ESCUELA POLITÉCNICA NACIONAL

FACULTAD DE INGENIERÍA ELÉCTRICA Y ELECTRÓNICA

PLANEACIÓN Y SEGUIMIENTO DE TRAYECTORIAS PARA UN ROBOT MÓVIL

PROYECTO PREVIO A LA OBTENCIÓN DEL TÍTULO DE INGENIERO EN ELECTRÓNICA Y CONTROL

YANDÚN TORRES ARACELY INÉS
ara19preciosa@gmail.com

DIRECTOR: NELSON SOTOMAYOR, MSc.
nelson.sotomayor@epn.edu.ec

Quito, Abril 2011

DECLARACIÓN

Yo, Aracely Inés Yandún Torres, declaro bajo juramento que el trabajo aquí descrito es de mi autoría; que no ha sido previamente presentado para ningún grado o calificación profesional; y, que he consultado las referencias bibliográficas que se incluyen en este documento.

A través de la presente declaración cedo mis derechos de propiedad intelectual correspondientes a este trabajo, a la Escuela Politécnica Nacional, según lo establecido por la Ley de Propiedad Intelectual, por su Reglamento y por la normatividad institucional vigente.

_____

Aracely Inés Yandún Torres

CERTIFICACIÓN

Certifico que el presente trabajo fue desarrollado por Aracely Inés Yandún Torres, bajo mi supervisión.

_____

Nelson Sotomayor MSc.
DIRECTOR DEL PROYECTO

AGRADECIMIENTO

Quiero agradecer a Dios por siempre iluminar mi vida incluso cuando me parece todo perdido, por darme a las dos personas más importantes en mi vida: mis padres, porque nunca me dejaron caer y estuvieron conmigo en todo momento, por ellos soy lo que soy y gracias a su ejemplo, educación y amor he logrado obtener grandes objetivos en mi vida. Papi, Mami gracias.

Agradecer también a todos mis maestros, por haberme brindado sus conocimientos y prepararme para ser una gran profesional, en especial a mi Director Nelson Sotomayor, Inge de verdad muchas gracias por todo.

A mis amigos, con quienes compartí los mejores y los peores momentos de la Poli y estuvieron siempre a mi lado. Gracias muchach@s.

En fin, a todos aquellos que de una u otra forma siempre me apoyaron, me ayudaron, se preocuparon y nunca dejaron de creer en mí, mil gracias. Siempre los llevo en mi corazón.

DEDICATORIA

A quienes amo

CONTENIDO

CAPÍTULO 2

RESUMEN

Dentro de la disciplina de la robótica, una de las metas más importantes es la creación de robots autónomos, mismos que se espera adquieran una descripción de alto nivel de la tarea a realizar y ésta sea ejecutada sin la intervención de los seres humanos.

La robótica móvil constituye una valiosa herramienta para el desarrollo de tecnologías para la creación de robots de navegación autónoma, permitiendo de esta manera explorar entornos inaccesibles a los seres humanos ya sea por su lejanía, costo, peligro o sencillamente por tratarse de tareas desagradables, repetitivas o laboriosas.

Para obtener los resultados deseados en cuanto a las acciones del robot, es importante tomar en cuenta muchos problemas a resolver, siendo uno de ellos la planificación de movimientos. Con el objetivo de que el robot tenga la capacidad necesaria para ejecutar ciertas tareas a través de su movimiento es importante tener un conocimiento previo del espacio de trabajo, por ejemplo, para planear una ruta libre de colisión es importante conocer la ubicación de los obstáculos, mismos que pueden ser conocidos u obtenidos a través de sensores.

Para este trabajo se consideran entornos estructurados fijos y conocidos, es decir estáticos, el robot es el único elemento en movimiento dentro del mapa de entorno, por lo que el problema de planificación de movimientos se simplifica a que dadas una posición inicial y final, se desea generar una trayectoria óptima libre de obstáculos misma a ser seguida por el robot móvil.

En la actualidad existe un gran número de métodos para la planeación y generación de trayectorias, desde algoritmos simples hasta lógicas de programación más complejas, por ende es de suma importancia seleccionar un método adecuado de acuerdo a las características y requerimientos de la aplicación.

En el presente proyecto se hace un estudio de los diferentes métodos para la planeación de trayectorias, haciendo énfasis en uno de ellos para ser desarrollado e implementado en el robot móvil Robotino® de Festo.

PRESENTACIÓN

En el presente trabajo se aborda el estudio de los métodos para la planeación y generación de trayectorias en robots móviles, se desarrolla uno de estos métodos, se genera una trayectoria óptima a seguir y se la descarga en un robot móvil para la comprobación de los algoritmos implementados.

En el Capítulo 1 se realiza una introducción al marco teórico necesario para el desarrollo de este proyecto, se describe a la robótica móvil, se detallan los métodos de planeación de trayectorias más importantes, se aborda más a fondo a los diagramas de Voronoi, método a ser implementado en el robot móvil y se describe la teoría necesaria para la programación y funcionamiento de la plataforma móvil Robotino® de Festo.

En el Capítulo 2 se describe el desarrollo del software implementado en LabVIEW, una descripción de la programación de las herramientas adicionales utilizadas (tool kits), las diferentes funciones y algoritmos, así como todos los subVI que conforman el programa para que en conjunto permitan el ingreso del mapa de entorno, la generación del diagrama de Voronoi, la obtención de la trayectoria y la descarga de la misma en el Robotino®.

En el Capítulo 3 se presentan las pruebas realizadas para evaluar el correcto funcionamiento del proyecto. Se explica el funcionamiento de la HMI y se realizan 3 pruebas con diferentes formas de trayectorias y distintos mapas de entorno, se muestran resultados a través del cálculo de errores de posición, evaluando los puntos teóricos de la trayectoria generada en el LabVIEW en comparación con los puntos obtenidos a través del seguimiento de la ruta por el Robotino®.

