello el algoritmo MbICP requiere para su ejecución en cada instante t de:

- Valor posición del robot en instante anterior

$$Q_t = \begin{pmatrix} x_t \\ y_t \\ \theta_t \end{pmatrix}$$

- Captura del sensor de rango en el instante i con relación a Z_{ref}

$$p_i \rightarrow \forall i = \{1 \dots n\}$$

- Captura nueva del sensor rango en el instante $i+1$ con relación a Z_{new}

$$r_j \rightarrow \forall j = \{1 \dots m\}$$

Todo esto permitirá representar el camino recorrido teóricamente por el robot y el camino real realizado aplicando la corrección mediante el algoritmo MbICP determinando los errores de odometría.

Los datos usados por el interfaz gráfico son:

- El conjunto de datos obtenidos del fichero.
- Los distintos valores que va tomando en cada iteración del algoritmo MbICP

$$Q_k = \begin{pmatrix} x_k \\ y_k \\ \theta_k \end{pmatrix} y\, p_{i,k} \rightarrow \forall i = \{1 \dots n\}$$

- El conjunto de puntos que se correlacionan en cada iteración

$$C_i = \underset{r_j \epsilon Z_{new}}{argmin} \left\{ d\left(p_i, q_k\left(r_j\right)\right) \; and \, d\left(p_i, q_k\left(r_j\right)\right) < d_{min} \right\}$$

- La salida final del algoritmo.

$$Q_{sol} \; y \; p_{i,sol} \rightarrow \forall i = \{1 \dots n\}$$

5.1.6. Fases y datos.

De acuerdo a las fases anteriormente mencionadas describiremos a continuación cuándo y en qué forma son necesarios los datos entre las distintas fases del algoritmo.

- Para la primera fase son necesarios los siguientes datos
  - L: valor de distancia estimado experimentalmente.
  - p_i: vector capturas que tomamos de referencia.
  - $c_i \, \epsilon \, R_i^0$: vector con las capturas correspondientes sin alterar provenientes del sensor.
  - Q; Matriz de transformación.
- Para la segunda fase es necesario:
  - La matriz de puntos correlacionados del paso anterior, $M\left(p_i, q\left(c_i\right)\right)$
  - L: valor de la distancia estimado experimentalmente.
- En la última fase es necesario:
  - q_{min}: Resultado de aplicar el algoritmo.
  - q_k: estimación actual.

5.1.7. Datos de salida.

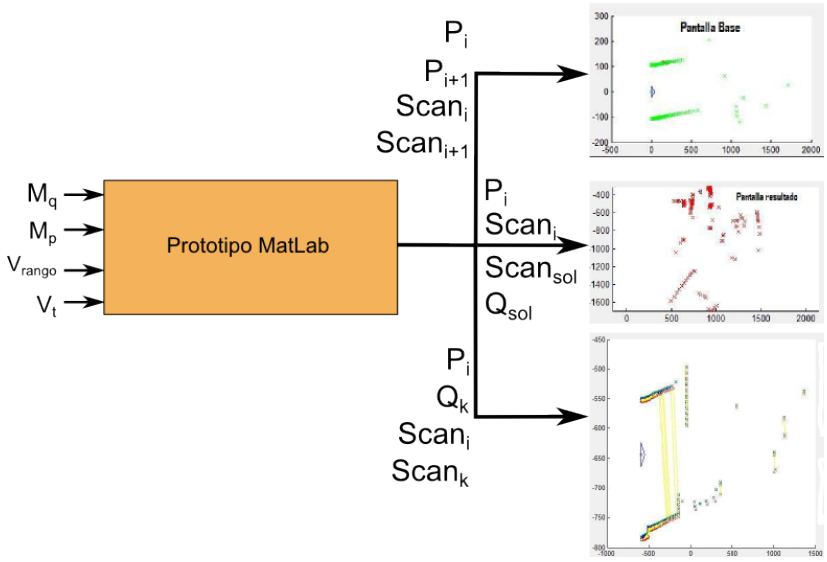


Figura 5.3: Datos entrada/salida manejados desde el prototipo a las gráficas de visualización

La salida que se le proporcionará a la ejecución del algoritmo implica un gran conjunto de variables a estudio, por tanto se propone para el diseño hacer un estudio de las variables implicadas, que son:

- Variables asociadas a los datos provenientes del fichero (más información, ver sección 5.1.5).
- Variables asociadas a la ejecución del algoritmo (más información, ver sección 2.2).
- Variables resultantes de las salidas del algoritmo como solución (ver sección 5).

En base a esto podemos anunciar que hasta el momento se tienen 10 variables de las cuales: 4 son matriciales y 6 vectoriales. Por tanto, la información que tenemos para mostrar en pantalla es compleja de interpretar y para ello hemos decidido recurrir a dos gráficas y un pequeño mapa donde se observará la evolución en el plano 2D del movimiento del robot por la superficie. La primera gráfica será elaborada en base al error cometido con respecto al tiempo de simulación. El segundo estudiará la convergencia del algoritmo con respecto al tiempo. Además se incluirá una tercera pantalla donde se pondrá la evolución en cada instante de tiempo del algoritmo teniendo como ejes en pantalla (x, y, θ) y veremos como va evolucionando la superficie en la iteraciones del algoritmo hasta darse por buena la solución.

Capítulo 6

Prototipo en Matlab.

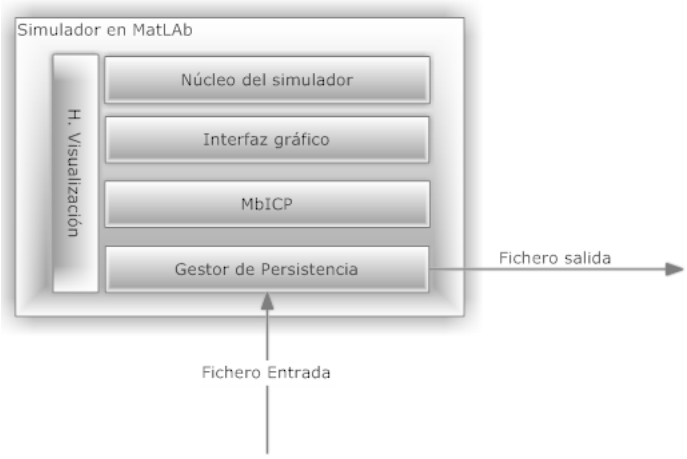


Figura 6.1: Arquitectura del prototipo en Matlab

Para hacernos una visión global del sistema la figura 6.1 nos muestra como está compuesto este prototipo asi como los módulos que lo integran.

6.1. Diseño núcleo Matlab.

En primer lugar, describiremos el funcionamiento del algoritmo que empleará el prototipo en Matlab del que forma parte el MbICP. Partimos de la idea que los datos de la simulación se encuentran en un fichero externo al sistema y que previo al cálculo es necesario leer los datos y almacenarlos.

1. Carga de datos simulación.

2. Simular avance del robot, bucle principal.

   a) Procesar para cada punto, aplicar MbICP, para corregir el error, función "algoritmo_MbICP".

   b) Visualizar valores reales y corrección del algoritmo.

   c) Continuar avanzando en paralelo con la distancia corregida y la obtenida por la odometría.

3. Generar estadísticas, comparativas, fichero de resultados, etc.

Por el momento este bucle no describe la visualización de datos con respecto al interfaz gráfico, ya que es una parte adicional que se ejecutará en paralelo al funcionamiento interno del núcleo de simulación. En suma, cada paso del bucle principal de la aplicación se describirá en los puntos sucesivos.

6.2. Gestor persistencia.

El gestor de persistencia es la capa intermedia entre el sistema de archivos y la aplicación su cometido será:

1. Comprobar la existencia del fichero de datos de sensores.
2. Abrir el fichero en modo lectura.
3. Determinar el formato de representación que presenta el fichero.
4. Procesar el fichero de acuerdo a su formato.
5. Validar los datos de salida.
6. Cerrar el fichero.

Esto nos permitirá abstraer la aplicación de como están almacenados los datos en el fichero, así como del formato en el que están codificados. Pudiéndose añadir nuevos formatos de simulación sin alterar la aplicación como tal, mediante la adición de la función correspondiente para el procesamiento de cada formato. Actualmente los formatos soportados son: ficheros con la extensión ".script" y ficheros con la extensión ".log", cuya descripción en detalle podemos encontrarlo en el apéndice C.

El objetivo de este módulo es realizar una tradución de los ficheros de datos existentes a un formato común con extensión ".mat" propio de Matlab, optimizado para trabajar con esta plataforma. Para ello, los datos almacenados en cada fichero a la representación interna del prototipo en Matlab, expuesta en la sección 5.1.5, creando así una representación homogénea a la hora de procesar los datos dentro del sistema.

6.2.1. Generar estructuras de datos.

A la hora de cargar los ficheros de datos de simulación existentes, es necesario tener en cuenta el formato en el que están almacenados dichos datos de simulación. Para ello, se establece que se tendrá una función de carga de datos por cada formato soportado por la aplicación. El cometido de esta función será extraer de las líneas del fichero los datos relevantes para la simulación, almacenándolos

de acuerdo descrito en el gestor de persistencia (6.2). Completado el proceso de carga se procederá a validar los datos, comprobando que todas las líneas poseen datos válidos, que poseen las dimensiones esperadas y se han suprimido los valores superfluos innecesarios para la simulación.

Por tanto el proceso iterará en la siguiente forma:

1. Apertura del fichero.
   - *a*) Comprobar que el nombre existe.
   - *b*) Abrir el fichero
   - *c*) Comprobar que el fichero se ha abierto con éxito.
2. Procesar el fichero.
   - *a*) Leer línea
   - *b*) Analizar línea.
     1) Seccionar datos.
     2) Almacenar según tipo.
     3) Descartar datos no relevantes para la representación.
   - *c*) Comprobar consistencia datos.
   - *d*) Avanzar Linea y repetir paso (b) hasta completar todo el fichero.
   - *e*) Comprobar y validar las estructuras de datos.
3. Atributos solo lectura en las estructuras datos.

Las comprobación de consistencia de cada línea se basa en comprobar si las dimensiones del vector generado coinciden con los anteriores valores almacenados en las matrices, así como el hecho de que se han rellenado de acuerdo al formato del fichero. Finalmente se validará que todas han sido rellenadas de acuerdo a la representación de datos elegida, comprobando que sus dimensiones coínciden, pues todas han de tener el mismo número de capturas de posición, barridos del sensor de rango y estampas de tiempo.

6.3. Interfaz gráfico.

El sistema ha de ser capaz de cargar desde los ficheros almacenados en el disco, los datos de simulación. Esta operación se basa en ficheros con extensión ".mat" puesto que su carga es veloz y sencilla por parte de la aplicación al ser un formato propio de Matlab. En caso de querer, cargar fichero con una extensión distinta o en un formato específico véase la sección Gestor de Persistencia (6.2).

Por tanto, el primer paso es cargar los datos de simulación dentro de la aplicación, para evitar que el usuario intente realizar una simulación, se imprimirá un mensaje de error mediante un cuadro de dialogo que resalte que es necesario cargar los datos en la computadora antes de comenzar cualquier tipo de ejecución. Facilitando con esto la percepción del usuario para conocer los motivos por los cuales no es posible realizar la simulación.

Una vez llevada a cabo la carga de datos, la botonera de control que nos permitirá simular, paso a paso o haciendo uso del temporizador que mostrará el siguiente movimiento del robot cada

cierto intervalo de tiempo. Unido a estos modos de funcionamiento se incorpora un botón de parada y otro para reiniciar la simulación desde el principio. Por otro lado, se tiene previsto contemplar la posibilidad de la adicción de un botón de retroceso, permitiendo regresar a iteraciones previas, como puede verse en la figura 6.2.

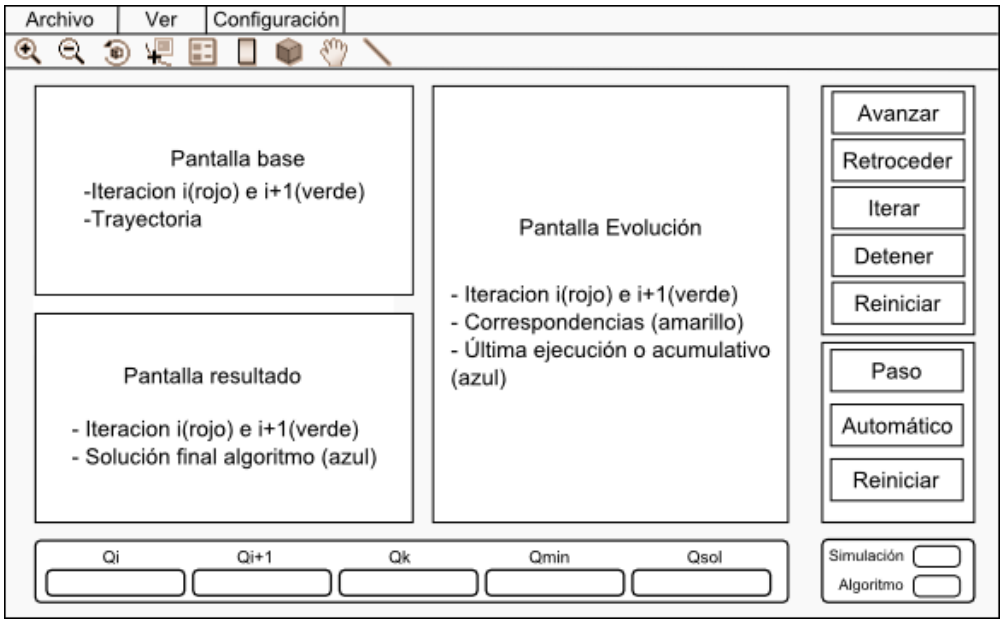


Figura 6.2: Esqueleto de la interfaz gráfica

Panel de control

Figura 6.3: Barra de navegación

El panel de control dispone de útiles cuyo objetivo es aportar una funcionalidad adicional a la aplicación como es el manejo de ficheros mediante el menú desplegable de **"archivo"**.

Figura 6.4: Desplegable del navegador "Archivo"

En él actualmente se ha incluido la funcionalidad **"Abrir"** cuyo objetivo es cargar un fichero desde una ubicación del disco sobre la aplicación Matlab, transformando la información en él contenida al formato explicado previamente por medio del gestor de persistencia (sección 6.2). No obstante, en caso de dicho fichero encontrarse ya con extensión ".mat" simplemente se realizará una carga del fichero que ya se encuentra en el formato necesario para su ejecución dentro de la aplicación.

Seguidamente, tenemos el menu **"Ver"** encargado de seleccionar el nivel de detalle que queremos observar tanto en la aplicación como en las distintas pantallas de simulación.

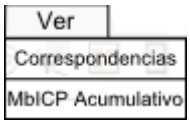


Figura 6.5: Desplegable del navegador "ver"

En principio, se dispone de la manipulación del contenido visionado en la pantalla de evolución, mostrando únicamente la iteración actual y siguiente, así como la última corrección realizada por el algoritmo. Sin embargo este menú nos permite seleccionar:

- **Correspondencias**: si esta seleccionada esta utilidad visualiza las correspondencias llevadas a cabo en el primer paso del algoritmo MbICP.
- **MbICP Acumulativo**: Al seleccionarla permite ver iterativamente como evoluciona la corrección del error odométrico a medida que itera el algoritmo.

Finalmente tenemos el menú de **"configuración"** que permitirá al usuario que realice la simulación modificar los parámetros sobre los que se hace la simulación.

Figura 6.6: Desplegable del navegador "configuración"

Parámetros MbICP.

A la hora de configurar la simulación existen algunos parámetros a tener en cuenta y para solventarlo se ha optado por crear un menú de configuración que aparece en la figura inferior.

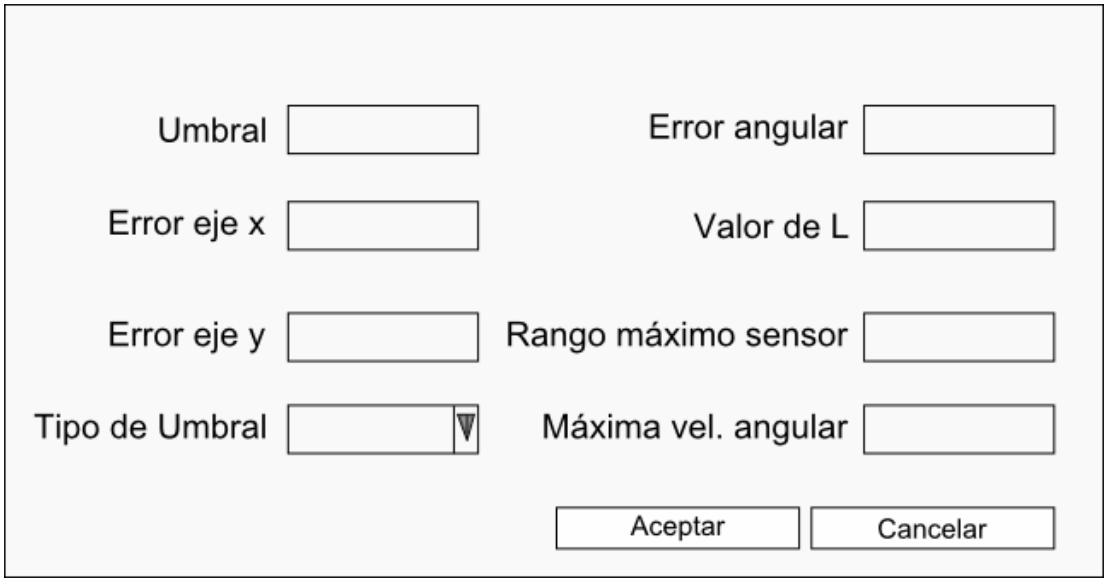


Figura 6.7: Esqueleto de la ventana de configuración de parámetros

Como se puede observar este panel de control esta plenamente orientado a configurar los parámetros de manejo del algoritmo MbICP y aunque su funcionamiento ha sido descrito en el análisis haremos especial mención a las unidades empleadas y su funcionamiento.

- Umbral: Está en mm y mide el máximo valor por el cual se considera que una pareja de puntos tiene correspondencia en el primer paso del algoritmo MbICP.
- Error $[X, Y, \theta]$: Se mide en $[mm, mm, rad]$ se encarga de evaluar si hemos encontrado la adecuada corrección de odometría y finalizamos la ejecución del algoritmo.
  - El criterio de convergencia del algoritmo MbICP es el siguiente:

$$q_{min} < q_{error} \equiv x_{min} < x_{error}\, and\, y_{min} < y_{error}\, and\, \theta_{min} < \theta_{error}$$

- Valor L: Esta en mm y su cometido es dar un valor al parámetro L de la distancia 2.23 en la página 26.
- Rango máximo sensor: se mide en mm y marca el límite del sensor de rango, cualquier punto con una medida superior se considerará ruido y será ignorado.
- Tipo de Umbral: Designa el tipo de umbral a emplear en la selección del punto sus valores son: *Estatico*, *Dinámico y Angular*, este tipo designa como se emplearán los valores de Umbral y Máxima Vel. Angular en la aplicación.
  - Umbral estático: El valor de umbral se calibra en función de las dimensiones del robot y permanece constante durante la ejecución del robot.

  - Umbral Dinámico: El valor de umbral decrece conforme la distancia ecuclídea aumenta entre los puntos, filtrando aquellos puntos que pueden dar una correspondencia erronea al estar proximos a la normal.
  - Umbral Angular: Se estima el valor de rotación angular del robot y se usa este parámetro para calcular el umbral en función de la velocidad angular del robot. Para ello se hace uso del parámetro Máxima Velocidad Angular que limita los posibles valores estimados durante la ejecución del robot.

- Máxima Velocidad Angular: Se mide en rad/s y marca cual es la máxima velocidad angular de rotación del sensor.

Cálculo de distancia.

El siguiente menú presentado consiste en la configuración de los parámetros que se manejarán en la visualización de la superficie 3D que se visualizará cuando sea presionado el botón  de la barra de herramientas que veremos posteriormente.

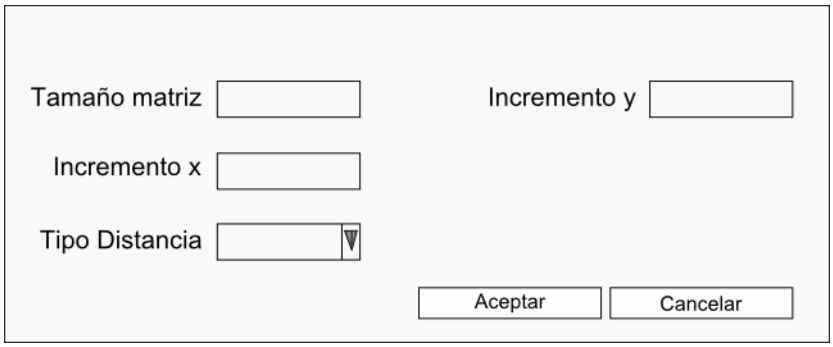


Figura 6.8: Esqueleto ventana configuración de la susperficie de correspondencias.

Como se puede observar los parámetros a priori son algo confusos de entender pero una vez expuestos son claramente intuitivos de manejar.

- Tamaño matriz: Definirá el tamaño de la matriz $n \times n$ donde n es este tamaño precisamente. Cuanto mayor incrementemos el valor de este parámetro mayor será el conjunto total de puntos de la superficie aumentado con ello el grado de detalle, ya que la discretización de la función de distancia a muestrear depende de este factor.
- Incremento $[x, y]$: Este incremento se mide en $[mm, mm]$ y define cuantas unidades hay de diferencia entre cada punto y su vecino adyacente, de forma que al incrementar este valor los puntos se alejarán proporcionalmente en distancia del centro, cuya proporción la definen estos incrementos.
- Tipo de Distancia: Define la función que se aplicará a los puntos obtenidos para calcular la distancia, actualmente disponemos de las *Distancia MbICP y Distancia Euclídea*

Barra de herramientas

Figura 6.9: Esqueleto barra de herramientas

Esta barra de herramienta es la encargada de manipular la visualización de las distintas pantallas, cuyo funcionamiento explicaremos en la siguiente sección, permitiendo obtener los siguientes resultados en función de la utilidad seleccionada:

- Permite ampliar el tamaño del punto o la zona de interés, falicitando así el análisis de puntos de pequeño tamaño.
- Permite alejarnos del punto o la zona sobre la que se aplique la herramienta y en caso de hacer doble click la imagen vuelve al estado previo antes de usar la herramienta.
- Es muy útil en casos, donde la orientación del robot nos confunda porque nos permite rotar la medidas impresas en pantallas adecuándolas a nuestras necesidades.
- Muestra la posición $[X, Y]$ de un punto seleccionado en dicha pantalla, cuando se selecciona otro la etiqueta cambia de lugar.
- Muestra la leyenda de los puntos.
- Organiza los puntos de acuerdo a la densidad de puntos.
- Genera una superficie 3-D en torno al punto seleccionado con la herramienta y cuyos parámetros se encuentran en "Configuración→Cálculo de distancia".
- Permite que nos desplacemos a lo largo de la superficie gráfica seleccionada navegando por las diferentes zonas de interés.
- Dado un punto seleccionado muestra la normal intrínseca a ese punto.