En el Capítulo 4 se muestran las conclusiones y recomendaciones obtenidas durante el transcurso del desarrollo del proyecto.

CAPÍTULO 1

FUNDAMENTOS BÁSICOS

En el presente trabajo se desarrolla un breve estudio sobre la planeación y seguimiento de trayectorias para robots móviles. Posteriormente se desarrolla en LabVIEW uno de los métodos de planeación denominado Diagramas de Voronoi para obtener la trayectoria más óptima acorde a un mapa de entorno, puntos de salida y llegada conocidos. Por último, se comprueba el funcionamiento de los algoritmos implementados descargando el programa en el robot móvil Robotino® de Festo vía wireless.

1.1 INTRODUCCIÓN A LA ROBÓTICA

El origen de la robótica data desde hace cientos de años atrás, y se encuentra ligado a la necesidad de las personas de crear “dispositivos” o “artefactos” a su semejanza con el objetivo de alivianarles trabajo y mejorar el rendimiento de tareas específicas, repetitivas o difíciles para ser realizadas por el ser humano.

En Octubre de 1942 el científico ruso Isaac Asimov (1920 – 1992) publica en la revista “Galaxy Science Fiction” una historia titulada: “The Caves of Steel” en la que por primera vez anuncia sus tres leyes de la robótica [1]:

1. Un robot no puede perjudicar a un ser humano, ni con su inacción permitir que un ser humano sufra daño.
2. Un robot ha de obedecer las órdenes que le son dadas por un ser humano, excepto si tales órdenes entran en conflicto con la primera ley.
3. Un robot debe proteger su propia existencia siempre y cuando ésta protección no entre en conflicto con la primera o segunda ley.

La robótica es una disciplina con sus propios fundamentos y leyes, se puede decir que básicamente se ocupa de todo lo concerniente a los robots, por ende se

encuentra incluido el control de motores, sistemas de comunicación, mecanismos automáticos, diferentes tipos de sensores, etc.

1.2 DEFINICIÓN DE ROBOT

Debido a la complejidad de definir a un robot, se ha dado lugar a una variedad de definiciones y opiniones.

Según el Instituto Norteamericano del Robot y de la ISO 8373: “Un robot es un manipulador reprogramable, multifuncional, controlado automáticamente, que puede estar fijo en un sitio o moverse, y que está diseñado para mover materiales, piezas, herramientas o dispositivos especiales, por medio de movimientos variables programados para la realización de diversas tareas o trabajos” [2].

Acorde con el Diccionario de la Real Academia Española: “Máquina o ingenio electrónico programable, capaz de manipular objetos y realizar operaciones antes reservadas solo a personas” [1].

Según la Enciclopedia Británica: “Máquina operada automáticamente que sustituye el esfuerzo de los humanos, aunque no tiene por qué tener apariencia humana o desarrollar sus actividades a manera de humanos” [1].

La existencia de un gran conjunto de sistemas definidos como robots, hace que las definiciones anteriores sean insuficientes para definir el concepto de un robot, para diferenciar y/o clasificarlos se les ha añadido un adjetivo para describir con más detalle su aplicación, por ejemplo: robots móviles, robots autónomos, robots espaciales, etc.

Se pueden identificar 2 grandes clases de robots: industriales y de servicio. Los primeros se los considera como manipuladores controlados automáticamente que se los utiliza en aplicaciones industriales. Los robots de servicio por su parte pueden funcionar parcial o totalmente de manera autónoma y su función es cumplir tareas útiles para el bienestar de los seres humanos y equipos, excluyendo tareas de operación.

1.3 ROBÓTICA MÓVIL

1.3.1 INTRODUCCIÓN

Con el propósito de aumentar la movilidad del robot y de esta manera su espacio de trabajo, se tiene la necesidad de utilizar la robótica móvil. En este caso el robot hace uso de su sistema locomotor para interactuar con el medio que lo rodea [3].

Los robots móviles siguen su camino por telemando o guiándose por la información recibida de su entorno a través de sensores [4].

1.3.2 CLASIFICACIÓN DE LOS ROBOTS MÓVILES

Dentro de los robots móviles, se encuentra una primera división en robots autónomos y no autónomos. Los primeros portan todo el software y hardware de control sobre la estructura mecánica. Esto les da un rango de alcance limitado únicamente por la duración de las fuentes de alimentación que utilicen, pero encarece y produce una mayor complejidad en el sistema. Por otra parte, un robot no autónomo es gobernado por un ordenador externo al que se comunica a través de un bus de señales de datos y control. Las fuentes de alimentación son así mismo externas [5].

Según el medio de locomoción que utilicen los robots móviles se puede establecer una segunda clasificación: Robots con patas, Robots con ruedas, Robots oruga, etc.

1.3.2.1 Robots con patas

Los robots con patas permiten desplazamientos más eficientes sobre terrenos de cualquier tipo (rugosos, con obstáculos o desniveles,…..), además de ofrecer un control de estabilidad más completo [5].

Dentro de este grupo de robots, acorde al número de patas que poseen se los puede clasificar en:

- Robots de una sola pata
- Robots bípedos
- Robots cuadrúpedos

- Robots hexápodos

Figura 1.1 Robot hexápodo, tomado de [6]

1.3.2.2 Robots de ruedas

Los robots de ruedas se podría decir son los más populares ya que son más sencillos y más fáciles de construir, además la carga que pueden transportar es relativamente mayor. Por lo general, tanto los robots basados en cadenas como en patas se pueden considerar más complicados y pesados que los robots de ruedas para una misma carga útil. A esto se puede añadir el que se pueden transformar vehículos de ruedas de radio control para usarlos como bases de robots.

Figura 1.2 Robot de ruedas, tomado de [7]

La principal desventaja de las ruedas es su empleo en terreno irregular, en el que se comportan bastante mal. Normalmente un vehículo de ruedas podrá sobrepasar un obstáculo que tenga una altura no superior al radio de sus ruedas, entonces una solución es utilizar ruedas mayores que los posibles obstáculos a superar; sin embargo, esta solución, a veces, puede no ser práctica [8].

1.3.2.2.1 Diseño de Ruedas Diferencial [8]

Tanto desde el punto de vista de la programación como de la construcción, el diseño diferencial es uno de los menos complicados sistemas de locomoción. El robot puede ir recto, girar sobre sí mismo y trazar curvas.

Un problema importante es cómo resolver el equilibrio del robot, hay que buscarle un apoyo adicional a las dos ruedas ya existentes, esto se consigue mediante una o dos ruedas de apoyo añadidas en un diseño triangular o romboidal. El diseño triangular puede no ser suficiente dependiendo de la distribución de pesos del robot, y el romboidal puede provocar inadaptación al terreno si éste es irregular lo que puede exigir alguna clase de suspensión.

Otra consideración a hacer en este diseño es cómo conseguir que el robot se mueva recto, para que el robot se mueva en línea recta sus ruedas tienen que girar a la misma velocidad.

Cuando los motores encuentran diferentes resistencias (una rueda sobre moqueta y la otra sobre terrazo) las velocidades de los motores varían y el robot girará incluso aún cuando se le haya ajustado inicialmente para que vaya recto. Esto quiere decir que la velocidad debe ser controlada dinámicamente, o sea, debe existir un medio de monitorizar y cambiar la velocidad del motor mientras el robot avanza. De esta manera la simplicidad del diseño queda minimizada por la complejidad del sistema de control de la velocidad.

1.3.2.2.2 Diseño de ruedas sincronizado [8]

En este diseño todas las ruedas (generalmente tres) son tanto de dirección como motrices, las ruedas están enclavadas de tal forma que siempre apuntan en la misma dirección. Para cambiar de dirección el robot gira simultáneamente todas sus ruedas alrededor de un eje vertical, de modo que la dirección del robot

cambia, pero su chasis sigue apuntando en la misma dirección que tenía. Si el robot tiene una parte delantera (es asimétrico) presumiblemente donde se concentran sus sensores, se tendrá que arbitrar un procedimiento para que su cuerpo se oriente en la misma dirección que sus ruedas. El diseño sincronizado supera muchas de las dificultades que plantean el diseño diferencial, en triciclo y de coche, pero a costa de una mayor complejidad mecánica.

1.3.2.2.3 Diseño de ruedas de triciclo y coche [8]

El diseño de coche con sus cuatro ruedas con suspensión proporciona una buena estabilidad, el diseño en triciclo tiene unas prestaciones similares con la ventaja de ser mecánicamente más simple ya que el coche necesita alguna unión entre las ruedas direccionables. En general en estos dos diseños las ruedas direccionables no son motrices, y no es necesario controlar la velocidad de las ruedas para que el robot se mantenga recto.

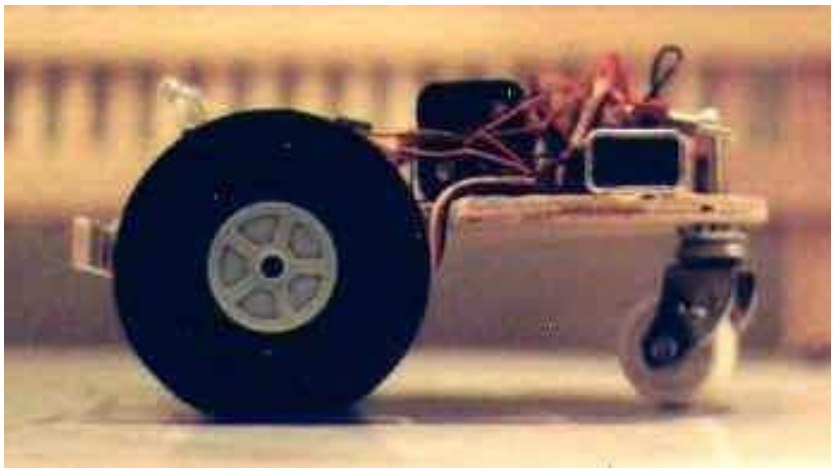


Figura 1.3 Robot triciclo, tomado de [8]

1.3.2.2.4 Diseño de ruedas omnidireccional

La cinemática, se centra en el estudio del movimiento del robot en función de su geometría. Entre las aplicaciones inmediatas se encuentran la posibilidad de utilizarlo como modelo matemático de partida para el diseño del controlador, la simulación del comportamiento cinemático del vehículo, o para establecer las ecuaciones de los cálculos odométricos [9].

El comportamiento cinemático se establece en el principio de que las ruedas en contacto con el suelo se comportan como una articulación planar de tres grados de libertad, tal y como aparece en la Figura 1.4 [9].

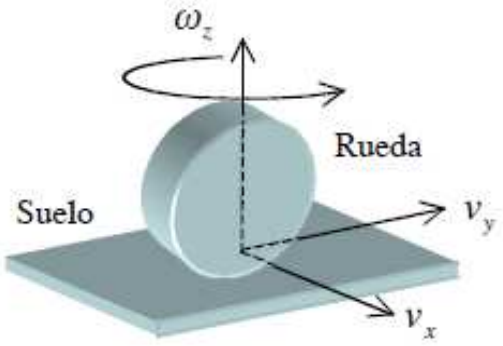


Figura 1.4 Rueda en contacto con el suelo, tomado de [9]

Al suponerse la rueda como un elemento rígido, ésta entra en contacto con el suelo en un solo punto, que sirve de origen al sistema de referencias solidario dibujado en la Figura 1.4. Se utiliza para definir los tres grados de libertad antes mencionados. La dirección V_y determina el sentido normal de avance de la rueda; el eje V_x indica los deslizamientos laterales, y W_z la velocidad rotacional que se produce cuando el vehículo realiza un giro. En el caso de una rueda convencional, la componente V_x, se supone siempre nula, sin embargo, existen ruedas diseñadas para eliminar la mencionada restricción. Este es el caso de la presentada en el esquema de la Figura 1.5 [9].