Pantalla base

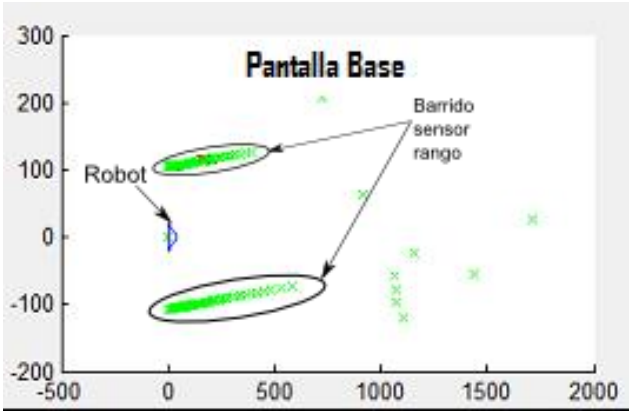


Figura 6.10: Representación directa de los datos del fichero

Esta pantalla nos otorga una visión global de la información extraída del fichero elegido por el usuario. Comenzará por mostrar el movimiento real producido por el robot durante la simulación, mediante una trayectoria definida en rojo, así como las lecturas de los sensores en ese instante y el instante posterior permitiendo conocer que próxima lectura le corresponderá. En suma se dibujará el robot haciendo uso de un triángulo azul cuyo vértice más elongado determinará el eje X (eje frontal de movimiento).

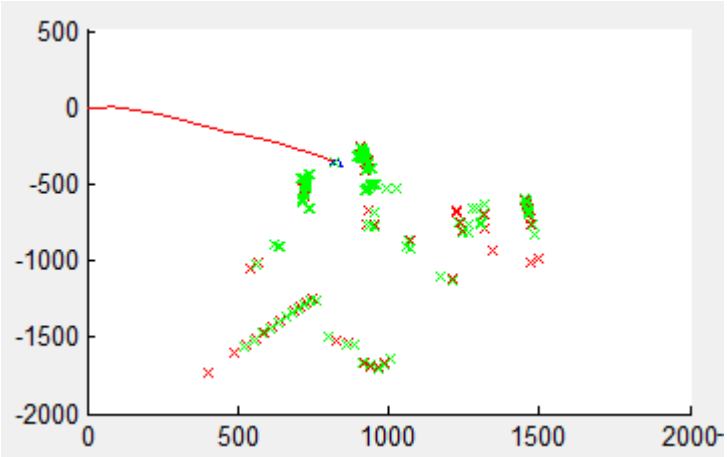


Figura 6.11: Trayectoria de robot tras la n-ésima iteración

Pantalla resultado

Esta pantalla es la encargada de llevar a cabo la representación de la lectura actual, mediante el color rojo, y la nueva lectura calculada por el algoritmo MbICP en color azul, junto con la orientación y posición del robot mediante su representación con un triángulo cuyo vértice más prolongado está sobre el eje X, por el cual se desplaza el robot.

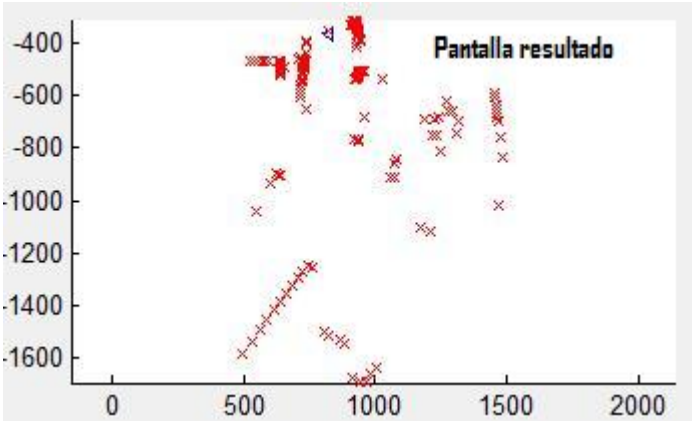


Figura 6.12: Posición i-ésima iteración previa a la correción

De esta forma, podemos observar que si no se lleva a cabo la ejecución del algoritmo hasta hallar la solución la pantalla "resultado" aparecerá únicamente con el color rojo de la lectura actual del robot en el instante i y cuando se alcance una solución que cumple los términos del algoritmo, se representará superpuesta una segunda lectura de color azul resultante de haber aplicado el algoritmo MbICP.

Pantalla evolución

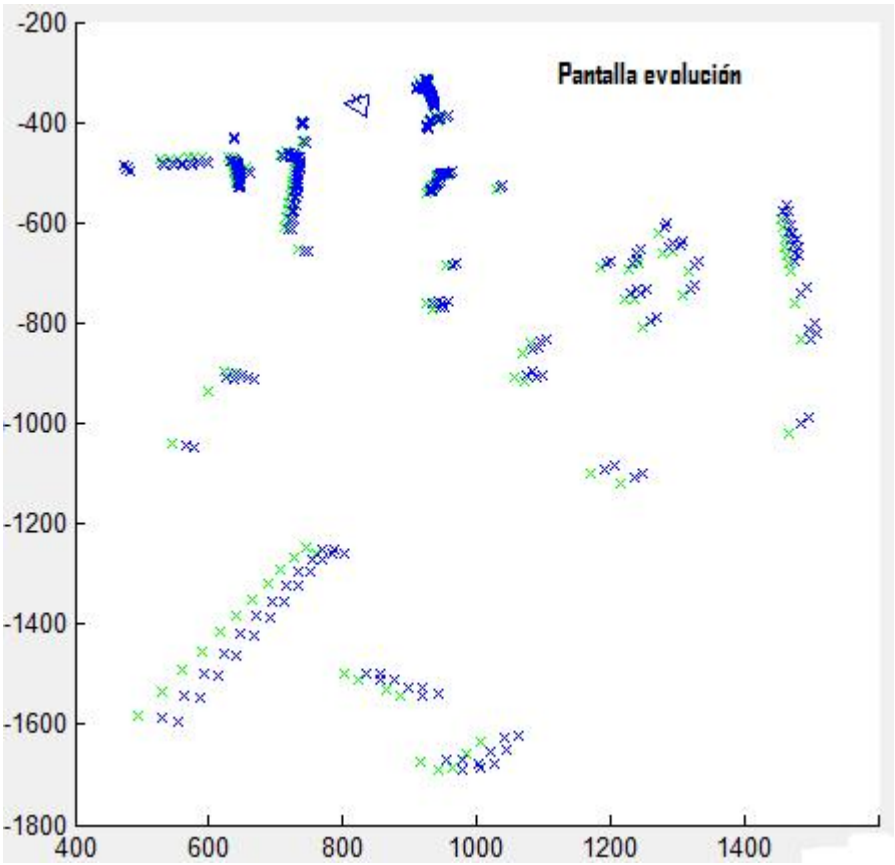


Figura 6.13: Correción realizada en la i-ésima iteración

Esta pantalla de mayor tamaño debido a su interés, será la encargada de almacenar primeramente la lectura y representación del robot en la posición actual, sobre esta iterativamente cuando se cambia el estado de la evolución del algoritmo mediante el menú "MbICP", que veremos más adelante, irá situando en la pantalla de color azul distintos conjuntos de puntos como resultado de aplicar el algoritmo MbICP sobre la lecturas en el instante i e $i+1$, iterativamente hasta alcanzar la correspondencia entre q_i y q_k obteniendo el q_{min} que figurará en la pantalla resultado antes mencionada.

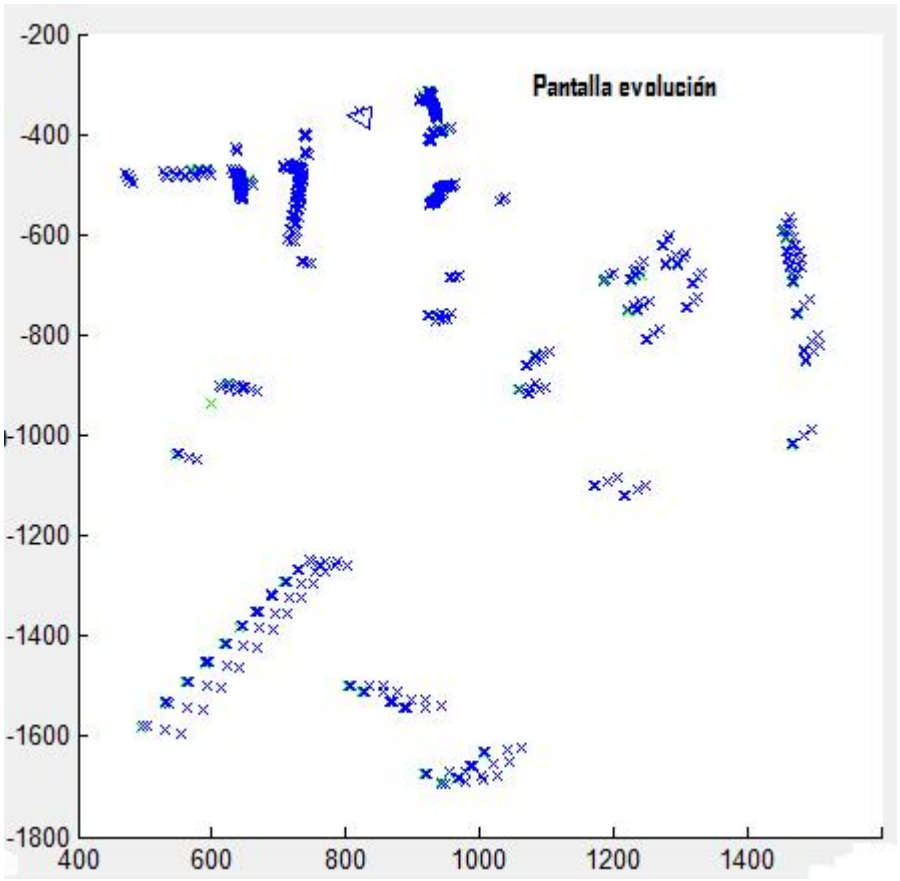


Figura 6.14: Acumulación correciones en las succesivas iteraciones

Como se puede observar en la imágenes 6.13 y 6.14 inicialmente el error de posición de robot da un gran error pero conforme iteramos se van ajustando los puntos hasta converger a la solución deseada. Por consiguiente, esta pantalla nos permite seguir la evolución de las distintas iteraciones del algoritmo MbICP con relativa facilidad.

Menú simulador

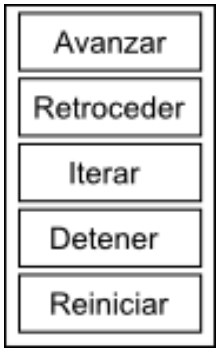


Figura 6.15: Herramientas de reproducción

Este menú nos permite fácilmente avanzar y retroceder en los datos extraídos del fichero con facilidad de forma que en cada instante podemos verificar rápidamente los resultados obtenidos en una iteración aparentemente errónea y extraer conclusiones precisas del funcionamiento del algoritmo. Por tanto, facilita enormemente la labor de depuración y análisis de los datos. Diseño del prototipo en Matlab.

No obstante hay que tener en cuenta que estas herramientas de iteración no implican la ejecución del algoritmo MbICP, a excepción de un caso que explicaremos más adelante en este apartado, visualizándose únicamente los datos cargados por el usuario haciendo uso del gestor de persistencia (Véase sección 6.2 en la página 80) apareciendo parcialmente completas las pantallas de "evolución" (Véase 6.3 en la página 89) y "resultado" (Véase 6.3 en la página 88).

Seguidamente, vamos a describir la utilidad que supone cada función en el menú de herramientas:

- Avanzar Este botón tiene como objetivo avanzar en la visualización de los datos. Es el encargado de definir el instante i y actualizar las pantallas "base" "resultado" y "evolución" así como los valores "q_i" y "q_{i+1}" del panel de variables de interés. Definiendo con esto el nuevo estado de la aplicación en un instante posterior, transformando $q_i = q_{i+1}$ después de pulsar el botón. No obstante, solo es aplicable si el instante i es inferior al tamaño muestral extraído del fichero, en otro caso, se imprimirá por pantalla el siguiente mensaje de error.

Figura 6.16: Dialogo de error, no hay más datos en el fichero

- Retroceder Este botón transformará el estado de la aplicación del instante i al $i-1$ y actualizará la pantalla de igual forma que en el botón "Avanzar". No obstante, el botón solo será aplicable cuando el instante i sea 2 o superior, en otro caso aparecerá el siguiente mensaje de error.

Figura 6.17: Dialogo de error, se encuentra al principio del fichero

- Iterar Este botón permite iterar de forma continuada por los datos automáticamente sin necesidad de que el usuario intervenga durante la evolución del robot por el espacio de simulación. Sin embargo, posee una característica especial y es la posibilidad de ir aplicando el algoritmo MbICP durante el proceso de avance del robot, facilitando al usuario seguir la evolución del error y su respectiva corrección conforme avanza la simulación. Esto se llevará a cabo mediante un diálogo de pregunta que tiene la siguiente forma:

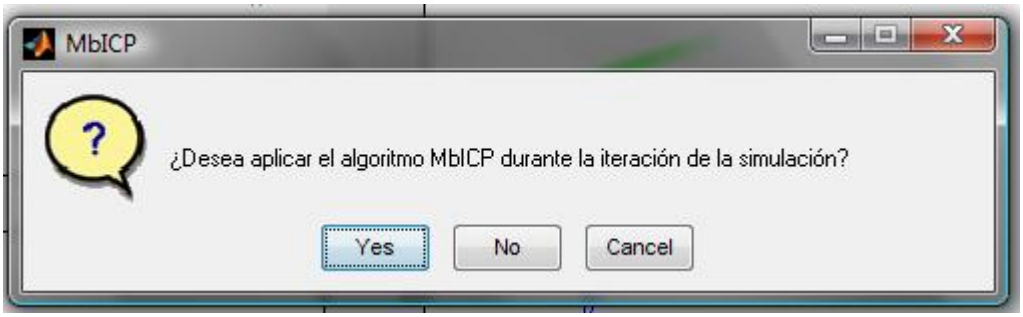


Figura 6.18: Dialogo selección: usar algoritmo MbICP en la reproducción

Esto permite indicar si se desea aplicar simultáneamente el algoritmo MbICP o simplemente ver la evolución del robot por los datos extraídos del fichero cargado.

- Durante cada iteración una vez presionado "Yes" se aplicará el algoritmo en cada instante i hasta obtener el resultado y una vez obtenido el resultado se avanzará al instante $i+1$ y se aplicará sucesivamente este proceso.
- Si es presionado "No" se iterará automáticamente como si, presionáramos sucesivamente el botón avanzar y continuará hasta que se presione el botón "Detener" o se llegue al final de los datos a simular.
- Aun si por error se presionó este botón es posible seleccionar "Cancel" que evita lanzar cualquiera de las dos opciones previas.

- Detener En caso de estarse ejecutando la simulación iterativamente de forma automática, detendrá la ejecución en otro caso no tendrá efecto.

- Reiniciar Tiene como cometido poner el iterador del instante i en la primera posición de la simulación, lo que equivale a $t = t_0$

Por consiguiente, cuando se cumplen todas las condiciones para su ejecución se llamará al gestor de dibujo (Véase sección 6.3.1) para que realice la actualización acorde al siguiente esquema:

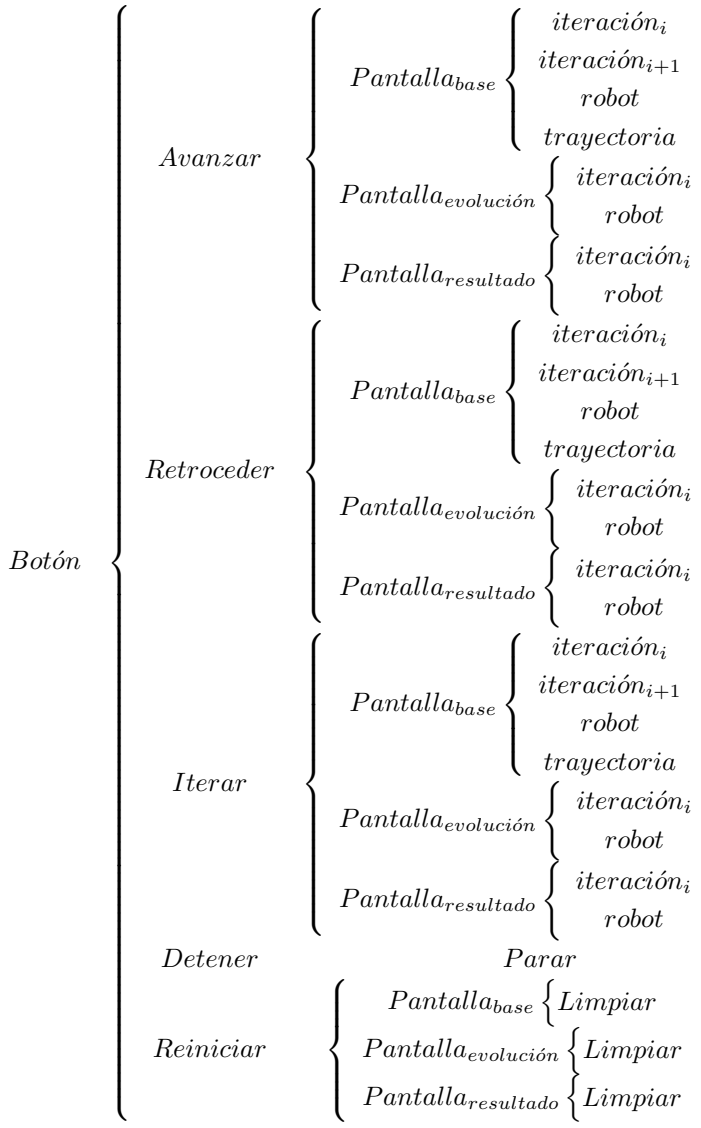


Figura 6.19: Esquema de las posibles acciones de reproducción

Menú algoritmo MbICP

Figura 6.20: Herramientas de ejecución del algoritmo MbICP

Este menú es el encargado de poner a funcionar el algoritmo de MbICP para corregir el error en la odometría del robot. El sistema consiste en que cada instante i, la aplicación facilitará la posibilidad de ver como se resuelve el mapeo entre la captura en el instante i y en el instante $i+1$, en función de la opción seleccionada en el menú:

- Paso Aplicará una vez el algoritmo MbICP y como salida actualizará los valores de la variables q_k, q_{min} y q_{error}, e imprimirá en color azul la nueva lectura de sensor con la nueva aproximación en la pantalla de evolución (Véase 6.3 en la página 89). Además, en cada pulsación se incrementará el iterador sobre el algoritmo de forma que cuando encuentre una solución, saldrá el siguiente mensaje y se imprimirá la lectura final sobre la pantalla de resultado (Véase 6.3 en la página 88).

Figura 6.21: Dialogo informativo: Finalizada la ejecución del algoritmo

- Automático Aplicará el método $n - veces$ hasta alcance que $q_{error} < error_{maximo}$ o $n > n_{max}$. En ambos casos se irá imprimiendo sucesivas ejecuciones del algoritmo hasta que se produzca la condición de parada. En el caso que se halle la solución se actualizará la pantalla resultado, en cualquier otro caso no se modificará.

- Reiniciar Reiniciará a cero el número de iteraciones ejecutadas de forma que el algoritmo volverá al estado inicial, antes de la ejecución de cualquiera de las dos herramientas anteriores.

Por otra parte, añadir que existe otro motivo de parada del algoritmo MbICP y es que no se encuentre correspondencia para los puntos de los diferentes scans en cuyo caso el algoritmo abortara su ejecución imprimiendo el siguiente mensaje de error.

Figura 6.22: Dialogo error: no se pudieron establecer correspondencias.

Por consiguiente, tenemos se nos presenta el siguiente esquema que dirige la visualización de los resultados obtenidos

$$
Botón \begin{cases} Paso \begin{cases} si\, iterador < N_{max} \rightarrow Pantalla_{evolución} \{ Lectura_k \\ si\, error < error_{umbral} \rightarrow Pantalla_{resultado} \{ Lectura_{corregida} \end{cases} \\ Automatico \quad \{Aplicar \quad Botón \rightarrow paso \\ Reiniciar \begin{cases} Pantalla_{base} \{Limpiar \\ Pantalla_{evolución} \{Limpiar \\ Pantalla_{resultado} \{Limpiar \end{cases} \end{cases}
$$

Figura 6.23: Esquema posibles acciones de las herramientas del algoritmo MbICP

Variables de interés.

Figura 6.24: Variables de interés inspeccionadas

Como su nombre indica este panel se encarga de mostrar los datos relevantes tanto de posición en el movimiento en cada instante i del robot como los valores más destacados de la ejecución exitosa de cada iteración del algoritmo MbICP. Por tanto, pasaremos a describir el cometido de cada valor:

- Q_i : Posee el valor de la posición del robot en el instante SC_i.
- Q_{i+1}: Posee el valor de la posición del robot en el instante SC_{i+1}.
- Q_k: Marca el valor en la $n-ésima$ iteración de la estimación corregida del robot en el instante SC_{i+1}
- Q_{min}: Determina el menor valor hallado como solución de la aplicación del algoritmo MbICP
- Q_{sol}: El resultado existente de aplicar la transformación de coordenadas sobre el SC q_k hasta llevarlo al SC q_{min} después de aplicar la ecuación 2.43:

$$q_{sol} = q_{min} \oplus q_k \equiv^{sol} A_i^{i+1} = A_i^{min} A_i^k$$

.

Por otro lado, estas variables solo serán actualizadas si se produce un cambio en el estado de la aplicación y su valor será expresado en milímetros para la distancia y radianes para la orientación.

Iteradores de guía

Figura 6.25: Herramientas de navegación rápida

Son los encargados de conocer el estado actual de la simulación tanto la iteración de los datos en la que nos encontramos como el valor de pasos realizados por el algoritmo MbICP en la búsqueda del valor solución en ese instante.

6.3.1. Gestor de dibujo.

A la hora de situar en pantalla cada iteración es necesario tener en cuenta que existe un orden de impresión de los datos. Este gestor será el responsable de ir situando en pantalla ordenadamente cada uno de los datos extraídos en la simulación, además de ser el encargado de preparar la pantalla donde se expondrán los datos.

Los datos necesarios a tener en cuenta para la impresión de la pantalla en cada instante

- Pantalla donde se desea pintar, Tipo: “Handles”.

- Posición eje de referencia $[x, y, \theta]$ donde esta situado el robot en el instante i
- Dimensiones aproximadas del robot (Parámetros fijos), tipo vector con dos enteros [ancho alto].
- Lectura que se desea pintar primer lugar, tipo array de dos reales $\begin{bmatrix} x \\ y \end{bmatrix}$.
- Lectura que se desea pintar en segundo lugar, tipo array de dos reales $\begin{bmatrix} x \\ y \end{bmatrix}$.
- Trayectoria que ha recorrido, de tipo array donde en cada columna estarán situados dos reales $\begin{bmatrix} x \\ y \end{bmatrix}$.

El proceso que se lleva a cabo tiene los siguientes pasos.