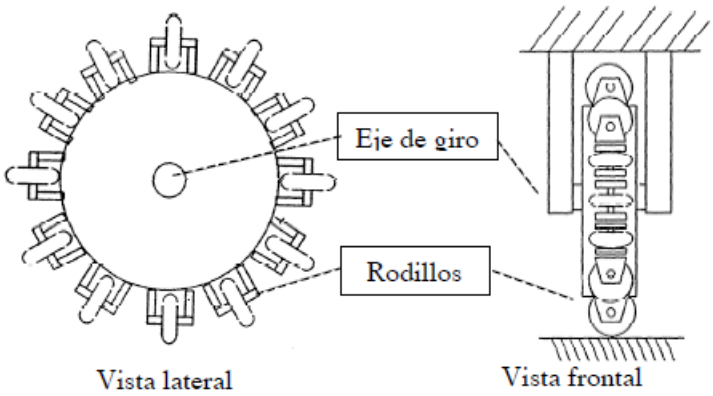


Figura 1.5 Rueda omnidireccional, tomado de [9]

La rueda omnidireccional se define como una rueda estándar a la cual se la ha dotado de una corona de rodillos, cuyos ejes de giro resultan perpendiculares a la

dirección normal de avance. De este modo, al aplicarle una fuerza lateral, los rodillos giran sobre sí mismo y permite que la componente V_x no sea nula, y por tanto, se elimina la restricción de no-holomicidad [9], siendo la holomicidad la capacidad de un robot para poder moverse en cualquier sentido y dirección de manera instantánea, por ejemplo, un robot con 2 ruedas es no-holonómico ya que no puede moverse hacia la izquierda o la derecha, siempre lo hace hacia delante en la dirección definida por la velocidad de sus ruedas.

Un robot omnidireccional tiene movilidad completa (3 ruedas), por lo que puede moverse en cada instante en cualquier dirección sin necesitar reorientación, de ahí que se le llame vehículo omnidireccional [10].

1.3.2.3 Robots de cadenas

Para robots que vayan a funcionar en un entorno natural las cadenas son una opción muy buena porque las cadenas permiten al robot superar obstáculos relativamente mayores y son menos susceptibles que las ruedas de sufrir daños por el entorno, como piedras o arena. El principal inconveniente de las cadenas es su ineficacia, puesto que se produce deslizamiento sobre el terreno al avanzar y al girar. Si la navegación se basa en el conocimiento del punto en que se encuentra el robot y el cálculo de posiciones futuras sin error, entonces las cadenas acumulan tal cantidad de error que hace inviable la navegación por este sistema. En mayor o menor medida cualquiera de los sistemas de locomoción adolece de este problema [8].

Figura 1.6 Robot de cadenas, tomado de [8]

A inicios de este capítulo se menciona el uso de la plataforma educativa Robotino® de Festo para descargar una trayectoria desarrollada en LabVIEW, por lo que en el siguiente subcapítulo se describe más a detalle la funcionalidad y las características de este robot.

1.4 ROBOTINO®

1.4.1 DISEÑO Y FUNCIÓN

Robotino® es un sistema de robot móvil de alta calidad, plenamente funcional con accionamiento omnidireccional. Cuenta con 3 unidades de accionamiento que permiten movimientos en todas las direcciones: adelante, atrás y lateralmente. Adicional a esto, el robot puede girar sobre un punto.

Además Robotino® se encuentra equipado con una webcam que permite visualizar una imagen de cámara en vivo y una serie de sensores analógicos para mediciones de distancia, es el caso de los sensores binarios para protección de colisiones y sensores digitales para detectar la velocidad real. Adicionalmente puede conectarse actuadores y sensores adicionales en el Robotino® a través de una interfaz de entradas/salidas.

Robotino® consiste en un PC embebido con una tarjeta compact flash, en esta se han instalado el sistema operativo Linux así como algunas aplicaciones de demostración. Estas aplicaciones pueden ejecutarse directamente desde el panel de control del Robotino®.

Para programar el Robotino® en un PC se utiliza el software Robotino®View, este software es capaz de transmitir señales de manera inalámbrica al controlador del motor, así como visualizar, cambiar y evaluar valores de los sensores. La programación se la puede hacer incluso durante el funcionamiento real. Otras alternativas de programación del Robotino® son APIs Linux y C++.

Debido a las altas prestaciones que aportan al sistema la necesaria “inteligencia”, a Robotino® se lo considera autónomo.

1.4.1.1 Datos Técnicos

Tabla 1.1 Datos técnicos, tomado de [11]

Parámetro	Valor
Alimentación de tensión	24 V DC, 4.5 A
Entradas digitales	8
Salidas digitales	8
Entradas analógicas	8 (0 – 10 V)
Salidas por relé	2