1. Limpiar la pantalla
2. Pintar Eje de coordenadas
3. Pintar Robot.
4. Pintar primer conjunto de puntos.
5. Pintar segundo conjunto de puntos.
6. Pintar trayectoria.

La idea consiste en dividir cada una de estas fases en una función independiente de forma que será posible llamarlas en caso de querer pintar múltiples trayectorias, varias posiciones del robot o incluso distintas posiciones de los ejes. Todo esto dentro del marco del gestor de pintura encargado de llamar en cada instante a la función correspondiente.

Actualmente se llevan a cabo solamente los pasos expuestos con anterioridad, con la salvedad de que alguno de los parámetros esté vacío en cuyo caso será ignorado en el proceso de pintado y se saltará al siguiente que cumpla la precondición de tener al menos un elemento.

6.3.2. Representar Datos simulación.

En este apartado nos encargaremos de diseñar el sistema que nos permitirá observar la evolución del robot, en función de los datos capturados desde el fichero.

Los datos provenientes de cada fichero independientemente del formato en el que están escritos ha de tener la siguiente estructura:

- La posición en cada instante viene descrita por una coordenadas $[X_i, Y_i, \theta_i]$ a lo que va asociado un marca temporal T_i
- El barrido del sensor laser viene descrito por una tupla de coordenadas polares por un lado contiene las posiciones $[V_1 \dots V_n]$ y por otro lado, el incremento de ángulo en radianes que supone cada muestra entre sí. Cuyo intervalo viene definido $-\frac{\Pi}{2} \leq eje_x \leq \frac{\Pi}{2}$
- TimeStamps, Relaciones entre sensor y posición, etc.

6.4. Visión final del prototipo en Matlab.

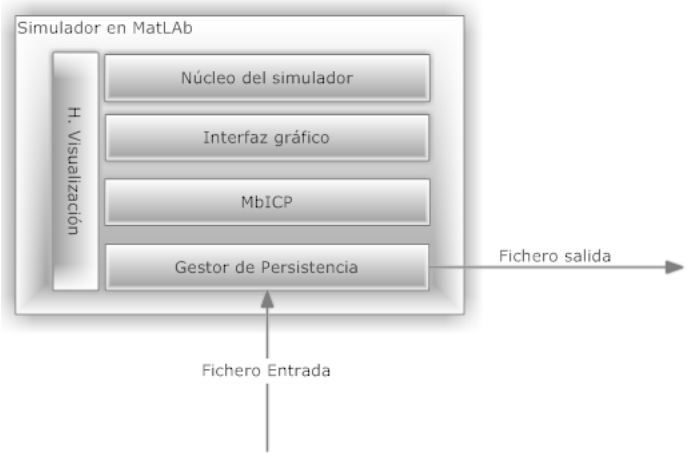


Figura 6.26: Arquitectura del prototipo en Matlab

Una vez hemos realizado el diseño y descrito cada componente pasamos a ilustrar la visión final que tiene esta aplicación como se observa en la figura 6.26. A lo largo del diseño y desarrollo hemos descrito los módulos que integran que son:

- Núcleo del simulador. Cada evento del sistema será controlado por este módulo, encargado del almacenamiento y distribución de la información por toda la aplicación.

- Interfaz gráfico. Será el encargado de presentar los datos en pantalla, de acuerdo al diseño presentado con anterioridad. Ilustrado en la figura 6.27

- Algoritmo MbICP. Este módulo destacado contendrá el algoritmo MbICP, así como las transformaciones de coordenadas entre los diferentes SC.

- Gestor de persistencia. Es el encargado de generar un formato legible por la aplicación de Matlab, así como de generar ficheros de salida de acuerdo al formato expuesto anteriormente.

- Herramientas de visualización. Se encargará de añadir herramientas para el análisis de los datos generados por la interfaz gráfica, como es la leyenda normal a un punto ó la superficie de distancias en torno al punto selecionado.

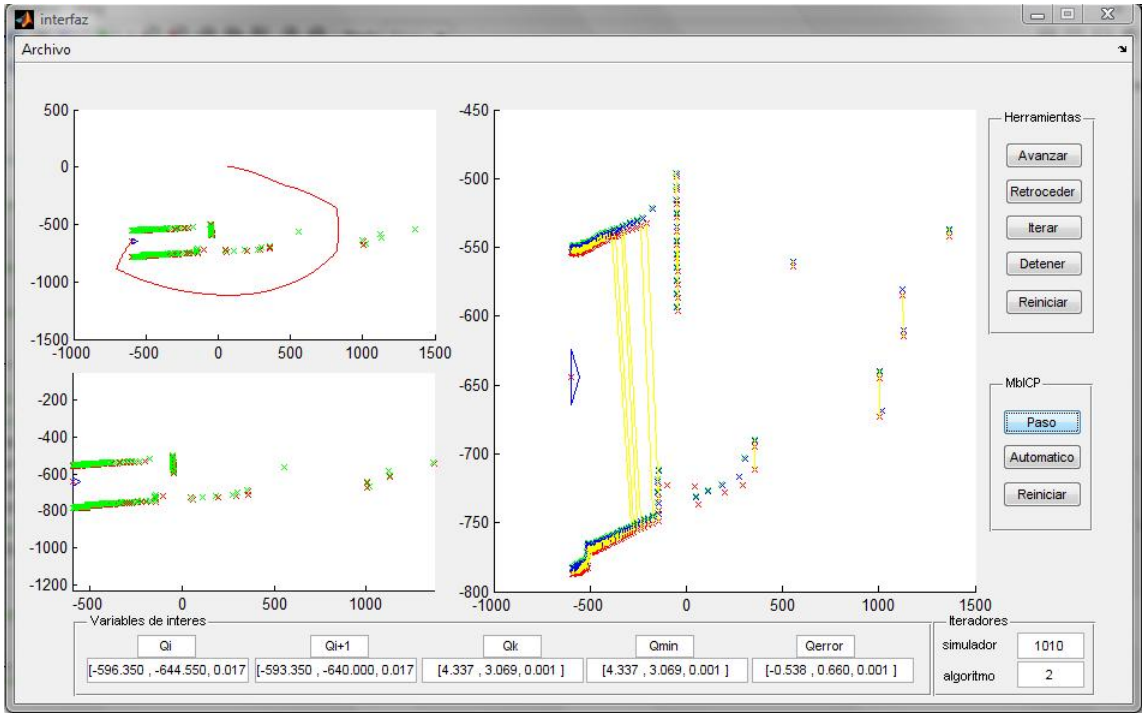


Figura 6.27: Visión final de la interfaz gráfica del prototipo en Matlab

Por otro lado, para hacernos una visión global de la interfaz gráfica del sistema tenemos la figura 6.27, que muestra el prototipo Matlab finalizado.

6.5. Test de validación del MbICP.

En primer lugar se han elegido los casos extremos donde la aplicación debería tener un comportamiento determinado a priori.

1. Cuando $q_i = q_{i+1} \rightarrow p_i = r_i \forall i \, \epsilon \, \{1, \ldots, n\}$ tenemos como resultado que q_{min} debe ser 0 pues estamos hablando de dos barridos idénticos.

2. Cuando $q_i = q_{i+1} \rightarrow \exists i \, \epsilon \, \{1, \ldots, n\} \mid p_i \neq r_i$ tenemos como resultado que q_{min} debe ser 0 puesto que no ha existo desplazamiento por parte del robot. En caso, contrario estamos hablando de que existe ruido en la captura.

3. Cuando dos lecturas en dos instantes lejanos entre sí $p_i \neq r_i \forall i \, \epsilon \, \{1, \ldots, n\}$ tenemos como resultado que nunca se encontrará correspondencia entre los puntos.

4. Cuando $q_i \neq q_{i+1}$ pero $p_i = r_i \forall i \, \epsilon \, \{1, \ldots, n\}$, caso general en el que los puntos se mueven solidariamente con el robot. Caso paradigmático. En este test se podrá observar que por

mucho que se desplace el robot el error producido será poco apreciable durante su movimiento. Incluso erroneamente se puede observar que siempre se incorporará en el caso genérico.

5. Caso general mundo estático y robot y capturas en movimiento, cuya evaluación se dividirá en:

   a) Conjunto de muestra obtenido del repositorio Radish [19], concretamente el conjunto de datos es "ap_hill_07b" confecionado por Andrew Howard.

   b) Una aplicación que crea conjuntos de datos automáticamente para la evaluación del algoritmo MbICP, bajo una serie parámetros previamente selecionados por el usuario, que describiremos con mayor detalle en el apéndice E.

En los test de prueba es necesario tener presente que se eliminará cualquier distancia superior a 8 metros pues se considera que el sensor no ha encontrado ningún obstáculo y este valor es superfluo de cara al cálculo del algoritmo MbICP distorsionando la medidas reales que son prioritarias a la hora de hallar el solapamiento de los valores.

Para el conjunto de test antes expuesto se definirán cuatro variables que regirán el tamaño de este conjunto de puntos:

- Número de puntos. Proporcionará el número de puntos a estudio durante la simulación [1].
- Número de puntos en el barrido.
- Rango para el que se definen los puntos. $(\alpha_{inicial}, \alpha_{final})$

Una vez claras las pruebas necesarias a llevar a cabo sobre el algoritmo MbICP en el prototipo en Matlab se procedió a desarrollar una aplicación capaz de generar conjuntos de puntos para evaluar el comportamiento del sistema. Esta aplicación se encuentra descrita en el apéndice E donde se describe la formulación matemática con mayor detalle.

6.6. Resultados y conclusiones del prototipo en Matlab.

Una vez fueron ejecutados los tests comentados en el apartado anterior, hemos podido extraer información valiosa. Comenzaremos por exponer los resultados de acuerdo al orden en el que se han realizado los test.

1. Recordamos consiste en puntos donde el robot no ha cambiado de posición y el entorno que rodea al robot es estático. Cuando ponemos en funcionamiento el algoritmo MbICP sobre este test, obtenemos que el punto cálculado es el mismo, consecuentemente no existe valor de correción y el valor de $q_{error} = 0$. Para todas las parejas de puntos y lecturas proporcionadas por el programa de test. A la vista de estos resultados podemos observar que **si ni el punto ni el entorno sufren variación el algoritmo devuelve la misma salida obteniendo repitibildad y consistencia,** dos aspectos necesarios para comprobar que nuestra implementación es coherente con la definición dada en [1].

[1] Recordamos que el algoritmo obliga a que cada punto lleve asociado un barrido sensorial, en otro caso, se interpolan los puntos de posición y se le asocia un único punto a la lectura

2. El segundo test valora que si se introduce algún tipo de ruido en la captura o algun objeto en movimiento que sea significativo. Cuando lo llevamos a cabo las diferentes ejecuciones del algoritmo podemos concluir como los resultados finales se ven desplazados debido a esta perturbación. En caso de ser un objeto, mientras más cercano se encuentre del robot mayor será el error producido conforme nos alejamos del punto. En cambio si se trata de un punto aislado o un conjunto de puntos disperso entre sí, será absorbido siempre y cuando el conjunto de puntos del sensor sea considerablemente superior al conjunto erróneo.

   Por todo esto, sacamos en claro que si el conjunto de puntos sensoriales es lo suficientemente grande, los errores aleatorios proximos a los valores de las lecturas serán absorbidos por los mínimos cuadrados. Sin embargo si esos puntos se alejan del conjunto de puntos inicial del sensor el resultado se verá desplazado. En conclusión **existe robustez sin sacrificar la sensibilidad** ante posibles errores aleatorios, siempre y cuando cumplan las reglas que están definidas en el algoritmo.

3. El tercer test tiene presente la regla que obliga a que **los valores deben tener continuidad entre dos instantes consecutivos**, no pueden existir grandes saltos entre los valores de captura. Esto guarda relación con las características íntrinsecas de un robot movil que devuelve lecturas sensoriales cada pocos milisegundos proveyendo de valores incrementales donde se puede apreciar un movimiento suave del robot por el entorno. En otro caso, no se cumplen una de las premisas del algoritmo y por tanto no serán evaluaves dichos instantes consecutivos.

4. El cuarto test es inmediato que viola todas las premisas que han de cumplirse para que el algoritmo converga a un resultado real. Actualmente, si el mundo se mueve solidariamente con el robot es imposible calcular errores debidos a la odometría, principalmente porque no existe dos capturas del sensor laser que divergan entre sí.

5. Finalmente el caso genérico presentó el mejor campo de pruebas, resaltando la importancia de ajustar los parámetros de simulación al caso especifico que se esta simulando, puesto que las variables $d_{max}, error_{max}, etc.$ dependen de las dimensiones del robot concreto, asi como la longitud que se encuentran los obstaculos y las capacidades sensoriales implicitas en el robot. Esto permitió ver errores como la necesidad de conocer el ángulo inicial donde se comienza el barrido, las dimensiones del robot y los intervalos de distancia donde se encuentran los valores mínimos y máximos de lectura, para ajustar los valores de acuerdo a los parámetros de la simulación.

El parámetro **L** al que inicialmente hemos restado importancia a este término durante la formulación y la hemos considerado un parámetro poco relevante durante ella. No obstante, uno de los mayores problemas que surgieron durante el desarrollo del prototipo en Matlab fue motivado por ella, pues es un parámetro dimensional que esta expresado en función de las unidades con las que estemos trabajando. En el artículo original de Javier Mínguez [1] se usa un valor concreto obtenido mediante calibración experimental, cuyo valor es $L = 3metros$.

Esto fue fuente de un error y evaluar L= 3, trabajando con milímetros en el simulador un valor de L tan pequeño originó un estudio intensivo del algoritmo en busca de fuente de errores, permitiéndonos conocer en mayor profundidad la influencia que la calibración este parámetro tiene sobre el algoritmo MbICP, como pudimos observar en la subsección 2.2.1.2.

Capítulo 7

Desarrollo algoritmo MbICP en CoolBOT

7.1. Introducción

Como ya hemos visto en la sección 3, CoolBOT [24] es una plataforma desarrollada para llevar a cabo una fácil implementación de aplicaciones con un alto grado de abstración de la máquina real con la que se trabaje.

Principalmente, cuando nos encontramos en sistemas miltihilo y multiproceso sobre maquinas de carácter distribuido, necesitamos un software como CoolBOT para reducir al máximo el tiempo de desarrollo de aplicaciones sobre sistemas robóticos, ya que esta herramienta proporciona un metodo sencillo con varios niveles de abstracción y control. Este proyecto en continuo desarrollo se apoya en el uso de autómatas de estado para generar una infraestrutura donde solo necesitamos conocer los estados que tendrá nuestra aplicación y con que nombre son generados e inmediatamente con un comando de la aplicación se genera el código que facilita su uso.

No obstante, la plataforma CoolBOT ni el lenguaje C++ de operaciones de cálculo matricial y vectorial capaces de llevar a cabo los computos necesarios en el algoritmo MbICP, como ocurría en la plataforma Matlab. Por ello, se ha seleccionado la librería Eigen que describiremos a continuación para seguir con el diseño y la implementación del algotimo MbICP.

Finalmente, cerraremos este capítulo con una evaluación de las pruebas realizadas y una comparativa de rendimiento y presición entre Matlab vs CoolBOT.

Despues de haber evaluado los aspectos interesantes y los datos más relevantes mediante la aplicación realizada sobre la plataforma Matlab. Se han podido constatar los datos más relevantes a mostrar en la vista de nuestro componente, asi como una previa estructura de organización de código que servió de guía para hacer un diseño más robusto y eficiente en CoolBOT. A continuación describiremos las partes esenciales que encapsulan la lógica de la aplicación y cerraremos este apartado con la descripción conceptual de la integración final que se ejecutará de forma definitiva sobre el robot real para el uso de futuros usuarios de la plataforma.

7.2. Eigen, librería matricial en C++.

7.2.1. Introducción.

Eigen [18] es una librería en C++ implementada bajo licencia GNU que permite llevar a cabo cálculos matriciales, vectoriales y cálculo numérico en general. Aporta una interfaz muy versatíl para trabajar de forma similar a Matlab aunque con la potencialidad de C++. Entre sus características podemos destacar:

- Versátil.
  - Soporta todos los tamaños de matrices, desde pequeñas matrices de transformación a grandes matrices de densidad, pasando por matrices escasas.
  - Soporta todos los tipos de datos numéricos estándar, incluyendo std::complex, integers, y permite crear facilmente tipos de datos númericos propios.
  - Soporta varias descomposiciones matriciales y operaciones geométricas.
  - Su ecosistema de módulos compatibles proporciona muchas funciones especializadas tales como la optimización no lineal, las funciones de la matriz, un solucionador de polinomio, FFT, y mucho más.
- Rapidez.
  - Expresión mediante plantillas permite eliminar variables temporales.
  - Vectorización explícita optimizada para SSE 2/3/4, ARM NEON, y establece una instrucción AltiVec, de agradecer en códigos no vectorizados.
  - Matrices de tamaño fijo están completamente optimizados: se evita la asignación de memoria dinámica , y los bucles se desenrollan cuando es posible.
  - Para matrices grandes, se presta especial atención a la caché de manejo.
- Reutilización
  - Los algoritmos son cuidadosamente seleccionados por su fiabilidad. Fiabilidad en cuanto a que están claramente documentadas y proporcionan muy seguras descomposiciones, aparte de ser gratuítos.
  - Eigen se prueba a fondo a través de su propia serie de pruebas (más de 500 archivos ejecutables), mediante el estándar privado de pruebas BLAS , y entorno de pruebas LAPACK.
- Elegancia.
  - Esta API se expresa mediante plantillas dando la sensación a los programadores de C++ de escribir en lenguaje matemático, similar a Matlab.
  - La implementación de un algoritmo usando Eigen nos da una percepción como si copiáramos pseudocódigo.
- Eigen cuenta con el soporde de un buen compilador. Corrieron el programa con muchos compiladores para garantizar la fiabilidad y evitar cualquier error generador por el compilador. Eigen tambien es un estándar de C++98 y mantiene unos muy razonables tiempos de compilación.

7.2.2. Estructuras de datos.

Entre las estructuras de datos que existen en esta amplia librería que es Eigen nosotros hemos echo uso de 2 estructuras de datos Vector3d y Matrix3d ambas estructuras orientadas al cálculo matricial en tres dimensiones y expresamente optimizadas para ello.

7.2.3. Como funciona.

A la hora de usar Eigen existen diferentes formas. La opción más cercana a nuestro proyecto parte del uso de CMake y Pkg-config. En nuestro caso basta con incluir la librería Eigen como un paquete de la siguiente forma en nuestros ficheros "CMakelist.txt" de CoolBOT:

```
#Código para la busqueda de la librería y sus flag de compilación
GET_FLAGS_PKGCONFIG("eigen3")
SET(EIGEN3_CFLAGS "${PKG_CFLAGS}")
GET_LIBS_PKGCONFIG("eigen3")
SET(EIGEN3_LIBS "${PKG_LIBS}")
#Código de inclusón de ruta en los ficheros bin a compilar
SET(FLAGS_BIN_COMPILER "${FLAGS_BIN_COMPILER} ${EIGEN3_CFLAGS}")
SET(FLAGS_BIN_LINKER   "${FLAGS_BIN_LINKER} ${EIGEN3_LIBS}")
#Código de inclusión de las librerías de eigen
SET(FLAGS_LIB_COMPILER "${FLAGS_LIB_COMPILER} ${EIGEN3_CFLAGS}")
SET(FLAGS_LIB_LINKER   "${FLAGS_LIB_LINKER} ${EIGEN3_LIBS}")
```

7.3. Clase MbICP.

La clase MbICP inpendiza el algoritmo MbICP de la plataforma CoolBOT, permitiéndo su portabilidad y reutilización para desarrollos que requieran la atenuación de errores odométricos y no estén relacionados con CoolBOT.

7.3.1. Estructura.

Por consiguiente, esta clase en C++ encapsula el algoritmo MbICP, los parámetros de configuración, el muestro sensorial en el instante i y el estado actual de la solución, por medio de la acumulación de correciones. El sistema mantiene un estado interno que evoluciona conforme se van recibiendo datos, donde los datos sensoriales del instante $i+1$ pasan a ser los del instante i y actualizando finalmente el instante $i+1$ con los nuevos datos proporcionados.

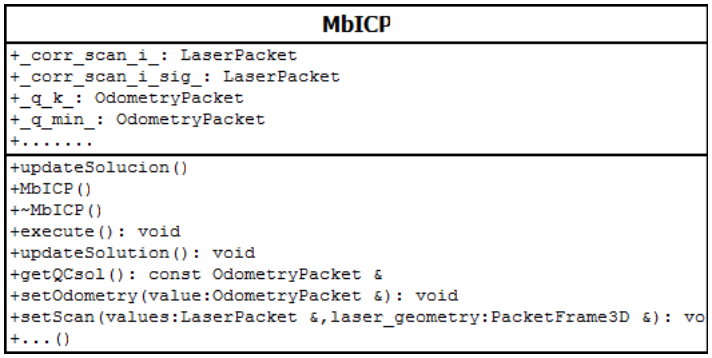


Figura 7.1: Diagrama clase MbICP

Comenzamos por describir la clase algoritmo MbICP, que podemos ver en la figura 7.1, será implementación portable en C++, de forma que en caso de querer usar el algoritmo externamente a CoolBOT solo requerirá incluir las librerias que usa la clase. Permitiendo reutilizar este sistema en otras aplicaciones o desarrollos futuros relacionados con el Scan Matching.

Esta estructurado en:

- una **función execute()** núcleo de ejecución del algoritmo
- una **función** finalizadora **updateSolution()** encargada de actualizar los cálculos obtenidos de la función anterior. En caso de no haberse ejecutado, la actualización no tendrá efecto sobre la correción acumulada, ni las variables de odometría corregida.
- Un conjunto de **funciones de configuración** basadas en la guía de estilo "Set" y "Get" más el nombre de la propiedad que se desea configurar. En base a Matlab, dichas variables son: L, Umbral, Precisión (Cota de error máximo), Número máximo de iteraciones.
- Un conjunto de **funciones de manejo de los datos** para la simulación principalmente dos "set/getOdometry" y "set/getLaser" que almacenarán la odometría y laser en base a la geometría porporcionada por el robot.
- Un conjunto de funciones de depuración y visualización de datos caracterizados únicamente por el método "get", que mostrarán valores como el valor de odometría en el instante i, $i+1$, sol, k, $error$, $incr$, el barrido del sensor laser, la geometría utilizada en cada instante, etc.

7.3.2. Datos Entrada/Salida.

Concretamente, debido a las características internas de muestreos de los sensores odométricos y de rango empleados por el algoritmo MbICP para hallar las correciones. Encontramos que los sensores odométricos generan información en periodos más cortos tiempo lo que implica que recibimos mayor número de entradas por unidad tiempo en oposición a los sensores de rango. Este comportamiento lo describiremos con mayor detalle en la subsección 7.3.3, pero antes veamos las entradas y salidas de datos de la clase MbICP.

Entradas de datos:

- **Odometría**: Captura de los sensores odométricos recibida en el instante k y que se almacena en una cola FIFO a la espera de un barrido.
- **Barrido de rango:** Contiene la lectura efectuada por el sensor de rango en el instance $i+1$ con respecto al SC del sensor.
- **Geometría del robot:** La geometría relaciona al SC del sensor de rango en el instante $i+1$ con respecto al SC del robot.

Salidas de datos:

- **Correción de odometría ($q_{change_{sol}}$):** Odometría resultante de ejecutar el algoritmo MbICP la correción entre el instante $i+1$ y el instante i.
- **Odometría corregida (q_{sol}):** Odometría final que tiene el robot en el instante $i+1$, tras aplicar la acumulación de correciones sobre $q_{change_{sol}}$.

Entradas de control:

- **Número de iteraciones máximo:** Marcará el máximo número de iteraciones que pueden ser ejecutadas en caso de que el algoritmo no converga a una solución concreta.
- **Valor del parámetro L:** Define el valor de L en la ecuación 2.23 en metros.
- **Error máximo:** Define el valor mínimo para dar como valida una solución.

Salidas de control:

- **Odometría instante i:** La odometría interpolada para el instante i .
- **Odometría instante $i+1$:** La odometría interpolada para el instante $i+1$.
- **Barrido de rango instante i:** Barrido de rango para el instante i tras haber aplicado la geometría del robot.
- **Barrido de rango instante $i+1$:** Barrido de rango para el instante $i+1$ tras haber aplicado la geometría del robot.
- **Geometría instante i:** Geometría usada para calcular el barrido de rango para el SC en el instante i.
- **Geometría instante $i+1$:** Geometría usada para calcular el barrido de rango para el SC en el instante $i+1$.
- **Valor actual parámetro L:** Devuelve el valor empleado en el algoritmo MbICP
- **Valor actual número máximo de interioces:** Devuelve la cota superior del número de iteraciones para converger a una solución.
- **Valor actual Error máximo:** Retorna el error máximo, para comprobar si se ha alcanzado la solución

- **Duración última ejecución:** Indica el tiempo que ha sido necesario para alcanzar la solución

- **Estampa tiempo comienzo ejecución:** Marca la estampa de tiempo de cuando se comenzó la ejecución del algoritmo MbICP.

Las variables de control nos permiten observar cual ha sido el comportamiento durante la ejecución. Extrayendo información útil a la hora de depurar su funcionamiento y visualizarlo.

7.3.3. Funcionamiento.

La clase tiene un comportamiento definido por una serie de fases: inicializado, preparado (esperando por datos sensoriales), ejecución algoritmo y actualización de resultados. A coninuación pasaremos a describir estas tres fases con más detalle. Para conocer mejor el comportamiento que tendrá esta componente describirémos el flujo de ejecución mediante el siguiente diagrama:

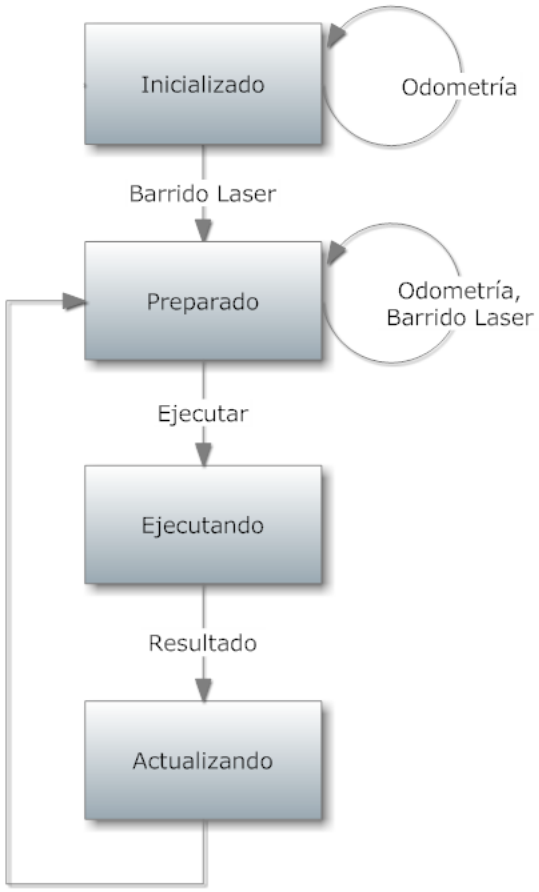


Figura 7.2: Diagrama de flujo de ejecución algoritmo MbICP

Inicializado.

Este estado de la clase MbICP responde a la necesidad de inicializar el algoritmo MbICP con la odometría y barrido del sensor de rango.

Preparado (esperando por datos sensoriales).

La recepción de datos sensoriales viene marcada por el sensor con mayor periodo de muestreo, en nuestro caso el sensor de rango, ocasionando que el sistema dispondrá de los datos necesarios únicamente cuando toda la información sensorial sea recibida (odométria y barrido de rango) gene-

rando así un nuevo instante de tiempo válido. Existiendo la posibilidad de recibir un cierto número de odométrias antes de que este disponible.

Por tanto, la odometría se almacena en una cola FIFO a la espera del sensor de rango. Una vez el barrido de rango esta disponible se genera un nuevo instante de tiempo mediante la siguiente secuencia:

1. Dado el barrido en el SC del sensor de rango se hace uso de la geometría asociada a dicho barrido para trasladar el barrido del SC del sensor al SC del robot.

2. Desplazamos los datos que describían el estado interno en el instante $i+1$ al instante i, pues los nuevos datos ahora formarán el nuevo instante a corregir por el algoritmo MbICP.

3. Interpolamos la odometría para estimar el valor más próximo a la estampa de tiempo del barrido de rango.

4. Almacenamos la odometría y el barrido de rango como nuevo instante $i+1$.

En consecuencia, debido a las características del algoritmo MbICP el instante i marcará la última ejecución para la que se encontró solución. Esto se representa muy claramente en la figura 7.3 viéndose la evolución en el tiempo del robot y como su posición se estima en función del instante en el que se ha recibido la información del sensor de rango.

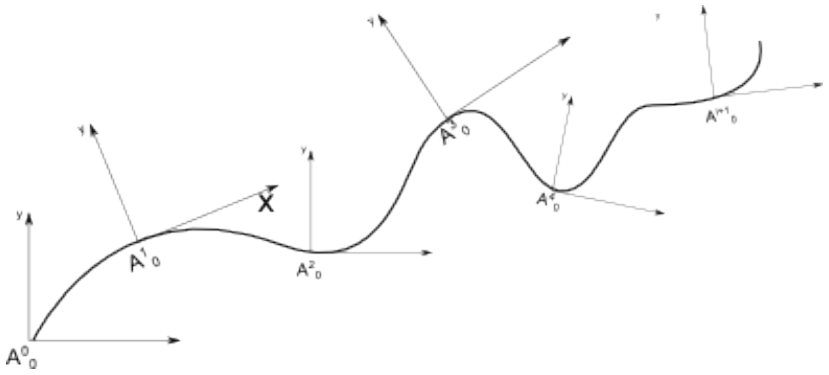


Figura 7.3: Evolución de la odometría en el tiempo

Ejecución algoritmo.

Una vez la información esta disponible y se puede ejecutar el algoritmo MbICP dando lugar a la consecuente solución entre el SC del robot en el instante $i+1$ respecto al SC del robot en el instante i.

Como ya hemos visto en la sección 1.3, dicha secuencia de ejecución se divide en 4 pasos cada uno de ellos con una función específica dentro de la clase teniendo como resultado los siguientes elementos:

1. **_calc_closest_points_(**$scan_i$, $scan_{i+1}$, A_i^{i+1}**):** encargada de calcular el conjunto de parejas más próximos entre sí, usando la distancia MbICP (**_distance_measure_(**p_1, p_2**)**) sobre los barridos del sensor de rango en los instantes $i+1$ e i.

2. **_leasure_cuadratic_**($corr_{scan_i}$, $corr_{scan_{i+1}}$): Dadas las correlaciones de puntos entre los barrido del instante $i+1$ e i hallamos el SC que hace mínima la transformación.

3. **_bigAddOp_**(q_{min}, q_k)**:** Calculamos la solución q_{csol} por medio de la transformación de coordenadas entre el SC q_{min} respecto al SC q_k.

4. Repetimos los pasos 1, 2 y 3 hasta que $q_{min} < q_{max_{error}}$, actualizando $q_{k+1} = q_{csol}$.

No obstante, al finalizar ejecución se almacena el valor de correción calculado en su estado interno a la espera que la señal de control solicite una actualización de los valores.

Actualización de resultados.

Una vez se ha asegurado que ningún hilo esta accediendo a variables sensibles de modificarse en la actualización de la solución se procede a ejecutar la actualización y generar las variables de salidas $q_{change_{sol}}$ y q_{sol}.

7.4. Componente MbICPCorrector.

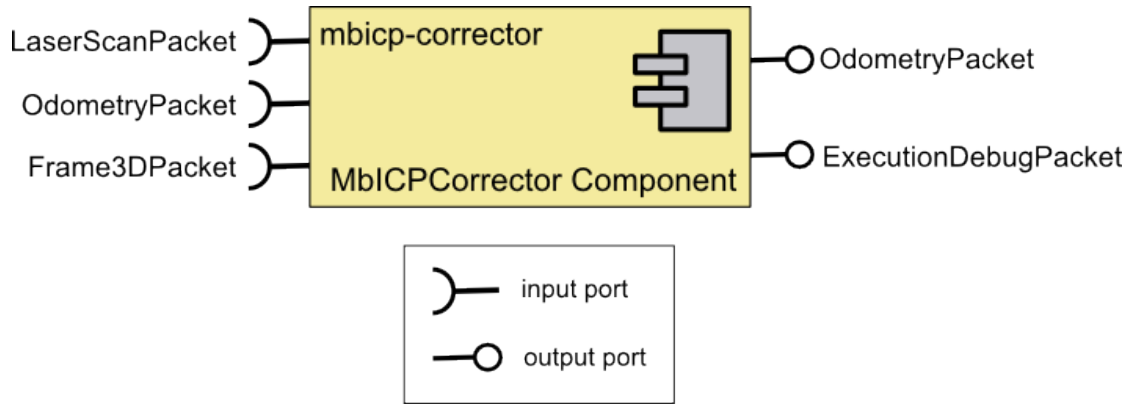


Figura 7.4: Componente MbICPCorrector

El componente MBICPCorrector creado haciendo uso de la plataforma CoolBOT posee un entramado lógico que implementará un sistema múltihilo con tres puertos de entrada y dos puertos de salida, como vemos en la figura 7.4, compuesto de dos estados para el control del componente durante la ejecución que son los siguientes:

- Waiting: Es el estado encargado de inicializar el sistema cuando se conecta un componente, que suministra datos de entrada al sistema. Inicializadas las variables "LASER_GEOMETRY" y "ODOMETRY" necesarias para procesar adecuadamente los barridos del sensor de rango.

- Main: Es el estado principal de ejecución encargado de gestionar la información y controlar todo el componente, dividido en:

- Dos hilos de ejecución, que acceden a variables compartidas y que transmiten comandos de control informando de la disponibilidad de nuevos datos en las mismas.
- Almacenar y procesar los datos recibidos por los puertos de entrada, "ODOMETRY", "LASER_SCAN" y "LASER_GEOMETRY".
- Extraer y enviar la información de visualización mediante el paquete "DebugExecutionPacket" que describirémos en la sección 7.5.
- Enviar o estimar la odometría corregida cada vez que se reciba una odometría de entrada, imprescindible para que el planificador y el evitador de obstaculos puedan tomar decisiones en tiempo real.

Por consiguiente, la figura 7.5 la estructura de estados e hilos de ejecución expuesta.

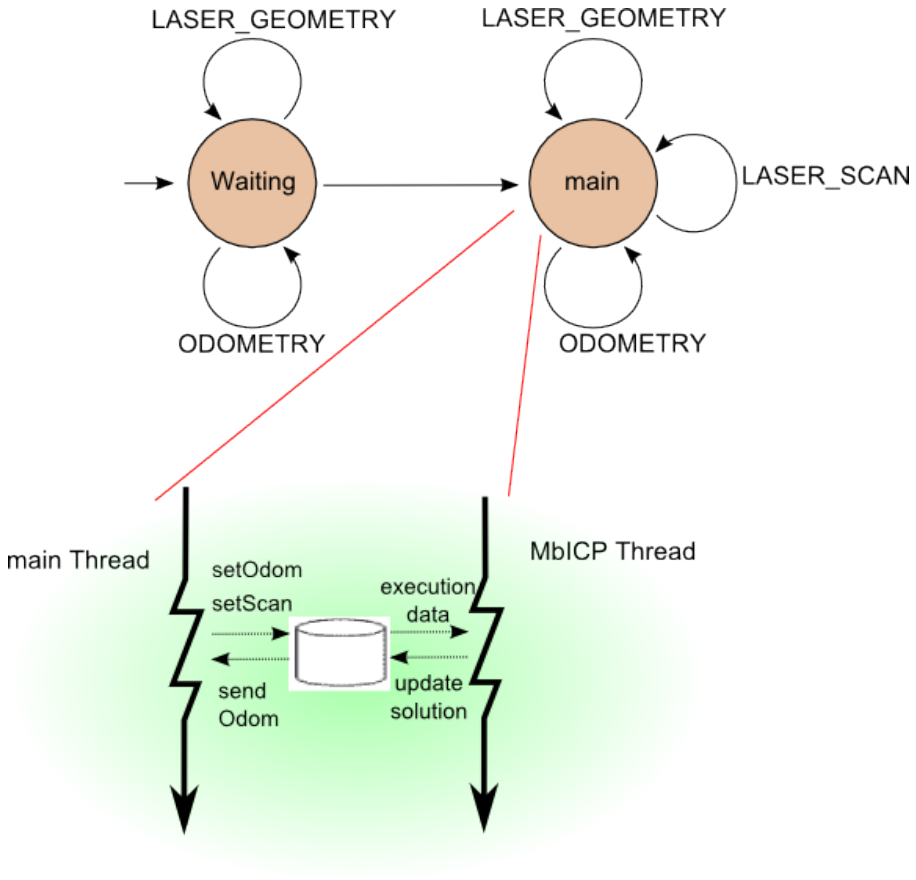


Figura 7.5: Diagrama de estados del componente MbICPCorrector

Una vez descrita la estructura básica del componente comenzarémos por describir que estos dos hilos de ejecución tienen un cometido específico imprenscindible por tratarse de un sistema en tiempo real, donde un retraso en la recepción de la odometría corregida en el evitador de obstáculos

podría ocasionar una colisión o en menor medida un error de planificación debido a no existir información precisa de la posición en el instante actual del robot.

El primer hilo "mbicpThread" se encuentra exclusivamente destinado al cálculo de correciones del algoritmo MbICP (implementado en la clase MbICP descrita en la sección 7.3) cuya ejecución se realiza asíncronamente con el sistema del componentes. En suma, este hilo tiene asociada la entrada "LASER_SCAN" donde la presencia de nuevos datos genera una actualización de la solución cálculada en el instante anterior mediante la ejecución nuevamente del algoritmo MbICP.

El segundo hilo "main" se encuentra destinado al control y gestión del flujo de datos del componente "MbICPCorrector". Las tareas principales de este hilo son:

- Almacenar la geometría (LASER_GEOMETRY) y odometría (ODOMETRY) del nuevo instante de tiempo.

- Generar el paquete "DebugExecutionPacket" con la información obtenida tras la ejecución del algoritmo MbICP en el instante $i+1$ para la visualización en la vista MbICP-gtk (descrita en la sección 7.6) mediante el puerto de salida "EXECUTION_INFO".

- Estimar de correción de odometría en base a la solución acumulada obtenida hasta el momento (como ilustra la figura 7.6). Enviando dicha estimación inmediatamente por el puerto "ODOMETRY_CORRECTED" calculado cada vez que se reciba un nuevo paquete en la entrada "ODOMETRY".

- Actualizar la solución (q_{sol}) en la correción acumulada en cuanto este disponible para mejorar la estimación para instantes de tiempo posteriores.

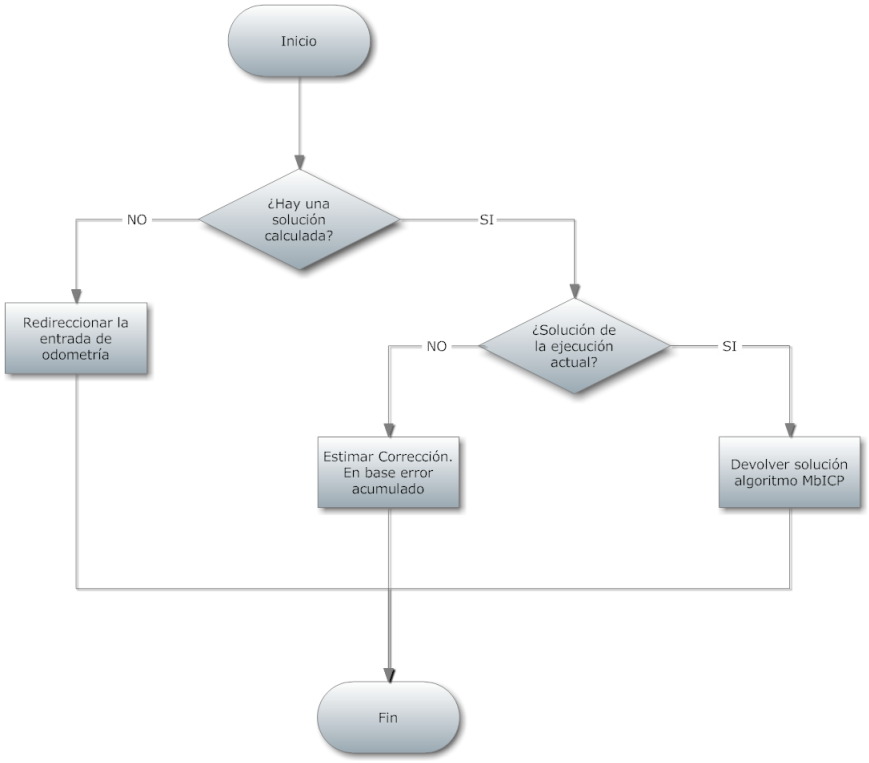


Figura 7.6: Diagrama de flujo del componente "MbICPCorrector"

Aunque muchos sistemas disponen de interpolación, **este sistema aporta una característica adiccional y es que interpolará aquellas odometrías que han ido recibiéndose previas al barrido del sensor de rango**. El motivo es simple el robot suele estarse desplazando por el mapa y los sensores odométricos por lo general tienen un periodo más corto de envío de información que los sensores laser.

Por todo esto, el estado interno del autómata que compone el componente MbICPCorrector viene definido por la figura 7.5. formando la siguiente secuencia de ejecución ilustrada por la figura 7.7que descompone el proceso que ha de seguir un componente CoolBOT, desde su inicialización al comienzo de ejecución hasta su finalización, tolerando errores de cierta complejidad y recuperándose siempre que sea posible para mantener su correcto funcionamiento.

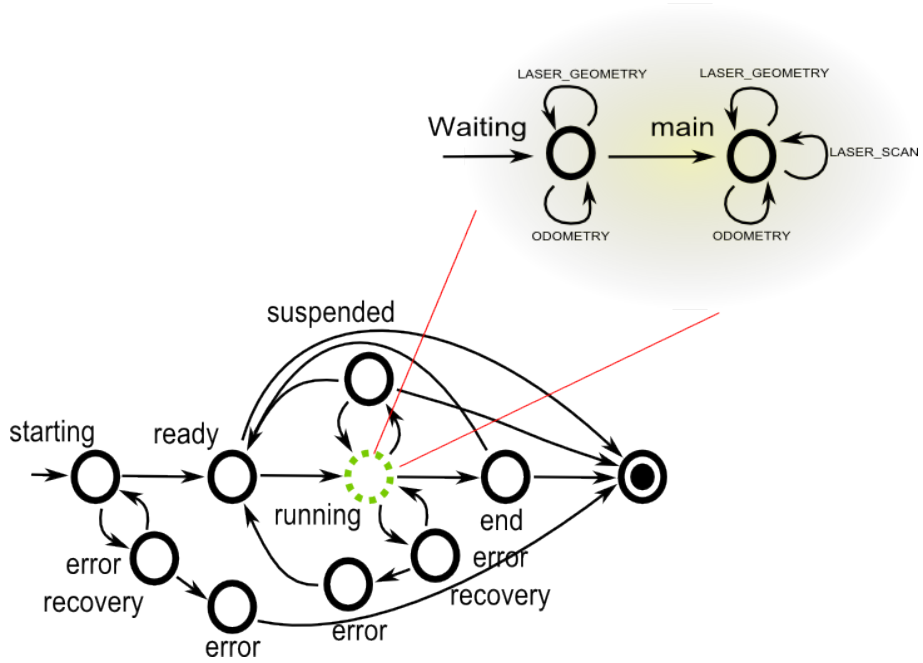


Figura 7.7: Diagrama de estados componente MbICPCorrector

Por otro lado, en caso de estarse ejecutando el algoritmo MbICP se permitirá almacenar más datos de entrada para preparar nuevas ejecuciones, donde cada nueva pareja $\{laser, geomería\}$ llevará al componente a un estado preparado para permitir nuevas ejecuciónes como vemos en la figura 7.5. Al finalizar su ejecución se pasa a un estado actualizable previo a una nueva ejecución que se encargará de actualizar las variables de solución actual y correción acumulada.

Sin embargo, la existencia de dos hilos dentro del estado "main" genera un estado de concurrencia donde se puede dar casos como encontrarse estimando una solución y cuando se desea actualizar la solución acumulada empleada para dicha estimación, existiendo la posibilidad de causar una inconsistencia en los dato compartidos. Por tanto, hay que destacar varias funcione del código que requieren exclusión mutua, que son:

- **"SetOdometer"**: que almacena en una cola FIFO la posición.
- **"SetLaser"**: que almacena el barrido laser aplicando la transformación de geometría, logrando que ya este preparado para su uso en el algoritmo sin cálculos adiccionales que lo ralenticen, y la actualización de la iteración i e $i+1$ donde el primer barrido será el i, el segundo el $i+1$ y el tercero y sucesivos serán el $i+1$ pasando el $i+1$ a ser el i.
- **"UpdateSolution"**: Encargado de actualizar la variables de solución resultantes.

Consecuentemente, la última labor de este componente aparte de la gestión de algoritmo es garantizar la consistencia de las variables compartidas por medio de variables cerrojo.

Para concluir, el componente MbICPCorrector tiene incorporada una utilidad llamada "Chronofiles" que nos permite disponer de un conjunto de archivos para la visualización del tiempo de computo (cuantos de tiempo requeridos) de los distintos componentes para mostrar un cronograma de tiempos en Matlab, nuestro componente no es una execepción y por tanto, la vista de conjunto del sistema presenta una vista similar a la figura inferior.