1.4.1.2 Chasis y Puente de Mando

El chasis consiste en una plataforma de acero inoxidable soldada con láser

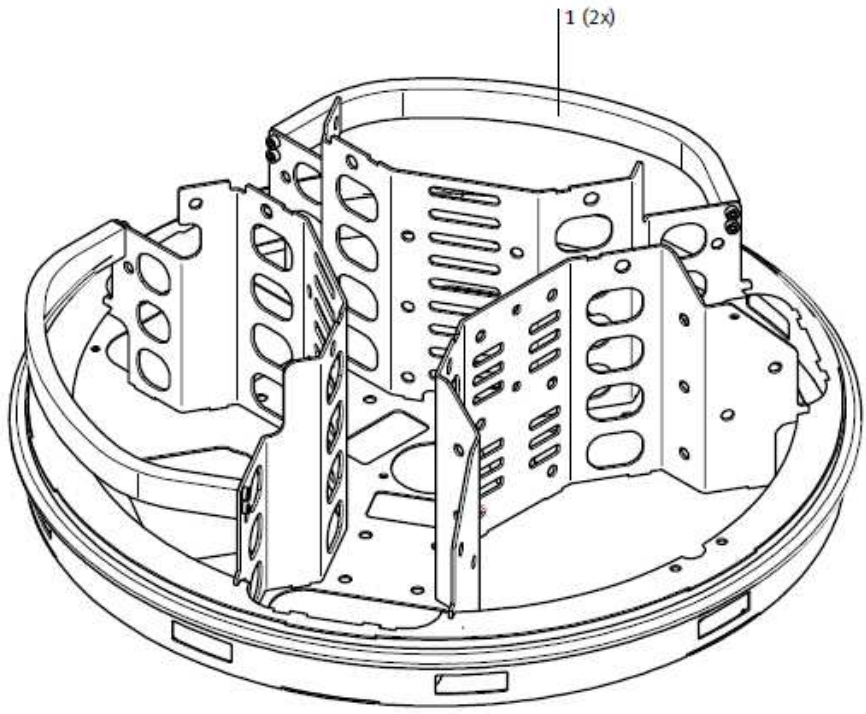


Figura 1.7 Chasis y puente de mando, Asa (1), tomado de [11]

En el chasis se encuentran montadas las baterías recargables, las unidades de accionamiento y la cámara, aquí también se hallan situados los sensores de medición de distancia y el sensor anticolisión.

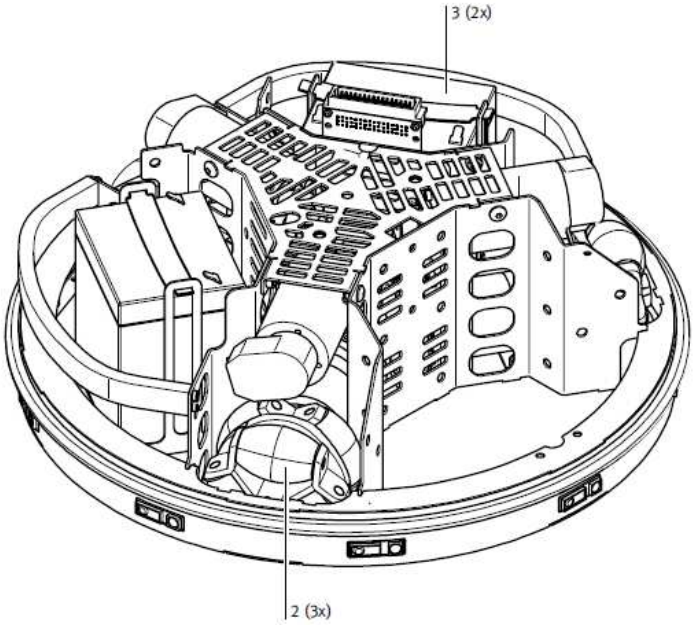


Figura 1.8 Unidades de accionamiento (2) y batería (3), tomado de [11]

El puente de mando está conectado a los demás módulos del sistema por medio de un conector, en el puente de mando se encuentran los componentes más sensibles del sistema, tales como el controlador, el módulo de E/S y las interfaces.

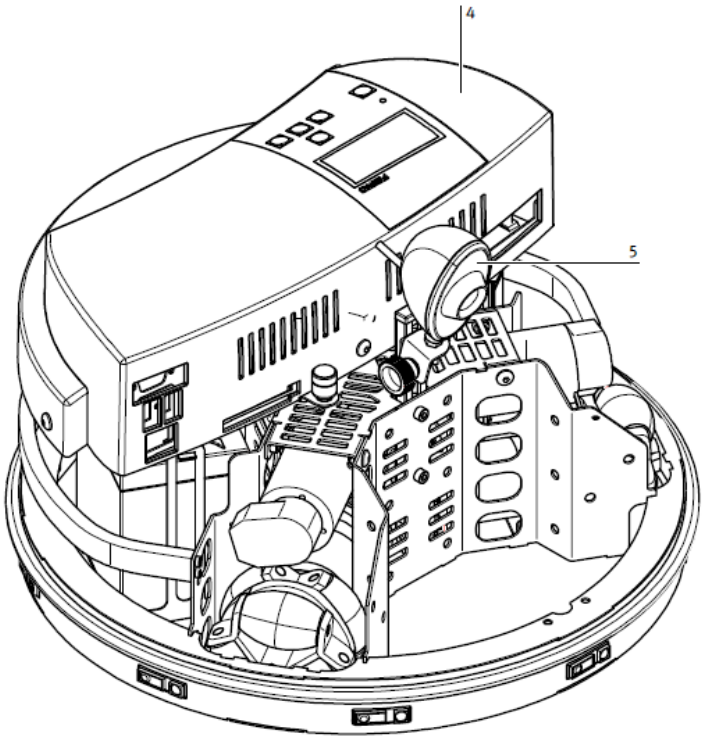


Figura 1.9 Puente de mando (4) y cámara (5), tomado de [11]

Además el chasis ofrece un espacio adicional y opciones de montaje para añadir sensores y/o actuadores.

1.4.1.3 Módulo de la Unidad de Accionamiento

Robotino® es accionado por 3 unidades de accionamiento omnidireccionales independientes. Cada una se encuentra montada formando un ángulo de 120º entre sí.