Código CoolBOT que representa al componente MbICPCorrector

```
component MbICPCorrector {

    header{
       author "Eduardo Aparicio Cardenes <eduardo.aparicio.cardenes@gmail.com>";
       description "MbICP Corrector component";
       institution "ULPGC - Universidad de Las Palmas de Gran Canaria";
       version "0.1"
    };
    constants{
       LASER_MAX_RANGE="LaserPacket::LASER_MAX_RANGE";
    };
    // input ports
    input port ODOMETRY type last port packet PlayerRobot::OdometryPacket;
    input port LASER_GEOMETRY type poster port packet PacketFrame3D;
    input port LASER_SCAN type last port packet PlayerRobot::LaserPacket;
    //output ports
    output port ODOMETRY_CORRECTED type generic port packet OdometryPacket;
    output port EXECUTION_INFO    type generic port packet DebugExecutionPacket;
    entry state Waiting
    {
       transition on LASER_GEOMETRY,ODOMETRY;
    };
    state Main
    {
       transition on ODOMETRY,LASER_GEOMETRY,LASER_SCAN;
    };
    thread main
    {
       input box { ODOMETRY, LASER_GEOMETRY };
    };
    thread mbicpThread
    {
       input box { LASER_SCAN };
       active in Main;
    };

};
```

7.5. Paquetes de datos.

Este espacio de nombres contiene el paquete "DebugExecutionPacket" empleado para transportar la información extraída del componente MbICPCorrector tras cada iteración y enviarla a la vista MbICP-gtk para que mediante su visualización posible sea posible analizar el comportamiento del algoritmo MbICP. El "DebugExecutionPacket" almacena en su interior los siguientes datos:

- Odometría, barrido Laser y geometría en el instante i.
- Odometría, barrido Laser y Geometría en el instante $i+1$.
- Correción entre el instante $i+1$ e i, que en la formulación recibe el nombre de ${}^{corr}\triangle_i^{i+1}$
- Tiempo de ejecución del algoritmo (T_{inicio},$T_{ejecución}$) y número de iteraciones requeridas.
- TimeStamp del paquete.

Esencialmente este paquete incorpora todos los recursos para la comuncicación por la red abstrayendo al programador de la capas de transmisión de datos. Únicamente debe conocer que todo componente y vista que use este paquete como entrada o salida en su construcción.

La forma de usarlo para enviar información es a través de la función "send" y como norma general podemos extrapolarlo a cualquier paquete usado en nuestra aplicación.

```
nombrePaquete *packet;
packet->setNombreCampo1(contenido1);
...
packet->setNombreCampoN(contenidoN);
pObox->send(packet);
```

CoolBOT define por cada entrada asociada a cada estado una función que es llamada cuando aparece un nuevo paquete en el puerto. Esta función tiene como nombre "estadoEntrada()" siendo estado el nombre del estado asociado a cada transición de Entrada. Para recibir un paquete es tan sencillo como hacer

```
tipoPaquete *packet = pIbox->receive(Entrada);
```

Dentro de cada ejecución podemos trabajar con el contenido de nuestro paquete que en nuestro caso nos permite comunicar el component "MbICPCorrector" con la vista "Mbicp-gtk" y mostrar el comportamiento del algoritmo MbICP durante la ejecución del robot a tiempo real.

Código CoolBOT que representa el conjunto de paquetes "MbICP"

```
packets MbICP
{
    /*
     * Packet's definition.
     */
header
{
author "Eduardo Aparicio Cardenes <eduardo.aparicio.cardenes@gmail.com>";
```

```
description "MbICP port packets";
institution "ULPGC - Universidad de Las Palmas de Gran Canaria - Spain";
version "1.0"
};
port packet DebugExecutionPacket
{
    data members
    {
scan_i:        type "LaserPacket";
scan_i_plus_1:        type "LaserPacket";
odom_i:               type "OdometryPacket";
odom_i_plus_1:        type "OdometryPacket";
odom_change_sol:      type "OdometryPacket";
geom_i:               type "Frame3D";
geom_i_plus_1:        type "Frame3D";
time_start_execution: type "Time";
duration_execution:   type "Time";
iteration_needed_ex:  type "int";
    };
};
};
```

7.6. Vista "Mbicp-gtk".

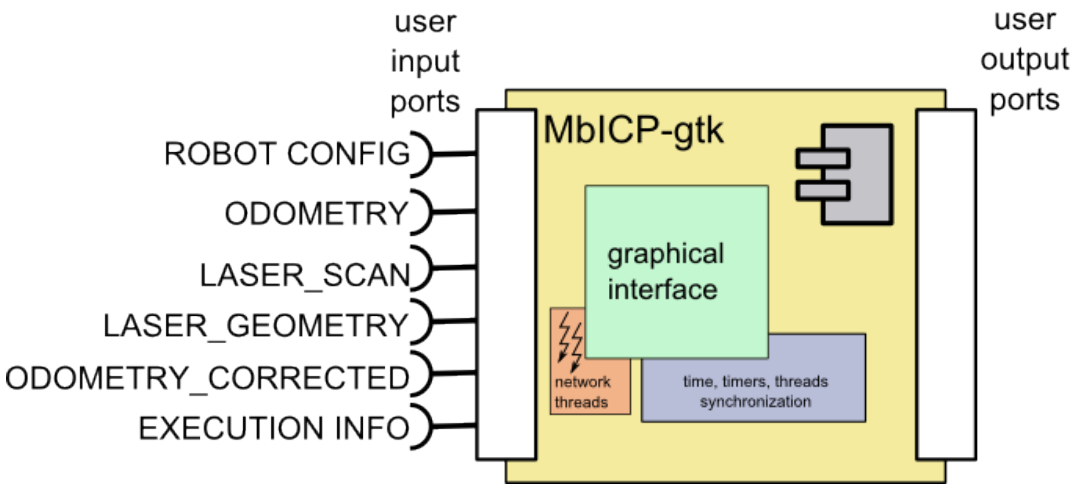
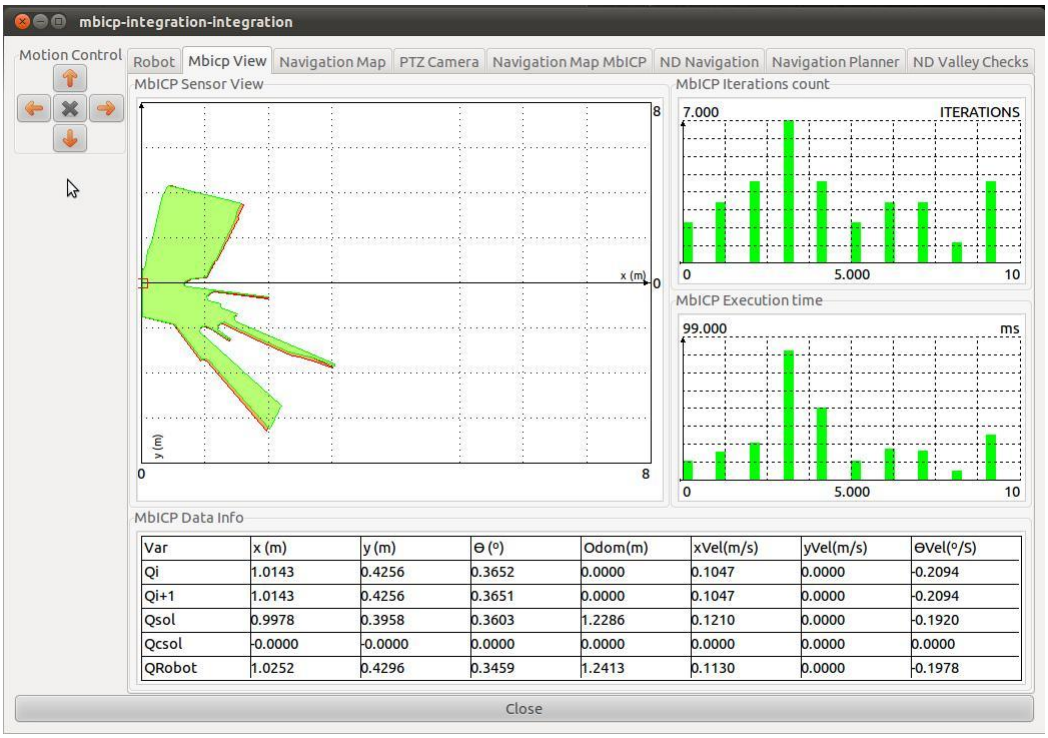


Figura 7.8: Vista MbICP-gtk

Esta vista implementada mediante la librería gtk [23], contendrá parte del diseño visto para el prototipo en Matlab en la sección 6.3 pero abstrayéndola de información de depuración, ya que esta

vista esta orientada a un entorno de ejecución en tiempo real y cualquier información de depuración adiccional penalizaría el tiempo de ejecución del componente. Notése que esta vista no dispone de puertos de salida porque genera información de control para ningún componente. Este diseño puede verse ilustrado en la figura 7.9.



Var	x (m)	y (m)	Θ (°)	Odom(m)	xVel(m/s)	yVel(m/s)	ΘVel(°/S)
Qi	1.0143	0.4256	0.3652	0.0000	0.1047	0.0000	-0.2094
Qi+1	1.0143	0.4256	0.3651	0.0000	0.1047	0.0000	-0.2094
Qsol	0.9978	0.3958	0.3603	1.2286	0.1210	0.0000	-0.1920
Qcsol	-0.0000	-0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
QRobot	1.0252	0.4296	0.3459	1.2413	0.1130	0.0000	-0.1978

Figura 7.9: Vista "MbICP-gtk"

Por consiguiente, la vista dispondrá de tres ventanas y una pequeña tabla de variables en la parte inferior de la pestaña. La organización de estos elementos es la siguiente:

- **La ventana izquierda:** mostrará los barridos del sensor de rango superpuestos. Además, para mantener la coherencia de colores con el prototipo en Matlab, el barrido verde se corresponde al instante $i+1$ y el rojo al instante i
- **La ventana superior derecha**: se mostrará un historico de ejecución de las 10 últimas iteraciones.
- **La ventana inferior derecha:** este histórico almacenará el tiempo de ejecución en milisegundos de las 10 últimas iteraciones.
- **la tabla de variables:** contendrá los datos de odometría del comportamiento del robot expresado de la siguiente forma:

- Q_i: la odometría para el instante i .
- Q_{i+1}: la odometría en $i+1$.
- Q_{csol} : visualizará la solución hallada entre los instante i e $i+1$
- Q_{sol}: será la solución acumulada Q_{sol} hasta el instante i más la nueva solución Q_{csol}.
- Q_{robot}: representa la odometría que recibiría el panificador y el evitador de obstaculos en caso de no aplicar el algoritmo MbICP.

Esta vista en Gtk [23] esta compuesta principalmente por medio del grupo de eventos **"expose()"** y **"configure()"** encargados de repintar o escalar la ventana respectivamente. Cada una de las ventanas e incluso la tabla tiene asociados sus propios eventos **"expose()"** y **"configure()"** encargados de gestionar el area de dibujo que las contiene, para ello se dispone de un Frame y un PixMap asociados a ellas que se almacenan el actual y el próximo Frame a visualizar en pantalla.

Código CoolBOT que representa la vista "MbICP-gtk".

```
view mbicp_gtk
{
    /*
    * View's definition.
    * Rules:
    * - You must have at least the definition of a port of entry or exit.
    *
    */
    header
    {

        author "Eduardo Aparicio Cardenes <eduardo.aparicio.cardenes@gmail.com>";
        description "MbICP Corrector component";
        institution "ULPGC - Universidad de Las Palmas de Gran Canaria";
        version "0.1"
    };
    constants
    {
        private DEFAULT_REFRESHING_PERIOD =250; // milliseconds
        private DRAW_MAP_VIEW_AREA_MARGIN = 3; // pixels
        private DRAW_MAP_VIEW_AREA_WIDTH =350; // pixels
        private DRAW_MAP_VIEW_AREA_HEIGTH =400; // pixels
        private DRAW_MAP_VIEW_AREA_XY_DIVS = 8; // should be an even number
        private DRAW_MBICP_ITERATIONS_AREA_MARGIN = 3; // pixels
        private DRAW_MBICP_ITERATIONS_AREA_WIDTH =170; // pixels
        private DRAW_MBICP_ITERATIONS_AREA_HEIGTH =100; // pixels
        private DRAW_MBICP_ITERATIONS_AREA_X_DIVS = 8; // number division
        private DRAW_MBICP_ITERATIONS_AREA_Y_DIVS = 8; // number division
        private DRAW_MBICP_TIME_AREA_MARGIN = 3; // pixels
```

```
        private DRAW_MBICP_TIME_AREA_WIDTH =170; // pixels
        private DRAW_MBICP_TIME_AREA_HEIGTH =100; // pixels
        private DRAW_MBICP_TIME_AREA_X_DIVS = 8; // number division
        private DRAW_MBICP_TIME_AREA_Y_DIVS = 8; // number division
        private DRAW_MBICP_TABLE_AREA_MARGIN = 3; // pixels
        private DRAW_MBICP_TABLE_AREA_WIDTH =520; // pixels
        private DRAW_MBICP_TABLE_AREA_HEIGTH =150; // pixels
        private DRAW_MBICP_TABLE_AREA_X_DIVS = 8; // number division
        private DRAW_MBICP_TABLE_AREA_Y_DIVS = 6; // number division
        private SENSOR_MAX_RANGE = "LaserPacket::LASER_MAX_RANGE/1000"; // meters
        private MBICP_AXES_FONT_SIZE = 8; // font points
        private MBICP_ARROW_WIDTH = 6; // pixels (should be an even number)
        private MBICP_ARROW_DEEP = 6; // pixels

    };
    input port ROBOT_CONFIG type poster port packet PlayerRobot::ConfigPacket;
    input port ODOMETRY type last port packet PlayerRobot::OdometryPacket;
    input port LASER_SCAN type last port packet PlayerRobot::LaserPacket;
    input port LASER_GEOMETRY type poster port packet PacketFrame3D;
    input port ODOMETRY_CORRECTED type last port packet PlayerRobot::OdometryPacket;
    input port EXECUTION_INFO type last port packet DebugExecutionPacket;

};
```

7.7. Integración MbICP.

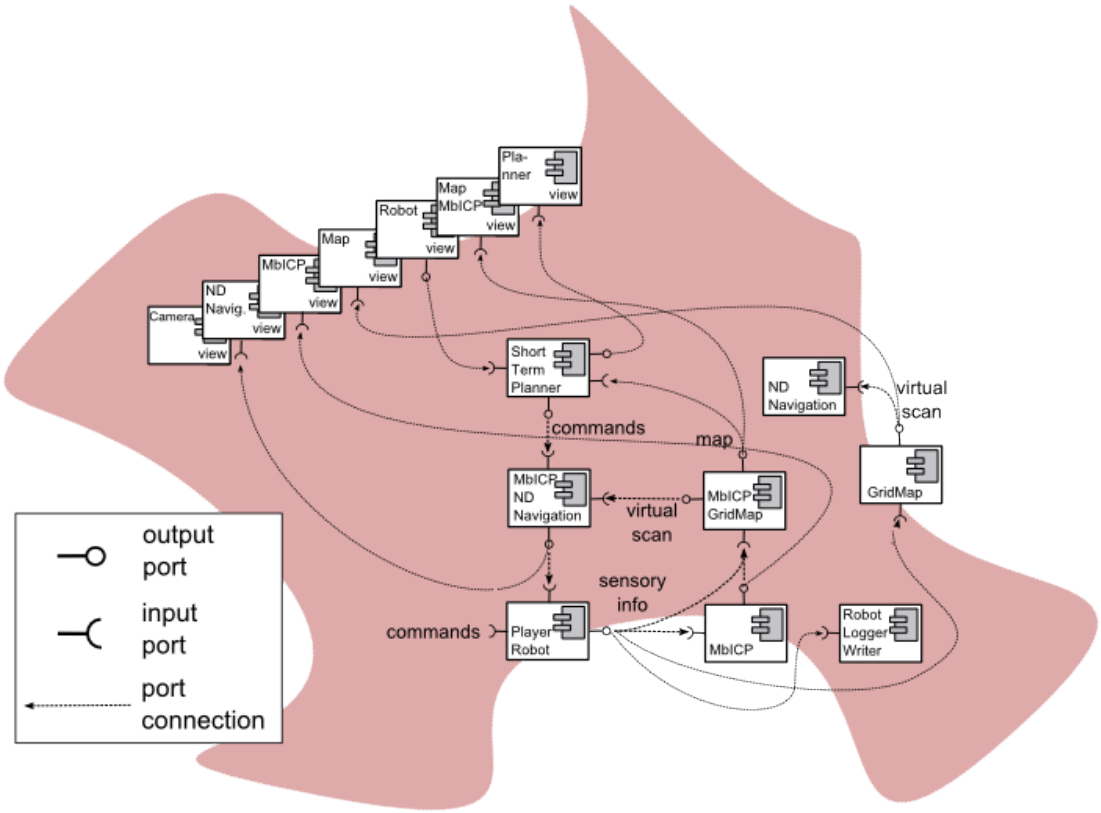


Figura 7.10: Esquema simplificado de la integración "MbICPIntegracition"

La integración ilustrada por la figura 7.10, cuya conexionado ha sido simplificado para mejorar la visualización de la figura, será la encargada de conectar la vista con los componentes de acuerdo al sistema deseado para hacer funcionar adecuadamente el robot. En este caso concreto existen dos versiones de esta integración la primera nos permite llevar a cabo el testeo en el entorno de pruebas del robot situando las vistas y componentes sobre el mismo computador. Y la segunda versión donde los componentes se encuentran ejecutándose remotamente dentro del robot y las vistas se encuentran en un computador distinto para la visualización.

En primer lugar hay que tener presente que durante la creación de este módulo de la aplicación que define la forma en la que están asociados los componentes con las vistas mediante una serie de declaraciones que se asemejan a las conexiones físicas de componentes electrónicos. Generado el fichero cpp mediante el comando "coolbot-c" se llevo a cabo los siguientes pasos para completar la integración:

1. Añadir las librerías de cada componente en el fichero "CMakeList.txt"

2. Incluir los ficheros cabecera conforme sean necesarios.

3. Cada componente y vista generará un constructor que tendrá los parámetros que hayamos definido, por ejemplo periodo de refresco.

4. Añadir una porción de código del que actualmente carece la integración y es la siguiente:

```
argc--;
argv++;
if(!argc)
{
   cout<<"usage: "<<pName <<" localhost | <tcp-ip-address>"<<endl;
   return 0;
}
cout<<"Robot connection: "<<argv[0]<<endl;
```

Con todo esto se está preparado para compilar y ejecutar nuestra integración. Sin embargo, antes de ello es necesario tener presente que debe haber en ejecución una aplicación que suministre los datos a la integración.

Para ello existe la aplicaciones Player/Stage [25]. Este es un proyecto de software libre consta de dos aplicaciones:

- Player implementación de un servidor para robots.
- Stage simulador con datos del mundo que usará Player.

Finalmente, para ejecutar el servidor que proveerá los datos a nuestra integración en el entorno de simulación, se usará el comando siguiente:

```
player “ruta-fichero-mapa-entorno.cfg”
```

En nuestro caso, el fichero a utilizar es “everything.cfg” que se encuentra dentro del servidor stage, en su interior encontramos una carpeta que recibe el nombre de “worlds/”, que almacena todos los ficheros del simulador stage, si utilizamos el simulador Stage. En caso, de que utilicemos el robot real habría que proporcionar el fichero de configuración de los sensores del robot.

Código CoolBOT que representa la integracion “MbICPIntegration”.

```
integration mbicp_integration
{

    header
    {

        author "Eduardo Aparicio Cardenes <eduardo.aparicio.cardenes@gmail.com>";
        description "MbICP integration - integrates only mbicp components";
        institution "ULPGC - Universidad de Las Palmas de Gran Canaria";
        version "0.1"
```

```
};
machine addresses
{
    /*
    * Machine addresses definition.
    */
    local mycomputer: "127.0.0.1";
};
local instances
{
    /*
    * Local instances definition.
    */
    //Components to use it
    component robot : PlayerRobot;
    component mbicpInstance: MbICPCorrector;
    component loggerWriter : RobotLoggerWriter;
    component navigationMap: GridMap;
    component navigationMapMbicp: GridMap;
    component nd: NDNavigation;
    component navigationPlanner: ShortTermPlanner;
    //Views to use it
    view robotView : PlayerRobotGtk
         with description "Robot";
    view mbicpRelaseView: mbicp_gtk
         with description "Mbicp View";
    view navigationMapView: GridGtk
         with description "Navigation Map";
    view cameraView: SphereGtk
         with description "PTZ Camera";
    view navigationMapView2: GridGtk
         with description "Navigation Map MbICP";
    view ndView: NDNavigationGtk
         with description "ND Navigation";
    view navigationPlannerView: PlannerGtk
         with description "Navigation Planner";
    view valleyCheckView: RasterGtk
         with description "ND Valley Checks";
};
port connections
{
    connect robot:ODOMETRY to robotView:ODOMETRY;
    connect robot:LASERGEOMETRY to robotView:LASER_GEOMETRY;
    connect robot:LASERSCAN to robotView:LASER_SCAN;
    connect robot:POWER to robotView:POWER;
```

```
connect robot:ODOMETRY to mbicpInstance:ODOMETRY;
connect robot:LASERGEOMETRY to mbicpInstance:LASER_GEOMETRY;
connect robot:LASERSCAN to mbicpInstance:LASER_SCAN;
connect robotView:COMMANDS to robot:COMMANDS;
connect robot:ODOMETRY to mbicpRelaseView:ODOMETRY;
connect robot:LASERGEOMETRY to mbicpRelaseView:LASER_GEOMETRY;
connect robot:LASERSCAN to mbicpRelaseView:LASER_SCAN;
connect mbicpInstance:ODOMETRY_CORRECTED
            to mbicpRelaseView:ODOMETRY_CORRECTED;
connect robot:ROBOTCONFIG to mbicpRelaseView:ROBOT_CONFIG;
connect mbicpInstance:EXECUTION_INFO
            to mbicpRelaseView:EXECUTION_INFO;
connect robot:ODOMETRY to loggerWriter:ODOMETRY;
connect robot:LASERSCAN to loggerWriter:LASER_SCAN;
connect robot:LASERGEOMETRY to loggerWriter:LASER_GEOMETRY;
connect mbicpInstance:ODOMETRY_CORRECTED
            to loggerWriter:ODOMETRY_RESULT;
connect robot:ROBOTCONFIG to nd:ROBOTCONFIG;
connect robot:ROBOTCONFIG to navigationMap:ROBOTCONFIG;
connect robot:ODOMETRY to navigationMap:ODOMETRY;
connect robot:LASERGEOMETRY to navigationMap:LASERGEOMETRY;
connect robot:LASERSCAN to navigationMap:LASERSCAN;
connect robot:ROBOTCONFIG to navigationPlanner:ROBOTCONFIG;
connect mbicpInstance:ODOMETRY_CORRECTED
            to navigationPlanner:ODOMETRY;
connect robot:SONARGEOMETRY to robotView:SONAR_GEOMETRY;
connect robot:SONARSCAN to robotView:SONAR_SCAN;
connect navigationMapMbicp:GRIDCONFIG to nd:GRIDCONFIG;
connect navigationMapMbicp:MAP to nd:MAP;
connect navigationMapMbicp:GRIDCONFIG
            to navigationPlanner:GRIDCONFIG;
connect navigationMapMbicp:MAP to navigationPlanner:GRIDMAP;
connect nd:ROBOTCOMMANDS to robot:COMMANDS;
connect navigationPlanner:NAVIGATIONCOMMANDS to nd:COMMANDS;
connect robot:ROBOTCONFIG to navigationMapView:ROBOT_CONFIG;
connect navigationMap:MAP to navigationMapView:GRID_MAP;
connect navigationPlanner:PLANNERPATH
            to navigationMapView2:PLANNER_PATH;
connect navigationPlanner:MATCHINGREGIONS
            to navigationMapView2:MATCHING_REGIONS;
connect navigationMapView2:PLANNER_COMMANDS
            to navigationPlanner:COMMANDS;
connect navigationMapView2:ND_COMMANDS to nd:COMMANDS;
connect robot:CAMERAIMAGE to cameraView:CAMERA_IMAGE;
connect robot:PTZJOINTS to cameraView:PTZ_JOINTS;
connect cameraView:COMMANDS to robot:COMMANDS;
```

```
            connect robot:ROBOTCONFIG to ndView:ROBOT_CONFIG;
            connect nd:GTKDATA to ndView:GTK_DATA;
            connect navigationMapMbicp:GRIDCONFIG
                        to navigationPlannerView:GRID_CONFIG;
            connect navigationPlanner:PLANNERMAP
                        to navigationPlannerView:PLANNER_MAP;
            connect nd:GTKDATA to valleyCheckView:GTK_DATA;
            connect robot:ROBOTCONFIG to navigationMapMbicp:ROBOTCONFIG;
            connect mbicpInstance:ODOMETRY_CORRECTED
                        to navigationMapMbicp:ODOMETRY;
            connect robot:LASERGEOMETRY to navigationMapMbicp:LASERGEOMETRY;
            connect robot:LASERSCAN to navigationMapMbicp:LASERSCAN;
            connect robot:ROBOTCONFIG to navigationMapView2:ROBOT_CONFIG;
            connect navigationMapMbicp:MAP to navigationMapView2:GRID_MAP;
        };
    };
```

7.8. RobotLogger

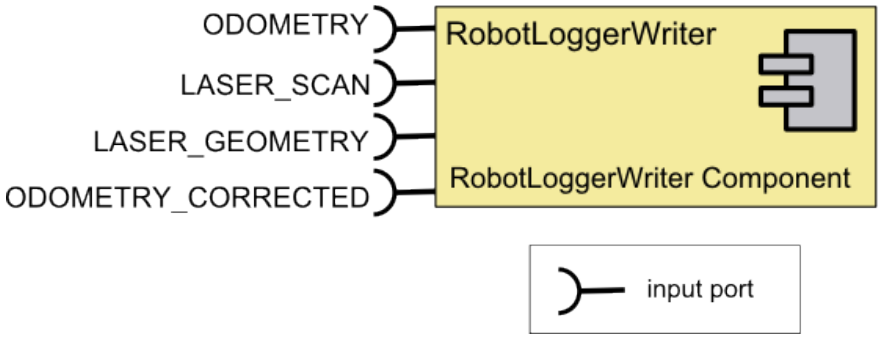


Figura 7.11: Componente RobotLoggerWriter

Hasta el momento hemos visto la existencia de dos aplicaciones una Matlab y otra CoolBOT pero actualmente no disponen de una relación directa entre ambas. Para aprovechar la versatilidad en simulación que aporta Matlab se ha realizado un nuevo componente "RobotLoggerWritter", ilustrado en la figura 7.11, orientado a extraer los datos de ejecución del robot hacia un fichero en un formato que puede leer Matlab permitiendo simulaciones de los datos para comprobar aspectos como las correspondencias realizadas por el algoritmo, el mapa que se genera en base a su correciones, etc.

Toma todos los datos de entrada: odometría, barrido laser, geometría y su respectiva solución. Volcandolos a un fichero llamado "RobotLogger.log". El formato de salida es el siguiente:

- odometry: timestamp (formato Matlab): X, Y, θ, V_x, V_y, ω
- laser-scan: timestamp (formato Matlab): Número elementos(n): x_1, y_1, x_2, $y_2, \cdots x_n$, y_n

- geometry: X, Y, Z, α, β, γ
- odometry-result: timestamp (formato Matlab): X, Y, θ, V_x, V_y, ω

Por otro lado, hay simulaciones bien testeadas en Matlab o ejecuciones anteriores del Robot en CoolBOT que queremos ver funcionando en el entorno final, para ello se ha creado el "RobotLoggerReader", ilustrado por la figura 7.12, que aporta a traves de una lectura de un fichero y un cuanto de tiempo definido, datos que alimentan a los distintos módulos del Sistema CoolBOT. El formato empleado en esta ocasión es el mismo que vemos en el "RobotLoggerWriter" para facilitar la compatibilidad entre el fichero de salida de uno y el fichero de entrada del "RobotLoggerReader".

Por tanto, este componente tiene como objetivo realimentar al sistema con datos tomados de un fichero y efectuar una ejecución idéntica a la que se vería sometido en caso de hacer uso del servidor Stage [25] donde la entrada "COMMANDS" indica el instante en el que suministran los datos al sistema.

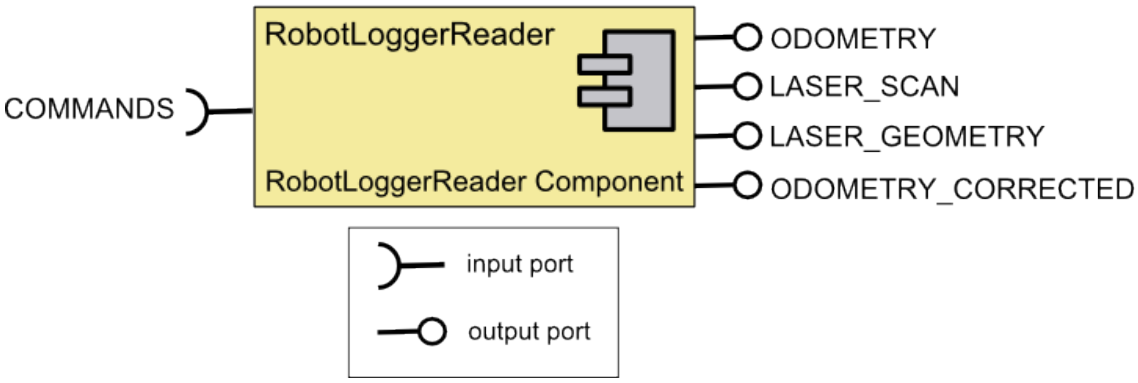


Figura 7.12: Componente RobotLoggerReader

Código CoolBOT que representa la integracion "RobotLoggerWriter".

```
component RobotLoggerWriter
{
    /*
    * State's definition.
    */
    header
    {
        author "Eduardo Aparicio Cardenes
                <eduardo.aparicio.cardenes@gmail.com>";
        description "Robot Logger Writer component";
        institution "ULPGC - Universidad de Las Palmas de Gran Canaria";
        version "0.1"
    };
    constants
```

```
        {
            // in odometry periods (about 100-200 millisecond each)
            private CIRCULAR_FIFO_LENGTH=100;
        };
        // input port COMMANDS type last port packet CommandConfigPacket;
        input port ODOMETRY type last port packet OdometryPacket;
        input port LASER_SCAN type last port packet LaserPacket;
        input port LASER_GEOMETRY type poster port packet PacketFrame3D;
        // network buffer FIFO_LENGTH;
        input port ODOMETRY_RESULT type last port packet OdometryPacket;
        entry state Main
        {
            //State's body
            transition on ODOMETRY,ODOMETRY_RESULT,LASER_SCAN,LASER_GEOMETRY;
        }
    };
```

Código CoolBOT que representa la integracion "RobotLoggerReader".

```
    component RobotLoggerReader
    {
        /*
        * State's definition.
        */
        header
        {
            author "Eduardo Aparicio Cardenes
                   <eduardo.aparicio.cardenes@gmail.com>";
            description "Robot Logger Reader component";
            institution "ULPGC - Universidad de Las Palmas de Gran Canaria";
            version "0.1"
        };
        constants
        {
            L=10;
        };
        input port COMMANDS type last port packet CommandConfigPacket;
        // network buffer FIFO_LENGTH;
        output port ODOMETRY_RESULT type generic port packet OdometryPacket;
        output port ODOMETRY type generic port packet OdometryPacket;
        output port LASER_SCAN type generic port packet LaserPacket;
        output port LASER_GEOMETRY type generic port packet PacketFrame3D;
        entry state Main
        {
```

```
        //State's body
        transition on COMMANDS;
        }



    };
```

7.9. Fase de evaluación: Tests.

Para la realización de la evaluación del algoritmo MbICP sobre la plataforma CoolBOT se partió del test de evaluación en Matlab cuyos resultados se habían depurado en detalle hasta validar su correcto funcionamiento.

Trasladando los datos de simulación se implementó un primer test con 689 casos de prueba donde únicamente se evalua la solución obtenida entre el algoritmo MbICP en el instante $i+1$ de CoolBOT, frente a la de Matlab en solución y número de iteraciones. Cuyas matrices figuran en el fichero "MbICP-corrector-test.cpp".

Partiendo de estas pruebas se ha podido constatar que existe una ligera diferencia de resultados, donde las correciones efecuadas por CoolBOT tiene una desviación de 0.003m en el eje x, 0.005m en el eje y y 0.0001 radianes en θ.

Una vez evaluado y parcialmente testeado, el siguiente paso ha sido trasladar la implementación del "test del pasillo" para llevar a cabo una evaluación más exhaustiva de como se comporta el robot en un entorno algo más parecido al entorno real de ejecución. Generando una aplicación para generar conjuntos de puntos de test que cumplan las restriciones físicas impuestas por la figura 7.13 y que se descríben con mayor detalle en el apéndice E. Que recorrerá un pasillo con dos tramos de tamaño 300 mm de largo y 100mm de ancho como se puede ver en la figura inferior.

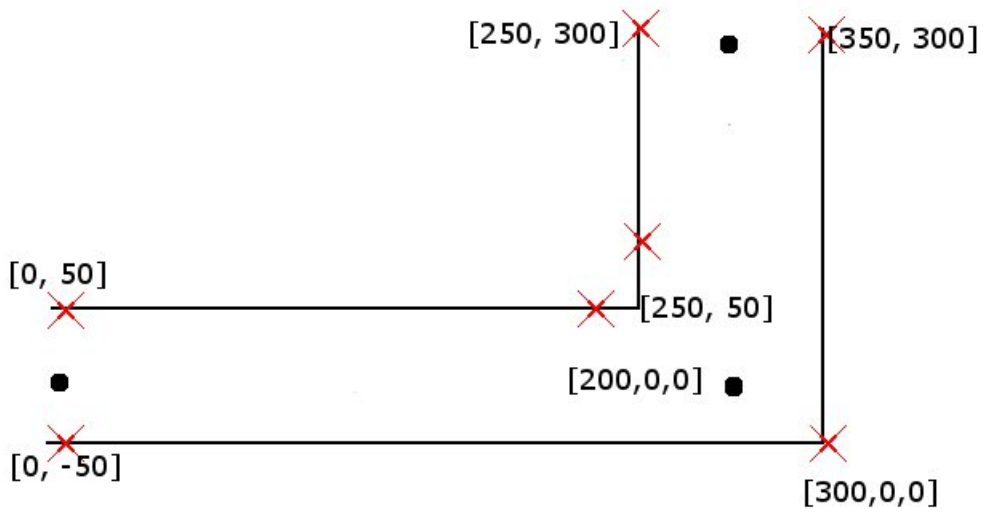


Figura 7.13: Esquema del mapa que recorrerá el robot

Con este test podiamos evaluar que los datos de correspondencia asi como los valores de solución encajan con los valores matemáticamente cálculados previamente. Certificando con esto que el algoritmo da la solución esperada en cada instante y no existen posibles errores escondidos en el traslado de datos de simulación en el test anterior.

En último término, se generó un fichero de test llamado "salida.txt" donde Matlab volcó todos los datos necesarios para realizar la simulación en C++ cargados mediante el fichero MbICPCorrectorTest.cpp. Automatizando con ello El formato empleado es:

- odometry[1]: *timestamp*: x_i, y_i, θ_i, $Odom_i$
- laser_geometry: x_i, y_i, θ_i, α_i, β_i, γ_i
- laser_scan:*timestamp*:x_1, y_1, x_2, y_2, ... , x_n, y_n
- odometry_result: *timestamp*: n_{iter}:x_i, y_i, θ_i, $Odom_i$

Por consiguiente, este fichero almacenará por un lado la información de evaluación en el desplazamiento del sistema robótico simulado y a su vez los resultados obtenidos en Matlab mediante la variable "odometry_result" que contendrá:

- $timestamp_i = timestamp_{odometría_i} + t_{ejecución\ MbICP}$

[1]Como se puede observar la única diferencia entre la salida de CoolBOT y Matlab es que Matlab no trabaja con la velocidad del robot.

- número de iteraciones ejecutadas.
- Solución en términos de sistema de referencia $\triangle_i^{i+1} = (x, y, \theta)$.

Una vez evaluado mediante estos tests el algoritmo ya se pudo proceder a evaluar el "MbICPCorrector" en su conjunto por medio de la integración "MbICP" descrita en la sección 7.7. Utilizando para ello la integración sobre el simulador Stage del proyecto Player/Stage.

7.10. Estudio del comportamiento.

Tras haber implementado el algoritmo MbICP sobre la plataforma CoolBOT, pasamos a estudiar como evoluciona su comportamiento en el tiempo. Para ello, usaremos Player/Stage [25] y nuestra integración MbICPIntegration.

Player/Stage proporciona el servidor de comunicación y el simulador del robot, junto a una colección de mapas de ejemplo, que sirve de soporte de datos para evaluar el comportamiento del algoritmo MbICP. El mapa seleccionado es "everything.cfg", figura 7.14, al que se le ha introducido ruido aleatorio con una cota máxima de $x = 0{,}02$ metros, $y = 0{,}03$ metros y $\theta = 0{,}001$ radianes.

La integración "MbICPIntegration" contiene los componentes, vistas y paquetes que permiten la ejecución del algoritmo MbICP sobre la plataforma CoolBOT, ilustrada por la figura 7.10. Además, la integración incluye otros módulos como son: mapa del mundo, esquivador de obstáculos, planificador, logger, etc. Para realizar un estudio más completo del comportamiento de nuestra implementación.

Partiendo inicialmente del robot situado en el extremo izquierdo del mapa, figura 7.14, se lo ha hecho desplazar en línea recta hacia la derecha del mapa siguiendo una trayectoria prácticamente recta.

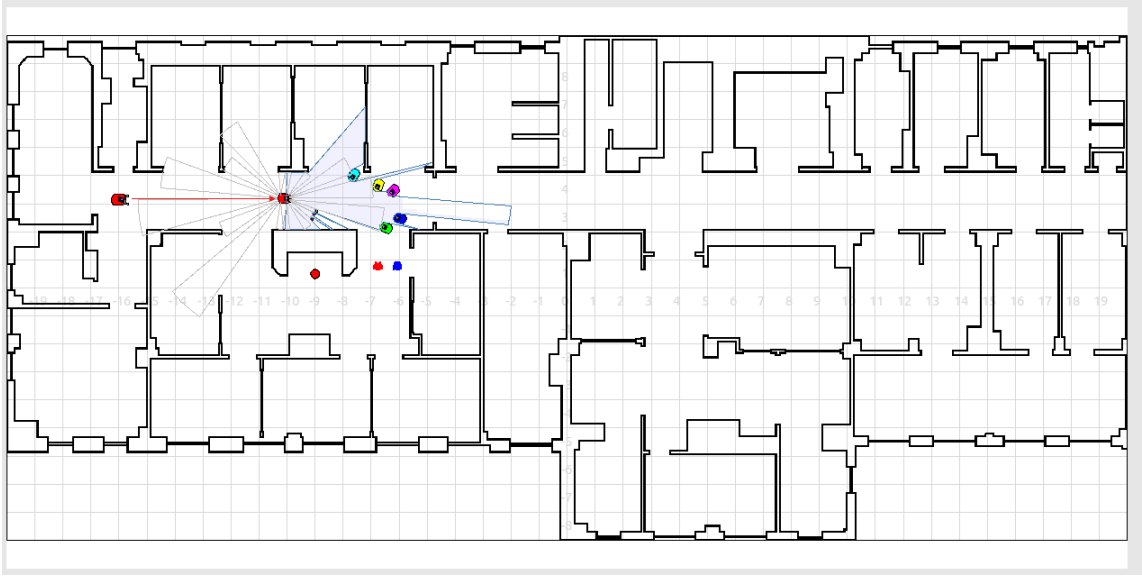


Figura 7.14: Mapa "everything.cfg" del conjunto de mapa de la aplicación Stage

A lo largo de su desplazamiento horizontal comenzamos a observar en la imagen izquierda de la figura 7.15, se produce una acumulación de error conforme se incrementa el desplazamiento horizontal, dando lugar a una desviación considerable a los pocos segundos segundos de comenzar la ejecución. Lo quiere decir que el robot no es capaz de estimar la odometría correctamente.

En consecuencia, este desplazamiento erróneo requeriría por parte del planificador estar recalculando rutas constantemente para alcanzar el destino, generando así una incertidumbre considerable para conocer si el robot llegará al destino fijado por parte del usuario que ejecutó la orden.

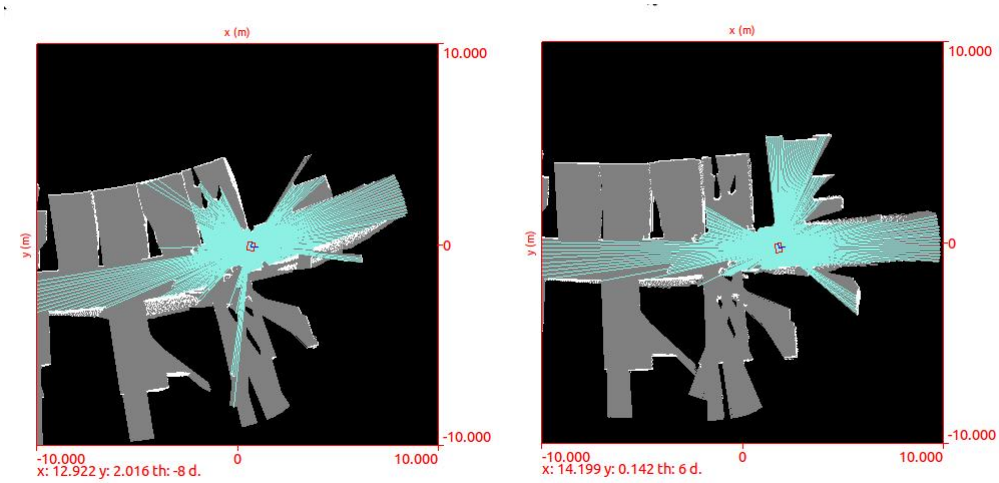


Figura 7.15: (izquierda) Mapa generado por la vista NavigationMap sin efectuar correción de odometría. (derecha)Mapa generado por la vista NavigationMap con correción mediante el algoritmo MbICP.

Por otro lado, el uso del algoritmo MbICP, como se observa en la imagen derecha de la figura 7.15, logra que el desplazamiento horizontal del robot se mantenga practicamente constante a lo largo del recorrido trazado, porque el sistema es realimentado con la odometría corregida en cuanto esta disponible por parte del componente MbICPCorrector. Atenuando considerablemente el error producido por el sistema y reduciendo la carga de trabajo por parte del planificador que muestra como la ruta no se ve alterada con tanta facilidad.

En conclusión, la incorporación del algoritmo MbICP al sistema CoolBOT mejora el comportamiento del global sistema. Incrementando su precisión a la hora de trazar rutas en el mapa interno del robot y reduciendo la carga computacional del planificador y el evitador de obstáculos porque no se ven forzados a corregir tan habitualmente la ruta del robot durante su ejecución.

7.10.1. Comparativa de implementaciones: Matlab vs C++.

A lo largo del desarrollo en CoolBOT se han realizado sucesivas comprobaciones para validar que los resultados obtenidos en CoolBOT concuerdan con los resultados obtenidos en Matlab para los mismos datos. Este proceso permitió establecer una comparativa en tiempo y número de iteraciones del algoritmo en ambas plataformas y así estudiar más detenidamente la influencia del lenguaje de programación en su rendimiento global. Los resultados de tal estudios se muestran en la tabla 7.1

Variable	Matlab	CoolBOT
Nº Iteraciones promedio	9,436865	1,805515
Tiempo promedio	$\simeq$ 1,5 segs	12,499299 ms
Nº Máx. Iteraciones	57	50
Nº Mín. Iteraciones	2	1
Soprepasado Límite Nº Iteraciones	3	1

Cuadro 7.1: Comparativa ejecución algoritmo MbICP entre Matlab y CoolBOT

En consecuencia, Matlab un entorno claramente científico donde se tienen muy presentes los posibles errores de redondeo y todos datos forman una estructura solida bien evaluada y robusta. Por ello, el algoritmo MbICP en Matlab obtuvo un conjunto de soluciones más preciso a costa de largos tiempos de ejecución.

En cambio, C++ un lenguaje claramente orientado al rendimiento carece de cálculo matricial, transformaciones de coordenadas, etc. unido a la librería Eigen [18] permiten tener un gran soporte para conservar la eficiencia del lenguaje. Al estudiar los resultados obtenidos se observa una penalización de precisión aunque es mínima y su tiempo de ejecución es considerablemente inferior al de Matlab.

Consecuentemente, en términos de rendimiento el lenguaje C++ efectua los cálculos a un promedio de 30ms por ejecución frente a los segundos que requiere la plataforma Matlab para llevar a cabo cada ejecución del algoritmo MbICP. Esto viene justificado porque el lenguaje C++ posee un gran número de optimizaciones de código en tiempo de compilación como son: el desenrollamiento de bucles, renombramiento de variables, etc. mientras que Matlab al ser un lenguaje interpretado premiando la precisión al rendimiento.

7.11. Resultados y conclusiones.

El desarrollo del proyecto sobre la plataforma CoolBOT, facilitó considerablemente el trabajo gracias al uso de plantillas que permitieron la generación más estructurada del código.

No obstante, actualmente CoolBOT carece de documentación propia dificultando su manejo y alargando el tiempo de desarrollo de software eficiente para la plataforma. Acudiendo en muchos casos a proyectos fin de carrera de otros alumnos para la comprensión de aspectos tales como: multihilos, conexión de módulos mediante "CMake" ó el manejo correcto del fichero ".coolbot-enviroment".

Por otro lado, tras implementar el algoritmo MbICP se ha podido constatar la importancia de los algoritmos de "scan-matching" sobre el sistema robótico. A pesar de que el sistema robótico "Pioneer 3" posee un buen evitador de obstaculos y planificador, los errores introducidos por el desplazamiento a lo largo del mundo real que habitualmente no son captados adecuadamente por los sensores odométricos. Esto provoca una diferencia en la percepción interna del robot y el mundo real donde se ejecuta.

Por tanto, el algoritmo MbICP agrega a la plataforma CoolBOT una herramienta con la que poder paliar errores indeseados. Liberando de la carga computacional al evitador de obstaculos y se mejorará la planificación debido a que existirá una aproximación mejorada de la odometría del robot en el mundo.