Cada unidad de accionamiento consta de los siguientes componentes:

- Motor DC
- Reductor con una relación de reducción de 16:1
- Rodillos omnidireccionales
- Correa dentada
- Encóder incremental

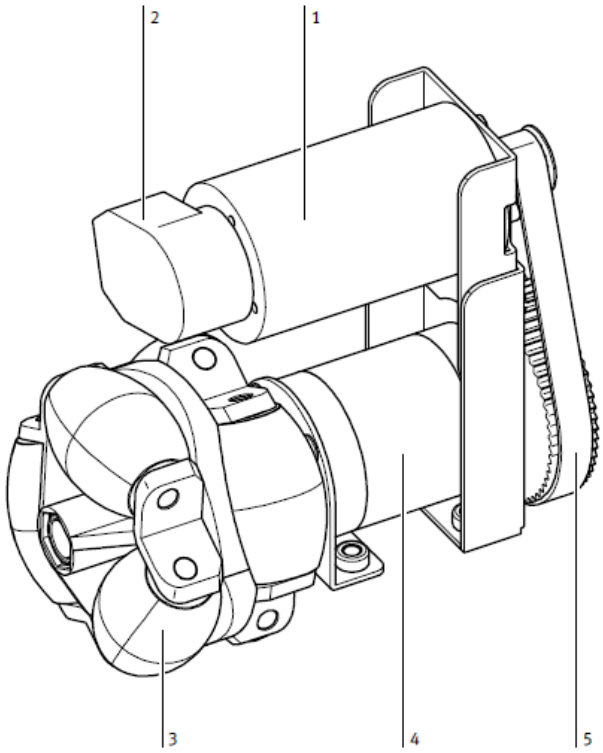


Figura 1.10 Motor (1), Encoder incremental (2), Rodillos omnidireccionales (3), Reductor (4), Correa dentada (5), tomado de [11]

Para asegurar el correcto posicionado de las unidades de accionamiento entre sí, todos los componentes individuales están fijados a la brida de montaje en la parte posterior, junto con la brida frontal, la unidad de accionamiento se encuentra sujeta al chasis con tornillos. Se puede comparar a través del encoder incremental la velocidad real del motor con la velocidad deseada, y puede regularse esta velocidad real con un regulador PID a través de la placa de circuito de E/S.

Tabla1.2 Datos del rodillo omnidireccional, tomado de [11]

Rodillo omnidireccional, accionado (ARG 80)	
Diámetro Ø	80 mm
Máxima capacidad de carga	40 kg

El rodillo omnidireccional es puesto en movimiento en una determinada dirección por medio de su eje de accionamiento y también es capaz de desplazarse en cualquier dirección si se ve forzado por otros accionamientos en direcciones diferentes. Como resultado de la interacción con las otras dos unidades de accionamiento, es posible obtener un recorrido en una dirección que difiere de la dirección de cada uno de los respectivos accionamientos.

Tabla 1.3 Datos de rendimiento del motor, tomado de [11]

Motor DC (GR 42x25)	**Unidad de medida**	
Tensión nominal	V DC	24
Velocidad nominal	RPM	3600
Par nominal	Ncm	3,8
Intensidad nominal	A	0,9
Par de arranque	Ncm	20
Intensidad de arranque	A	4
Velocidad sin carga	RPM	4200
Intensidad sin carga	A	0,17
Intensidad de desmagnetización	A	6,5
Momento de inercia de la masa	gcm^2	71
Peso del motor	gr.	390

Tabla 1.4 Datos del reductor, tomado de [11]

Reductor planetario (PLG 42 S)	
De una sola etapa, Nm:	3,5
De una sola etapa, i:	4 :1 – 8 :1
2-etapas, Nm:	6
2-etapas, i:	16 :1 – 64 :1
3-etapas, Nm:	14
3-etapas, i:	100 :1 – 512 :1

1.4.1.4 El Módulo de Cámara

El sistema Robotino® cuenta con un módulo de cámara, ajustable en altura e inclinación. Con la ayuda del software Robotino®View, la cámara permite visualizar imágenes en directo, así como este programa ofrece opciones de procesamiento de imágenes, con lo que se puede evaluar imágenes para el controlador Robotino. A través de un segmentador se localiza superficies del mismo color, se puede determinar la posición y tamaño de cualquier segmento.

Tabla 1.5 Especificaciones técnicas de la cámara, tomado de [11]

Especificaciones técnicas	
Sensor de imágenes	Color VGA CMOS
Profundidad de color	24 Bit Color verdadero
Conexión a PC	USB 1.1
Resoluciones de vídeo	160 x 120, 30 fps (SQCGA) 176 x 144, 30 fps (QCIF) 320 x 240, 30 fps (QVGA) 352 x 288, 30 fps (CIF) 640 x 480, 15 fps (VGA)
Resoluciones a imagen parada	160 x 120 (SQCGA) 176 x 144 (QCIF) 320 x 240 (QVGA) 352 x 288 (CIF) 640 x 480 (VGA) 1024 x 768 (SVGA)
Formato de captura a imagen parada	BMP, JPG

Se recomienda conectar la cámara de ser posible en el puerto USB en el lado derecho para de esta manera reducir el riesgo de destrucción del cable.

1.4.1.5 La Unidad de Control [11]

Esta unidad puede configurarse de forma flexible utilizando diversos módulos plug-in.

El controlador del Robotino® consta de 3 componentes:

- Procesador PC104, compatible con MOPSlcdVE, 300 MHz, y sistema operativo Linux con kernel en el tiempo real, SDRAM 128 MB
- Tarjeta Compact flash (256MB) con API C++ para controlar el Robotino®
- Punto de acceso LAN inalámbrico

La unidad de control está equipada con las siguientes interfaces:

Ethernet, 2ea. USB y VGA. Estos se utilizan para conectar un teclado, un ratón y una pantalla. Con ello puede accederse al sistema operativo y a la librería C++ sin un PC, si no es posible o no se desea utilizar la conexión WLAN. Con la versión básica no puede utilizarse la conexión Ethernet.

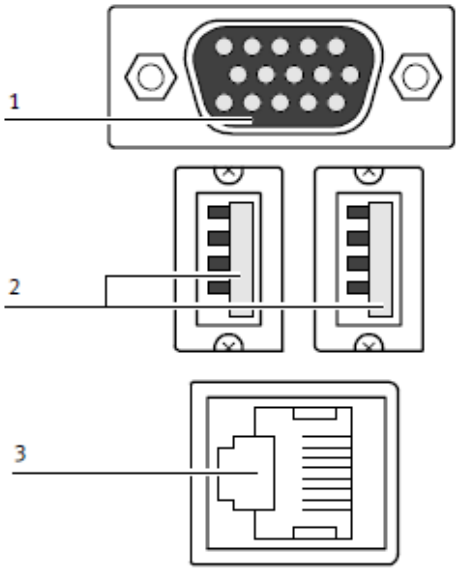


Figura 1.11 Zócalo conector VGA (1), Puertos USB 1y2 (2), Interface Ethernet (3), tomado de [11]

1.4.1.6 Módulo Tarjeta de Circuito de E/S

Esta tarjeta de E/S establece la comunicación entre la unidad de control y los sensores, la unidad de accionamiento y el interface E/S incluidos con el Robotino®.