Capítulo 8

Resultados y conclusiones.

A lo largo del desarrollo de este proyecto ha sido estudiado en detalle el algoritmo MbICP e implementado en la plataforma CoolBOT que era el objetivo principal de este proyecto. Este proceso ha sido enriquecido por la realización de un prototipo en Matlab encargado de analizar y estudiar el comportamiento del algoritmo, como paso previo a su implementación en CoolBOT.

Al estudiar la formulación del algoritmo MbICP se pudo ir profundizando en el estudio de los artículos científicos utilizados hasta el momento. Comprendiendo en detalle la mejora que suponen los algoritmos de scan-matching sobre el comportamiento del robot basado sólo en sus sensores odométricos.

Este proyecto supuso un complejo desarrollo haciéndose esencial una buena planificación y una aplicación eficiente de la ingeniería del software para llevar un desarrollo estructurado, cumpliendo poco a poco los plazos fijados, siendo capaces de salvar los casos de riesgo imprevistos. Un ejemplo de esto fue la rutura del disco duro a mitad del desarrollo, este fallo podría haber ocasionados serios problemas pero en la planificación se incluían copías de seguridad diarias que evitaron que este problema hardware ocasionará un segundo comienzo de 0 del proyecto.

Otro de los objetivos perseguidos en este proyecto es la aplicación de los conocimientos adquiridos en la carrera en este proyecto. Este objetivo fue cubierto en bastantes disciplinas como son: las matemáticas mediante el estúdio del algoritmo MbICP, la instrumentación mediante el modelado de los sensores del robot Pioneer, la programación mediante Matlab y C++ sobre la plataforma CoolBOT, etc.

Por otro lado, las aportaciones de este proyecto son los siguientes:

- Consecución de los objetivos especifícados en la sección 1.4 obteniéndose unas aplicaciones útiles e intuitivas.
- Se ha realizado un desarrollo más ámplio de lo que en principio se había planteado, añadiendo opciones que facilitan el manejo y estudio del algoritmo MbICP sobre un prototipo en Matlab y mejoran el tiempo de respuesta en la implementación sobre la plataforma CoolBOT.
- Se han ampliado los conocimientos en la titulación sobre todo en las disciplinas de sistemas robóticos móviles e ingeniería del software.

Para concluir, vamos a hablar de los posibles **trabajos futuros** para este proyecto y el algoritmo MbICP.

- Este proyecto tiene la posibilidad de extenderse a otros sistemas de percepción capaces de generar un mapa de puntos como son los sensores sonar, las camaras, etc. Adaptando el algoritmo MbICP al sistema de percepción del robot ya que existen sistemas robóticos que pueden no tener integrado el sensor de rango como parte de su sistema de percepción del mundo.
- Otra posible vía es ampliar el algoritmo MbICP a las tres dimensiones (3D) que actualmente resulta muy útil en las camaras que disponen de imágenes 3D (compuestas por conjuntos de puntos en el espacio) para percibir el mundo que los rodea.

Apéndice A

Manual de usuario prototipo Matlab.

A.1. Requisitos.

El requisito básico es tener instalado la distribución Matlab.

A.2. Instalación.

Para llevar a cabo una instalación satisfactoria, realizar los siguientes pasos:

1. Descargar la aplicación del servidor git:
   git clone http://newmozard
2. Abrir Matlab y añadir al path del sistema el directorio y subdirectorios descargados.
3. Cambiar el directorio actual de Matlab al directorio raíz donde se encuentra el interfaz.fig
4. Pulsar sobre la tecla guide y cargar interfaz.fig.
5. Presionar sobre ejecutar o presionar la flecha de reporducir verde en el menú de herramientas de guide.

Una vez realizados estos pasos ya tenemos el interfaz funcionando con el prototipo en Matlab y preparado para ejecutar simulaciones.

A.3. Estructura de ficheros.

La organización del prototipo es la siguiente:

- Carpeta "test/" los test de pruebas incorporados con la aplicación.
- "interfaz.fig" e "interfaz.m" que poseen el interfaz gráfico de la aplicación.
- El algoritmo MbICP se encuentra en los ficheros "controlador_pintura_interfaz.m", "actualiza_estado_algoritmo_MbICP.m", "calc_closest_points.m", "leasure_square_minimization.m", "extraer_error.m" y "distance_measure.m".

- Las herramientas de visualización se encuentran en: "pintar_robot.m","pintar_sistema_referencia.m", "pintar_trayectoria.m" y "pintar_sistema_referencia.m" .
- El reproductor del robot se encuentra en: "cambia_estado_interfaz.m".
- Adiccionalmente se han creado funciones orientadas a las transformaciones de coordenadas que son: "calc_pol2cart.m", "filter_distMax.m", "transformar_q_al_origen.m", "transformar_i_sig_a_i.m".
- El gestor de persistencia esta en: "gestor_persistencia.m".
- Cada extensión posible tiene un procesamiento correspondiente ubicados en: "procesar_formato_a.m", "procesar_formato_log.m" y "procesar_formato_script.m".
- Existe una función capaz de generar test que se encuentra en: "genera_tests.m" y "genera_puntos_test.m"

A.4. Como usarlo.

Previamente a comenzar a usarlo hay que tener presente si el fichero a simular esta ó no adaptado al formato Matlab. En caso de estar procesado, a la hora de usarlo sencillamente que cargar el fichero ".mat" de la siguiente forma:

- Presionar archivo->abrir y selecionar el fichero que se desea ejecutar.
- Una vez cargada la simulación, exiten diferentes formas de reproducción:
  - Reproducir solo el desplazamiento del robot mediante el menu de herramientas: "Avanzar", "Retroceder", "Iterar", "Detener" y "Reiniciar". Donde iterar puede hacerse incluyendo en su funcionamiento la ejecución simultánea del algoritmo MbICP, si se desea.
  - Hacer funcionar el algoritmo MbICP mediante el panel "MbICP" cuyas posibilidades son: "Paso", "Automático" y "Reiniciar". "Automático" se encarga ejecutar el algoritmo MbICP hasta converger a una solución.

En caso de ser necesario adaptar el formato del fichero a Matlab, es necesario llamar al gestor de persistencia cuya función es identificar la extensión del fichero y procesarlo para generar un formato legible para el prototipo Matlab. No obstante, si se desconoce la extensión en el gestor se avisa al usuario de que el fichero no tiene una extensión reconocible para el sistema.

Apéndice B

Manual usuario CoolBOT.

B.1. Paquete "MbICP-Packets"

B.1.1. Requisitos.

Para poder hacer uso de este paquete es necesario tener instalado y funcionando CoolBOT.

B.1.2. Instalación.

La instalación requiere tener conocimiento de la estructura de ficheros de CoolBOT, principalmente "coolbot-envoirement" ubicado en el directorio cuando se halla instalado siguiendo los pasos para su instalación. Partidiendo de esta base los pasos a llevar a cabo para instalar este paquete son:

1. Descargarlo desde el servidor Git mediante el comando:
2. Accedemos a la carpeta y efectuamos una compilación del paquete dentro del subdirectorio "build" mediante el comando:
   Cmake ..
3. Una vez compilado habrán sido generado los ficheros ".lib" en la carpeta *lib/* y ".pkg" en la carpeta *pkg-config/*, comprobamos que así ha sido.
4. Nos vamos al fichero ".coolbot-envoirement" y añadimos al path de liberias indicado por COOLBOT_LIB_PATH y ficheros pkg-config indicado por COOLBOT_PKG_PATH.

Relizados estos pasos ya es posible incluir nuestro paquete en cualquier componente, vista o integración según se necesite. Para incluirlo hay que seguir los siguientes pasos:

- Incluir el fichero cabecera

  ```
  include "MbICP-Packets.h";
  using namespace MbICP-PacketsSpace;
  ```

- Añadir al fichero "CMakeList.txt" las liberías y ficheros que debe buscar.

```
GET_FLAGS_PKGCONFIG("mbicp-packets")
SET(MBICP_PACKETS_CFLAGS "${PKG_CFLAGS}")
GET_LIBS_PKGCONFIG("mbicp-packets")
SET(MBICP_PACKETS_LIBS "${PKG_LIBS}")
```

- Seguidamente en el "CMakeList.txt" añadir las librerias y ficheros a la ruta de compilación del Cmake.

```
SET(FLAGS_BIN_COMPILER "${FLAGS_BIN_COMPILER} ${MBICP_PACKETS_CFLAGS}")
SET(FLAGS_BIN_LINKER   "${FLAGS_BIN_LINKER} ${MBICP_PACKETS_LIBS}")
SET(FLAGS_LIB_COMPILER "${FLAGS_LIB_COMPILER} ${MBICP_PACKETS_CFLAGS}")
SET(FLAGS_LIB_LINKER   "${FLAGS_LIB_LINKER} ${MBICP_PACKETS_LIBS}")
```

B.1.3. Ejecución.

Este paquete es un contenedor que almacena datos para transitar la información por la red. No obstante su ejecución es a traves de un conjunto de funciones "set/get" seguido del nombre de la variable en cada caso.

B.2. Componente "MbICP-corrector".

B.2.1. Requisitos.

Para poder hacer uso de este paquete es necesario tener instalado y funcionando CoolBOT. Además de este requisito, este componente hace uso del paquete "MbICP-Packets" para enviar informaciónde simulación por el puerto, por este motivo, cada vez que se desee incluir este componente previamente hay que añadir el paquete al sistema.

B.2.2. Instalación.

La forma de instalarlo no cambia mucho con respecto a la instalación de un paquete, únicamente varía en el lugar donde encontramos el repositorio, siendo los pasos a seguir:

1. Descargarlo desde el servidor Git mediante el comando:

2. Accedemos a la carpeta y efectuamos una compilación del paquete dentro del subdirectorio "build" mediante el comando:
   Cmake ..

3. Una vez compilado habrán sido generado los ficheros ".lib" en la carpeta *lib/* y ".pkg" en la carpeta *pkg-config/*, comprobamos que así ha sido.

4. Nos vamos al fichero ".coolbot-envoirement" y añadimos al path de liberias indicado por COOLBOT_LIB_PATH y ficheros pkg-config indicado por COOLBOT_PKG_PATH.

Una vez incorporado nuestro componente al sistema operativo para incluirlo dentro de una integración es necesario:

- Incluir el fichero cabecera

```
include "MbICPCorrector.h";
using namespace MbICPCorrectorSpace;
```

- Añadir al fichero "CMakeList.txt" las liberías y ficheros que debe buscar.

```
GET_FLAGS_PKGCONFIG("mbicp-corrector")
SET(MBICP_CORRECTOR_CFLAGS "${PKG_CFLAGS}")
GET_LIBS_PKGCONFIG("mbicp-corrector")
SET(MBICP_CORRECTOR_LIBS "${PKG_LIBS}")
```

- Seguidamente en el "CMakeList.txt" añadir las librerias y ficheros a la ruta de compilación del Cmake.

```
SET(FLAGS_BIN_COMPILER "${FLAGS_BIN_COMPILER} ${MBICP_CORRECTOR_CFLAGS}")
SET(FLAGS_BIN_LINKER   "${FLAGS_BIN_LINKER} ${MBICP_CORRECTOR_LIBS}")
SET(FLAGS_LIB_COMPILER "${FLAGS_LIB_COMPILER} ${MBICP_CORRECTOR_CFLAGS}")
SET(FLAGS_LIB_LINKER   "${FLAGS_LIB_LINKER} ${MBICP_CORRECTOR_LIBS}")
```

B.2.3. Ejecución.

Para ejecutar el componente es necesario incorporarlo en una integración CoolBOT para que pueda recibir los paquetes necesarios para su ejecución y mostrar su comportamiento dentro del sistema. Siguiendo los siguientes pasos en la integración:

- Declarar la instancia de la vista mediante la siguiente sentencia

```
component mbicpInstance: MbICPCorrector;
```

- Conectar los puertos adecuadamente a los componentes que suministrarán la información para su ejecución.

```
connect robot:ODOMETRY                  to mbicpInstance:ODOMETRY;
connect robot:LASERGEOMETRY             to mbicpInstance:LASER_GEOMETRY;
connect robot:LASERSCAN                 to mbicpInstance:LASER_SCAN;
```

No obstante, cuando se quiere evaluar el comportamiento del componente existe un fichero llamado "mbicp-corrector-test.cpp" ubicado dentro de la carpeta "*src/examples*" que contiene un programa de evaluación en cuatro fases con un total aproximado de 3000 casos de prueba con valores distintos. Para ejecutarlo, únicamente es necesario seguir los siguientes pasos:

1. Compilar la aplicación mediante el comando "*make*" en la carpeta "*build/*"
2. Dirigirse a la carpeta "*bin/*" donde se encuentra el binario.
3. Ejecutarlo mediante el comando "*./mbicp-corrector-test*".

B.2.4. Interpretación de los datos.

Los datos de salida del componente durante su funcionamiento, su primera salida esta orientada a la ejecución a tiempo real con únicamente el paquete de la odometría corregida y la segunda salida contiene información de depuración para la visualización del comportamiento del algoritmo tras cada ejecución del mismo, mostrando aspectos como los barridos del sensor de rango, sus correspondientes odometría y los resultado obtenidos usados durante su ejecución.

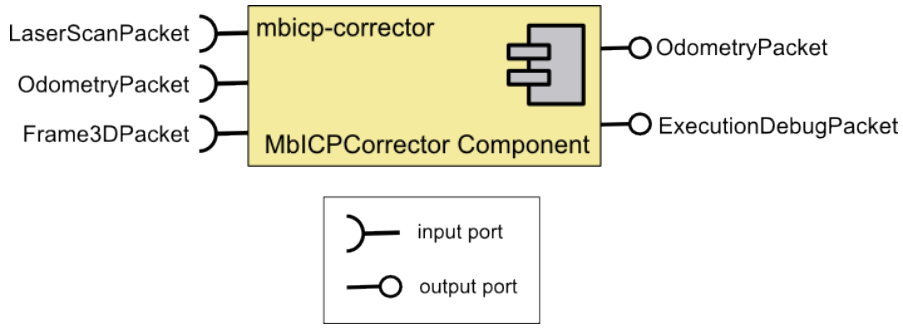


Figura B.1: Componente MbICPCorrector

Como se puede ver en la figura superior es muy importante conocer que "ExecutionDebugPacket" contiene importante información para el análisis del funcionamiento del algoritmo y en base a ellos podemos evaluar el comportanmiento del algoritmo MbICP.

B.3. Vista "MbICP-gtk".

B.3.1. Requisitos.

Para el funcionamiento de la vista adecuadamente, es necesario haber incorporado al sistema operativo la plataforma CoolBOT, el paquete "MbICPPackets" y el componente "MbICPCorrector". En caso, de no saber al forma de hacerlo diríjase a los puntos anteriores para más información.

B.3.2. Instalación.

La forma de instalarlo no cambia mucho con respecto a la instalación de un componente o un paquete, únicamente varía en el lugar donde encontramos el repositorio, siendo los pasos a seguir:

1. Descargarlo desde el servidor Git mediante el comando:

2. Accedemos a la carpeta y efectuamos una compilación del paquete dentro del subdirectorio "build" mediante el comando: ***"Cmake .."***

3. Una vez compilado habrán sido generado los ficheros ".lib" en la carpeta *lib/* y ".pkg" en la carpeta *pkg-config/*, comprobamos que así ha sido.

4. Nos vamos al fichero ".coolbot-envoirement" y añadimos al path de liberias indicado por COOLBOT_LIB_PATH y ficheros pkg-config indicado por COOLBOT_PKG_PATH.

Una vez incorporado nuestro componente al sistema operativo para incluirlo dentro de una integración es necesario:

- Incluir el fichero cabecera

```
include "MbICPGtk.h";
using namespace MbICPGtkSpace;
```

- Añadir al fichero "CMakeList.txt" las liberías y ficheros que debe buscar.

```
GET_FLAGS_PKGCONFIG("mbicp-gtk")
SET(MBICP_GTK_CFLAGS "${PKG_CFLAGS}")
GET_LIBS_PKGCONFIG("mbicp-gtk")
SET(MBICP_GTK_LIBS "${PKG_LIBS}")
```

- Seguidamente en el "CMakeList.txt" añadir las librerias y ficheros a la ruta de compilación del Cmake.

```
SET(FLAGS_BIN_COMPILER "${FLAGS_BIN_COMPILER} ${MBICP_GTK_CFLAGS}")
SET(FLAGS_BIN_LINKER   "${FLAGS_BIN_LINKER} ${MBICP_GTK_LIBS}")
SET(FLAGS_LIB_COMPILER "${FLAGS_LIB_COMPILER} ${MBICP_GTK_CFLAGS}")
SET(FLAGS_LIB_LINKER   "${FLAGS_LIB_LINKER} ${MBICP_GTK_LIBS}")
```

B.3.3. Ejecución.

Para ejecutar la vista es necesario incorporarla en una integración CoolBOT para que pueda recibir los paquetes necesarios para su ejecución y mostrar su comportamiento dentro del sistema. Siguiendo los siguientes pasos en la integración:

- Declarar la instancia de la vista mediante la siguiente sentencia

```
view mbicpRelaseView: mbicp_gtk with description "Mbicp View";
```

- Conectar los puertos adecuadamente a los componentes que suministrarán la información para su ejecución.

```
connect robot:ODOMETRY                   to mbicpRelaseView:ODOMETRY;
connect robot:LASERGEOMETRY              to mbicpRelaseView:LASER_GEOMETRY;
connect robot:LASERSCAN                  to mbicpRelaseView:LASER_SCAN;
connect mbicpInstance:ODOMETRY_CORRECTED to mbicpRelaseView:ODOMETRY_CORRECTED;
connect robot:ROBOTCONFIG                to mbicpRelaseView:ROBOT_CONFIG;
connect mbicpInstance:EXECUTION_INFO     to mbicpRelaseView:EXECUTION_INFO;
```

- Recordar: Cuando se ejecute el comando de compilación de la integración será necesario incorporar al constructor de la vista el periodo de refresco del interfaz.

B.4. Integración "MbICP-integration".

B.4.1. Requisitos.

La integración es un caso particular pues sus requerimientos varian de los componentes que quieras incorporar dentro del sistema a ejecutar aunque hay un denominador común CoolBOT debe estar instalado en el sistema operativo. En nuestro caso, hacemos uso de todos los componentes, vistas y paquetes de CoolBOT, añadidos al compenente, vista y paquete MbICP creados durante este proyecto fin de carrera.

B.4.2. Instalación.

Para llevar a cabo la integración de los módulos de la plataforma CoolBOT, únicamente es necesarios añadirlos en el CMakeList de la siguiente forma:

- Añadir al fichero "CMakeList.txt" las liberías y ficheros que debe buscar.

```
GET_FLAGS_PKGCONFIG("nombre_módulo")
SET(NOMBRE_MODULO_CFLAGS "${PKG_CFLAGS}")
GET_LIBS_PKGCONFIG("nombre_módulo")
SET(NOMBRE_MODULO_LIBS "${PKG_LIBS}")
```

- Seguidamente en el "CMakeList.txt" añadir las librerias y ficheros a la ruta de compilación del Cmake.

```
SET(FLAGS_BIN_COMPILER "${FLAGS_BIN_COMPILER} ${NOMBRE_MODULO_CFLAGS}")
SET(FLAGS_BIN_LINKER   "${FLAGS_BIN_LINKER} ${NOMBRE_MODULO_LIBS}")
SET(FLAGS_LIB_COMPILER "${FLAGS_LIB_COMPILER} ${NOMBRE_MODULO_CFLAGS}")
SET(FLAGS_LIB_LINKER   "${FLAGS_LIB_LINKER} ${NOMBRE_MODULO_LIBS}")
```

Una vez incorporados al "CmakeList.txt" es necesario ejecutar el comando siguiente

```
coolbot-c src/mbicp-integraction.coolbot-integration
```

para generar el fichero "mbicp-integraction.cpp" que contiene el esqueleto en C++ de la integración y al que hay incorporar los ficheros de cada componente, vista y paquete añadido a la integración para que su complicación posterior sea satisfactoria.

```
include "nombre_modulo.h";
using namespace nombreModuloSpace;
```

En suma, para cada compenente, vista y paquete que tenga parámetros adiccionales en su es necesario incluirlos en su constructor, por ejemplo el periodo de refresco en las vistas, o la siguiente funcion en el main para elegir la ip donde se desea ejecutar el interfaz:

```
argc--;
argv++;
if(!argc)
```

```
{
   cout<<"usage: "<<pName <<" localhost | <tcp-ip-address>"<<endl;
   return 0;
}
cout<<"Robot connection: "<<argv[0]<<endl;
```

Añadiendo esto para terminar la instalación de la integración solo resta desplazarnos a la carpeta "build/" y ejecutar el siguiente comando:

make

B.4.3. Ejecución.

Para lanzar la ejecución de esta integración hay que seguir los siguientes pasos:

1. Abrir el "player" con el mapa que se desea simular, (ejemplo, player /path/of/worlds/world.everything
2. Ir al directorio "bin/" de la integración y ejecutar el comando:

```
./mbicp-integration localhost (ó la ip de la maquina donde se desea ejecutar)
```

Al realizar estos pasos ya tenemos en ejcución el interfaz con todos los componentes, vistas y paquetes incorporados.

Apéndice C

Formatos de fichero soportados.

C.1. Extensión ".script"

El formato ".script" posee el siguiente formato:

- POS timestamp: X Y Z
- LASER-RANGE timestamp ángulo_inicial número_puntos ángulo_final: distandia_1 ... distancia_n

Las unidades empleadas son: mm, radianes y segundos. Además, cabe destacar que el orden de las líneas en el archivo refleja el orden en que se han recibido datos del robot, como resultado, las marcas de tiempo de datos están organizadas cronológicamente.

Para clarificar el formato vemos el siguiente ejemplo de formato:

```
POS         976052857 337284: 0.000000 0.000000 -0.140850
LASER-RANGE 976052857 337530 0 180 180.0: 107.0 ... 105.0
POS         976052857 337916: 0.000000 0.000000 -0.140850
LASER-RANGE 976052857 348896 0 180 180.0: 108.0 ... 105.0
...
POS         976055548 522255: -5088.399900 -3582.500000 -214.577450
LASER-RANGE 976055548 624744 0 180 180.0: 100.0 ... 112.0
POS         976055548 625127: -5088.399900 -3582.500000 -214.577450
```

Notesé que los espacios en blanco han sido añadidos para mejorar la legibilidad del ejemplo, la idea es que solo exista un único espacio en blanco.