Cada uno de los motores de las unidades de accionamiento individuales es controlado por un regulador PID. Cada motor puede ser regulado individualmente.

Las señales del encoder de pasos, la de todos los sensores y actuadores instalados que están conectados al interface de E/S son transferidas a la unidad de control o a los actuadores adicionales [11].

1.4.1.7 Fuente de Alimentación/Cargador de Batería

La alimentación eléctrica es suministrada por dos baterías recargables de 12 V con una capacidad de 4 Ah. Ambas baterías recargables están montadas en el chasis.
Robotino® se suministra con 2 baterías adicionales y un cargador de baterías. Así, mientras dos baterías se hallan en funcionamiento, las otras dos pueden estar en proceso de recarga [11].

1.4.1.8 Sensores

En el Robotino® se han integrado sensores para la medición de distancias a objetos y para detectar la velocidad del motor. Se dispone de los siguientes sensores:

1.4.1.8.1 Sensores de medición de distancia por infrarrojos

Se cuenta con 9 de estos sensores, se hallan montados en el chasis formando un ángulo de 40º entre sí. Son capaces de medir distancias con precisión o relativas a objetos, con valores entre 4 y 30 cm. Cada uno de estos sensores puede ser interrogado individualmente por medio de la placa de E/S.

La conexión del sensor es sencilla ya que incluye tan sólo una señal de salida analógica y la alimentación.

1.4.1.8.2 Encoder incremental

El encoder incremental mide la velocidad real del motor en RPM. Si esta velocidad real difiere de la velocidad de consigna se puede ajustar la misma al valor deseado por medio de un regulados PID.

Los parámetros del PID están configurados con la ayuda del software Robotino®View.

1.4.1.8.3 Sensor anticolisión

Consta de una banda de detección fijada alrededor de un aro que circunda el chasis. Una cámara de conmutación se halla situada dentro de un perfil de plástico.

Dos superficies conductoras se hallan dispuestas dentro de la cámara, manteniendo una determinada distancia entre sí, estas superficies entran en contacto cuando se aplica una mínima presión a la banda. Con ello, una señal perfectamente reconocible es transmitida a la unidad de control [11].

1.4.1.8.4 Sensor de proximidad inductivo analógico

Este sensor se suministra como un componente adicional y sirve para detectar objetos metálicos en el piso y para control filoguiado.

Dependiendo de si se halla en el medio o borde de una tira metálica el sensor lee señales de distinta intensidad.

1.4.1.8.5 Sensores de reflexión directa

Se puede realizar el seguimiento de una ruta a través de los dos sensores de reflexión directa (luz difusa) incluidos.

Los cables flexibles de fibra óptica se conectan a una unidad óptica que funciona con luz roja visible, la luz reflejada se detecta teniendo en cuenta que diferentes superficies y colores producen diferentes grados de reflexión

Figura 1.12 Encoder incremental (1), Banda de impacto, sensor aticolisión (2), Sensores de medición de distancia (3), tomado de [11]

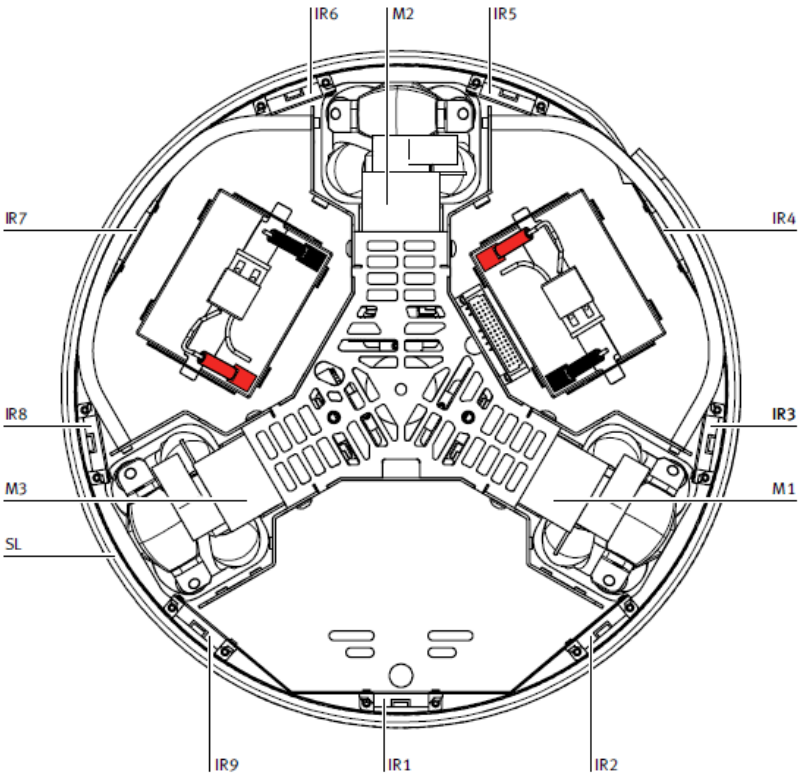


Figura 1.13 Sensores de medición de distancias IR1 – IR9, Motores M1 – M3, Banda de colisión/Sensor anti-colisión SL, tomado de [11]

1.4.1.9 Teclado de membrana y display

En la parte superior del cuerpo se halla un teclado de membrana y un display, por medio de los cuales pueden seleccionarse diversas opciones, solicitar información y ejecutar los programas que se incluyen.