C.2. Extensión ".log"

Toma todos los datos de entrada: odometría, barrido laser, geometría y opcionalmente esta soportada su respectiva solución, son incluidos en el fichero con extensión ".log" donde cada instante del robot viene reperensentado por la siguiente tupla de valores:

- odometry: timestamp (formato Matlab): X, Y, θ, V_x, V_y, ω

- laser-scan: timestamp (formato Matlab): Número elementos(n): x_1, y_1, x_2, $y_2, \cdots x_n$, y_n
- geometry: X, Y, θ, α, β, γ
- odometry-result: timestamp (formato Matlab): X, Y, θ, V_x, V_y, ω

La tupla odometria-barrido-geometría debe ir siempre agrupada para ser identificada. Esta extensión establece la correspondencia por su posición en el fichero y no por su estampa de tiempo . No es posible encontrar lecturas dos odometrías, barridos o geometrías consecutivas. Dando lugar a un fichero con el siguiente contenido:

```
tupla_1
odometry-result_1: ...
tupla_2
tupla_3
odometry-result_2: ...
odometry-result_3: ...
...
tupla_n:
odometry-result_n: ...
```

No obstante, la solución del algoritmo MbICP que se traduce a un paquete de odometría asíncrono pudiendo encontrar dos ó más líneas en el fichero seguidas de resultados. Además es necesario destacar que cada elemento de la tupla de valores esta almacenado en una línea diferente.

Para concluir, añadimos un pequeño ejemplo de valores reales obtenidos por el RobotLogger.

```
odometry:        2011 05 19 22 22 50.728448:0,0,0,-1.96721e-44,0,0,0
laser-scan:      2011 05 19 22 22 50.728448:-4.7457e-08,-1.08569,...,6.36921e-08,0.935997
laser-geometry:  0.03,0,0,1,1,1
odometry:        2011 05 19 22 23 38.340441:0,0,-0.0350518,-1.96721e-44,0,0,-0.349066
laser-scan:      2011 05 19 22 23 38.340441:-4.75077e-08,-1.08685, ... ,6.38931e-08,0.93895
laser-geometry:  0.03,0,0,1,1,1
odometry:        2011 05 19 22 23 38.340848:0,0,-0.0351939,-1.96721e-44,0,0,-0.349066
laser-scan:      2011 05 19 22 23 38.340848:-4.75077e-08,-1.08685, ... ,6.38931e-08,0.93895
laser-geometry:  0.03,0,0,1,1,1
odometry-result: 2011 05 19 22 23 38.339990:0,0,-0.0348944,-4.22795e-143,0,0,-0.349066
odometry:        2011 05 19 22 23 38.439684:0,0,-0.0348944,-1.96721e-44,0,0,0
laser-scan:      2011 05 19 22 23 38.439684:-4.75077e-08,-1.08685, ... ,6.38931e-08,0.93895
laser-geometry:  0.03,0,0,1,1,1
odometry-result: 2011 05 19 22 23 38.438824:8.07991e-05,-0.000120011,-0.0342315,0.000144676,0,0,0
...
odometry:        2011 05 19 22 23 57.098896:0,0,-2.93113,-1.96721e-44,0,0,0
laser-scan:      2011 05 19 22 23 57.098896:-4.16379e-08,-0.952564, ... ,7.46165e-08,1.09654
laser-geometry:0.03,0,0,1,1,1
odometry-result: 2011 05 19 22 23 57.098050:0.0292902,-0.0021409,-2.77347,0.0332882,0,0,0
odometry-result: 2011 05 19 22 23 57.209247:0.0292902,-0.0021409,-2.77376,0.0332882,0,0,0
```

Notesé que los espacios en blanco han sido añadidos para mejorar la legibilidad del ejemplo, la idea es que solo exista un único espacio en blanco.

Apéndice D

Código del algoritmo MbICP en Matlab.

Comenzaremos describiendo el fichero principal llamado "actualiza_estado_ algoritmo_MbICP.m" encargado de procesar los datos de entrada proporcionados por la aplicación y visualizar los resultados en pantalla en cada iteración del algoritmo y la simulación. La estructura que posee es la siguiente:

```
actualiza_estado_algoritmo_MbICP(handles)
...
// Adaptamos los datos a las estructuras internas del algoritmo.
scan_i    ← calc_pol2cart(scan_i);
scan_i_next ← calc_pol2cart(scan_i_next);
...
datos ← calc_closest_points(scan_i,scan_i_next);
leasure_square_minimization(datos);
q_sol ←extraer_error(q_k,q_min);
...
//Actualiza_pantalla_evolucion;
controlador_pintura(handles.evolucion, ...);
si( q_min < Error_max)entonces
   //Actualiza_pantalla_resultado;
   controlador_pintura(handles.resultado, ...);
   return;
fin;
```

Este algoritmo muestra que se llaman a 4 funciones distintas que son la división estructural que se le ha dado al código, donde la primera es una función auxiliar, llamada "calc_pol2cart", encargada de eliminar los puntos que superen un cierto umbral de distancia, para no contaminar la muestra sensorial y transformar los puntos de coordenadas de polares a cartesianas

```
functionn calc_pol2cart(p,vrango,distmax)
indice ← 1;
```

```
....
inicioAnguloScan ← -((vrango(2)-vrango(1))/2.0);
mientras n=1 < nelementos hacer

    Si( p(n) < distmax) entonces
        Xn ← p(n)*cos(inicioAnguloScan + n*angulo);
        Yn ← p(n)*sen(inicioAnguloScan + n*angulo);
        salida(indice) ← [Xn,Yn];
        indice ← indice + 1;
    fin;
    n ← n + 1;

fin;
```

La siguiente función a tener en cuenta, "calc_closest_points", se encarga de hallar las correspondencias entre los puntos de dos scan diferentes, cuya distancia sea mínima y no supere los límites máximos definidos en la simulación.

```
calc_closest_points(scan_i,scan_k, L, Umbral)
...
// Se aplica q(scan_k)
Qk = transformar_i_sig_a_i(q,q_sig,scan_k);
i ← 1;
j ← 1;
//Se calcula los puntos cuya distancia es mínima entre sí
mientras i < numero_elementos_scan_i

    LessDistance ← Umbral * 10000000;
    mientras j < numero_elementos_scan_i+1


        NowDistance ← distance_measure(scan_i,scan_q(i+1));
        if(NowDistance < LessDistance)
           LessDistance ← NowDistance;
           Pcorr        ← ;
           LessPoint    ← k(:,j);
        end;
        j ← j + 1;
    end;
    if( LessDistance < Umbral )
        hayElementos= hayElementos + 1;
        cCorr (:,hayElementos) = Pcorr;
        cArray(:,hayElementos) = LessPoint;
    end;
    i ← i + 1

end;
```

Consecutivamente es necesario explicar un poco de la función "leasure_square_minimization" que es la encargada de calcular los mínimos cuadrados dentro del algoritmo. Este algoritmo consta de aplicar sistemáticamente la formulación matemática.

```
leasure_square_minimization()
...
calcular los sumatorios para cada elemento
de la matriz A y el vector columna B (Véase sección 2.2.2)
...
Aplicar q_min = -A^-1 * B sobre el sumatorio calculado.
...
devolver resultado.
```

Posteriormente aplicamos el operador $\oplus$ mediante la función "extraer_error" que calcula el q que transforma el sistema de referencia q_k a q_{min} como resultado tendremos un nuevo q que medirá la diferencia existente entre los dos sistemas de referencia.

```
extraer_error ()
...
Calcular A_min y O_min expresados siempre con respecto al sistema referencia q_i
Calcular A_k  y O_k
...
Hallar q_sol donde
O_error = A_min * (O_k - O_min)
A^k_min = A_min * A_k = ( cos^k_min   sin^k_min  )
                        ( -sin^k_min  cos^k_min  )
Θ_sol = arctan( sin^k_min / cos^k_min )
```

Por último, describir la implementación de la función de distancia "distance_measure" que implementa el concepto de distancia descrito en el análisis (Véase 2.2.1 en la página 26), su implementación:

```
distance_measure()
...
calculamos distancia, ecuación 2.29 en la página 28.
...
devolver valor calculado.
```

Apéndice E

Test de evaluación del algoritmo MbICP.

Para evaluar el rendimiento del algoritmo MbICP se decidió elaborar un sistema matemático capaz de generar diversos conjuntos de puntos a fin de probrar el caso general de ejecución del algoritmo, donde el robot se mueve por el mundo que lo rodea considerando todos los elementos existentes en el estáticos. En nuestro caso, se ha elegido como mapa un pasillo en L que representa adecuadamente este test, como observamos en la figura E.1.

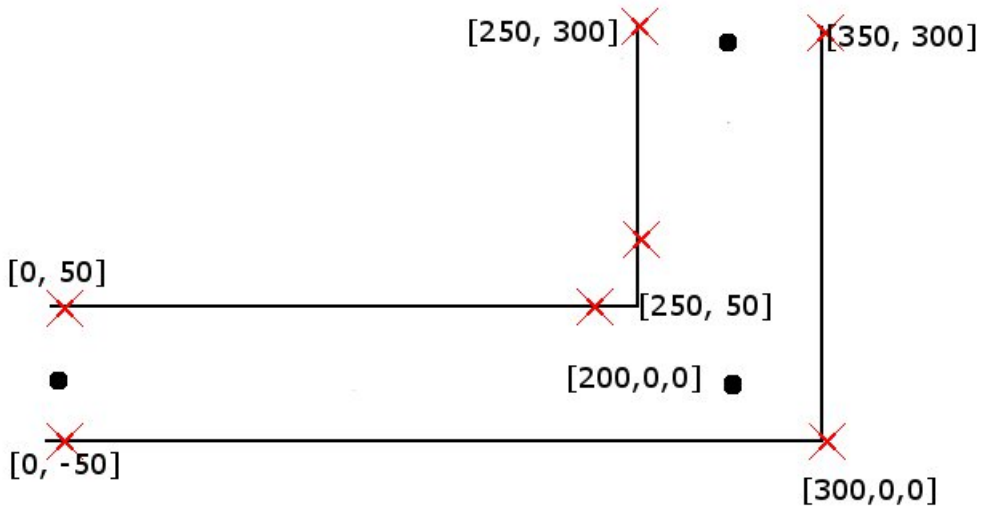


Figura E.1: Mapa del pasillo en L

Consecuentemente 4 rectas que definirán las paredes del camino que ha de recorrer el robot y 6 vértices que definirán los puntos límite. Además, habrá que prestar especial atención a aquellos lugares donde no hay ningún objeto que haga obstaculo y la lectura es la máxima del rango del sensor.

Por otro lado, estas cuatro rectas, que hacen de pared, vienen definidas por las siguientes ecuaciones:

- $R_1 : y = 50$ definida únicamente en el rango $0 \leq x \leq 250$.
- $R_2 : y = -50$ cuyo rango es $0 \leq x \leq 350$.
- $R_3 : x = 250$ en el rango $50 \leq y \leq 300$.
- $R_4 : x = 350$ en el rango $-50 \leq y \leq 300$.

En suma, los vértices clave que utilizaremos para calcular las intercepciones con las rectas y asi obtener con la posición en cada instante del robot son:

- $V_1 = \begin{bmatrix} 0 & 50 \end{bmatrix}$
- $V_2 = \begin{bmatrix} 250 & 50 \end{bmatrix}$
- $V_3 = \begin{bmatrix} 250 & 300 \end{bmatrix}$
- $V_4 = \begin{bmatrix} 350 & 300 \end{bmatrix}$
- $V_5 = \begin{bmatrix} 350 & -50 \end{bmatrix}$
- $V_6 = \begin{bmatrix} 0 & -50 \end{bmatrix}$

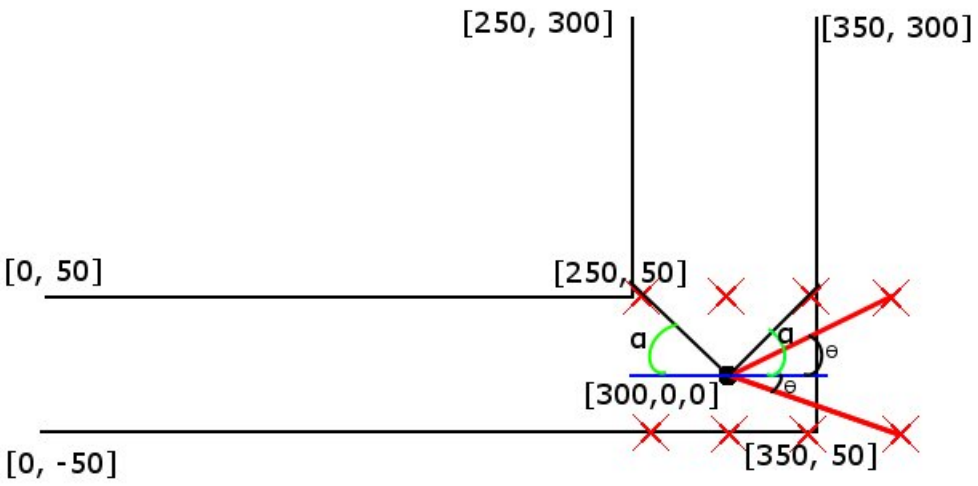


Figura E.2: Ilustración de los ángulos delimitadores del conjunto de puntos

Seguidamente, para que el robot tenga barridos del sensor de rango, es necesario crear un barrido virtual, que vendrá definido como la intercepción de cada vértice V_i con el punto proveniente de la odometría. Como ilustra la figura E.2, cada ángulo θ_i vendrá definido como:

$$\theta_i = \arctan\left(\frac{V_{y_i} - O_y}{V_{x_i} - O_x}\right)$$

Una vez hemos definido los vértices, las rectas y sus ángulos θ_i que definirán nuestro mapa, es hora de acotar el conjunto de valores válidos posibles por parte del sistema para estimar el barrido virtual que efectuará nuestra aplicación.

Para ello, vamos a definir un total de 7 condiciones que delimitarán el punto P^j, el cual representa la estimación de distancia medida por el haz laser j del sensor de rango en ese instante. Estas condiciones determinarán como calcular el valor del punto $(\ x_j,\ \ y_j\ )$ mediante la activación de una única condición que definirá la recta de puntos a emplear para dicha estimación

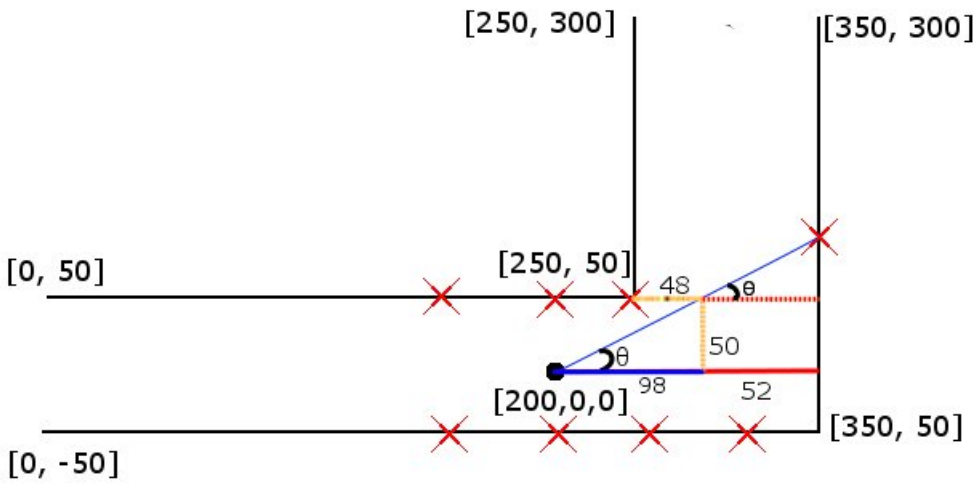


Figura E.3: Evolución del haz virtual del sensor de rango

Por tanto, para evaluar las condiciones es necesario conocer la posición O_i donde se encuentra el robot sobre el mapa y el ángulo θ_j que representa la dirección donde el sensor de rango realiza la captura virtual de distancia, generando la siguientes condiciones:

$$(x_j, y_j) \begin{cases} C_1 : \theta_j < \theta_6 & \rightarrow d = \infty \\ C_2 : \theta_6 < \theta_j < \theta_5 & \rightarrow \left[\; x = x_1 + \frac{-50 - y_1}{\tan \theta_j} \quad y = -50 \; \right] \\ C_3 : \theta_5 < \theta_j < \theta_4 & \rightarrow \left[\; x = 350 \quad y = y_1 + (x - x_1) * \tan \theta_j \; \right] \\ C_4 : \theta_4 < \theta_j < \theta_3 & \rightarrow d = \infty \\ C_5 : \theta_3 < \theta_j < \theta_2 & \rightarrow \left[\; x = 250 \quad y = y_1 + (x - x_1) * \tan \theta_j \; \right] \\ C_6 : \theta_2 < \theta_j < \theta_1 & \rightarrow \left[\; x = x_1 + \frac{50 - y_1}{\tan \theta_j} \quad y = 50 \; \right] \\ C_7 : \theta_1 < \theta_j & \rightarrow d = \infty \end{cases}$$

Por tanto, tenemos 7 condiciones que designarán de donde se tomará el valor de distancia en cada variación del ángulo en la captura del barrido efectuado por el sensor de rango. Además, de que el valor del ángulo θ_j se encuentre dentro de los valores de estas condiciones, existe una cierta relación entre los ángulos acotan las condiciones que son:

- C_3 se puede activar siempre y cuando $C_6 = falso$
- C_4 se puede activar siempre y cuando $\begin{cases} \theta_3 < \theta_2 \\ \theta_4 < \theta_2 \end{cases}$

- C_5 se puede activar siempre y cuando $\theta_3 < \theta_2$
- C_6 se puede activar siempre y cuando $\theta_2 < \theta_1$
- C_7 se puede activar siempre y cuando $\theta_2 < \theta_1$

No obstante, entre las condiciones existe un cierto solapamiento para solucionarlo es necesario definir una precedencia.

1. C_1
2. C_2
3. C_6
4. C_3
5. C_4
6. C_5
7. C_7

Bibliografía

[1] Javier Mínguez, Luis Montesano, Florent Lamiraux, "Metric-Based Iterative Closest Point. Scan Matching for Sensor Displacement Estimation.". IEEE Transactions on Robotics, 2005.

[2] PJ. Besl and N.D. McKay, "A method for registration of 3-D shapes." IEEE Transations on Pattern Analysis and Machine Intelligence, Vol. 14, pp. 239-256, 1992.

[3] Robot Analysis and Control by Slotine, Jean-Jacques E.; Asada, Haruhiko

[4] A. Grossman and R. Poli, "Robust mobile robot localization from sparse and noisy proximetry readings using hough transform and probability grids", Robotics and Autonomous Systems, vol. 37, pp. 1-18, 2001.

[5] IJ. Cox, "Blanche: An experiment in guidance and navigation of an autonomous robot vehicle", IEEE Transactions on Robotics and Automation, vol.7, pp. 193-204, 1991.

[6] J.A. Castellanos, J.D. Tardós, and J. Neira. "Constraint-based mobile robot localization", in Advanced Robotics and Intelligent Systems, IEE, Control series 51, 1996.

[7] Antonio D. Brito, "CoolBOT: C++ Coding Programming Conventions"

[8] Nocedal J. & Wright, S. (1999). Numerical optimization. New York: Springer.

[9] Cabrera-Gámez, J., Domínguez-Brito, A. C., and Hernández-Sosa, D. (2000). Coolbot: A component-oriented programming framework for robotics. Dagstuhl Seminar 00421, Modelling of Sensor-Based Intelligent Robot Systems, To appear in Springer Lecture Notes in Computer Science in summer 2001. Centro de Tecnología de los Sistemas y de la Inteligencia Artificial (CeTSIA), University of Las Palmas de Gran Canaria, Edificio de Informática y Matemáticas, Campus Universitario de Tafira, 35017 Las Palmas, SPAIN.

[10] Domínguez-Brito, A. C., Andersson, M., and Christensen, H. I. (2000a). A Software Architecture for Programming Robotic Systems based on the Discrete Event System Paradigm. Technical Report CVAP 244, Centre for Autonomous Systems, KTH - Royal Institute of Technology), S-100 44 Stockholm, Sweden.

[11] Domínguez-Brito, A. C. (2003). CoolBOT: a Component- Oriented Programming Framework for Robotics. Tesis Doctoral. Universidad de Las Palmas de Gran Canaria.

[12] Domínguez-Brito, A. C., Hernández-Tejera, F. M., and Cabrera-Gámez, J. (2000b). A Control Architecture for Active Vision Systems. Frontiers in Artificial Intelligence and Applications: Pattern Recognition and Applications, M.I. Torres and A. Sanfeliu (eds.), pp. 144-153, IOS Press, Amsterdam.

[13] Domínguez-Brito, A. C., Hernández-Sosa, D., and Cabrera-Gámez, J. (2002). Programming with Components in Robotics. Waf 2002 - III Workshop Hispano-Luso en Agentes Físicos, Murcia.

[14] Steenstrup, M., Arbib, M. A., and Manes, E. G. (1983). Port automata and the algebra of concurrent processes. Journal of Computer and System Sciences, 27:29–50.

[15] Stewart, D. B. and Volpe, R. A. and Khosla, P. Design of Dynamically Reconfigurable Real-Time Software Using Port-Based Objects. IEEE Transactions on Software Engineering, December, 1997, vol 23,num 12, pag 759-776

[16] Stevens, Richard. UNIX Network Programming, Volume 2, Second Edition: Inter- process Communications. Prentice Hall, 1999.

[17] "MATLAB User's Guide", MathWorks, 2.011.

[18] Eigen: API matricial library project in C++ . http://eigen.tuxfamily.org/index.php?title=Main_Page

[19] Radish The Robotics Data Set Repository. http://radish.sourceforge.net/

[20] Ciclos de vida. Paradigmas del ciclo de vida, Wikipedia.

[21] Roger S. Pressman: "Ingeniería del software: un enfoque práctico". McGraw-Hill,, México : (2006) - (6ª ed.) 970-10-5473-3

[22] Modelos de ciclo de vida por la UNED, Modelos de ciclo de vida escrito por la UNED http://www.ia.uned.es/ia/asignaturas/adms/GuiaDidADMS/node10.html

[23] The GTK+ Project. Librería para la creación de interfaces de usuarios. http://www.gtk.org/

[24] CoolBOT The CoolBOT Project by Antonio Carlos Domínguez Brito. http://www.coolbotproject.org/

[25] Player/stage: The Player Project. http://playerstage.sourceforge.net/

[26] Mathworks: página oficial del software Matlab http://www.mathworks.com/

[27] Kdevelop: KDE development environment. http://kdevelop.org/

[28] CVS: Concurrent version system. http://www.cvshome.org/

[29] GIT: Fast version control system. http://git-scm.com/

[30] LyX: The Document Processor. http://www.lyx.org/

[31] Inkscape: Open source scalable vector graphics editor. http://inkscape.org/

[32] Gimp: GNU image manipulation program. http://www.gimp.org/

[33] Gcc: the GNU Compiler Collection. http://gcc.gnu.org/

[34] MiKTeX: Typesetting beatiful documents. http://miktex.org/

[35] Cmake Una plataforma para la compilación sencilla de proyectos desarrollada por Kitware. http://www.cmake.org/

[36] GDB: The GNU Project Debugger. http://www.gnu.org/software/gdb/

[37] CGDB: The curses debugger. http://cgdb.sourceforge.net/.

[38] DDD: Data Display Debugger. http://www.gnu.org/software/ddd/.

[39] SmartDraw: Comunicate visually. http://www.smartdraw.com/.

[40] Wireshark: Network Protocol Analyzer. http://www.wireshark.org/.

[41] Pkg-config Repositorio del se encuentra almacenado la aplicación. http://www.freedesktop.org/wiki/Software/pkg-config