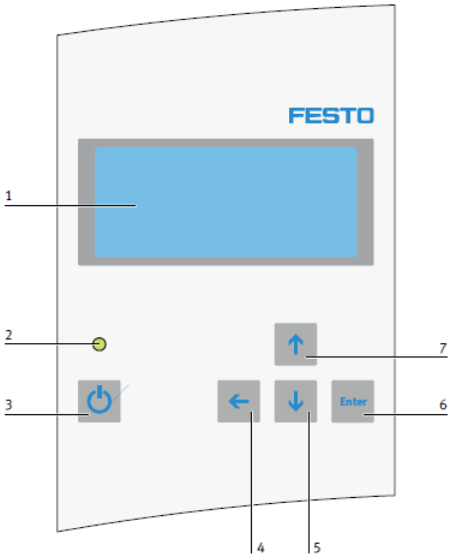


Figura 1.14 Display (1), Led (2), Marcha/Paro (3), Subir un nivel el menú (4), Deslizar abajo una selección (5), Aceptar selección (6), Deslizar arriba una selección (7), tomado de [11]

1.4.1.10 Punto de acceso LAN inalámbrico [11]

El punto de acceso LAN inalámbrico es un componente que permite la comunicación con el robot por medio de una dirección en la red.

- El punto de acceso se caracteriza por su bajo consumo de corriente. Es posible una alimentación a través del puerto USB.
- El punto de acceso cumple con los siguientes estándares: IEEE 802.11g y 802.11b.
- Permite velocidades de transmisión de hasta 54 Mb por segundo para 802.11g y 11 Mb por segundo para 802.11b con un amplio alcance de las transmisiones (hasta 100 m dentro de edificios)
- Permite establecer una red segura con encriptación WEP y función WPA-PSK
- Es rápida y simple de configurar a través de la utilidad de gestión de la web

1.4.1.11 La tarjeta compact flash

La unidad de control está equipada con una ranura en la cual se ha insertado una tarjeta de PC. Esta tarjeta de PC contiene el sistema operativo, las librerías de funciones y los programas incluidos.

Las actualizaciones pueden instalarse fácilmente con la simple sustitución de la tarjeta PC. La ranura para la tarjeta PC está situada a la derecha de los interfaces de la unidad de control.

1.4.1.12 El interface E/S

El interface E/S permite conectar sensores y actuadores adicionales. Estos se conectan por medio de un conector incluido [11].

- 8 entradas analógicas (0 a 10V) (AIN0 hasta AIN7)
- 8 entradas digitales (DI0 hasta DI7)
- 8 salidas digitales (DO0 hasta DO7)
- 2 relés para actuadores adicionales (REL0 y REL1). Los contactos de los relés pueden utilizarse como NA, NC o conmutados.

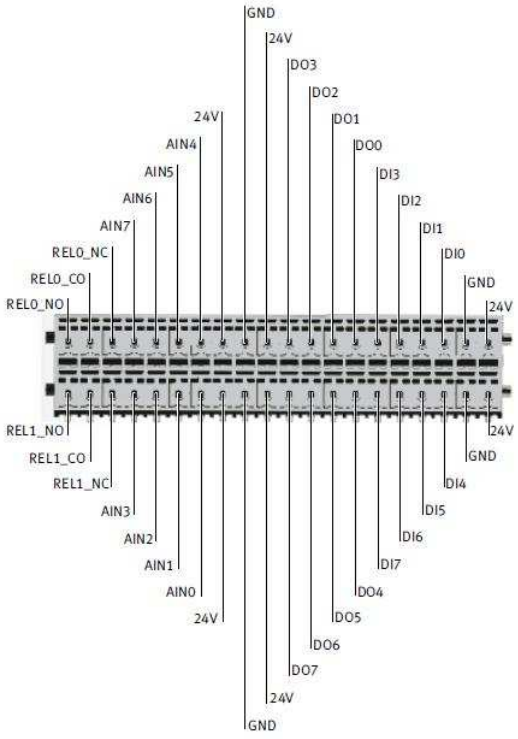


Figura 1.15 Asignación de bornes del interface E/S, tomado de [11]

1.5 NAVEGACIÓN EN ROBOTS MÓVILES

La navegación es la técnica de conducir un robot móvil mientras atraviesa un entorno para alcanzar un destino o meta sin chocar con ningún obstáculo [12].

Cuando se desconoce el entorno, el robot debe poseer la capacidad de reaccionar ante situaciones inesperadas, esto se logra a través de la percepción del entorno mediante el uso de sensores. Mientras que si se trata de un entorno conocido, el uso de los sensores se vuelve secundario y las tareas a seguir serían: planificar una óptima trayectoria libre de obstáculos dependiendo de los puntos de partida y llegada y obviamente que el robot pueda seguir y cumplir físicamente esta trayectoria.

1.5.1 ESQUEMAS DE NAVEGACIÓN EN ROBOTS MÓVILES

El problema de la navegación se lo puede dividir en 4 etapas:

1. *Percepción del mundo:* Se la realiza a través de sensores externos y la creación del mapa de entorno donde va a desplazarse el robot. En el caso de este proyecto, el mapa de entorno es conocido.
2. *Planificación de la ruta:* En base al mapa de entorno, se crea una secuencia ordenada de submetas, la cual se basa en la descripción de la tarea a realizarse y el uso de algún procedimiento estratégico.
3. *Generación del camino:* Interpola la secuencia de submetas definida en la planificación y se procede a la discretización de la secuencia para así generar el camino.
4. *Seguimiento del camino:* Se refiere al desplazamiento del vehículo conforme al camino generado y a través del control adecuado de los actuadores del robot móvil.

Estas etapas pueden llevarse a cabo de forma separada, aunque en el orden especificado. La interrelación existente entre cada una de estas tareas conforma la estructura de control de navegación básica en un robot móvil [13].

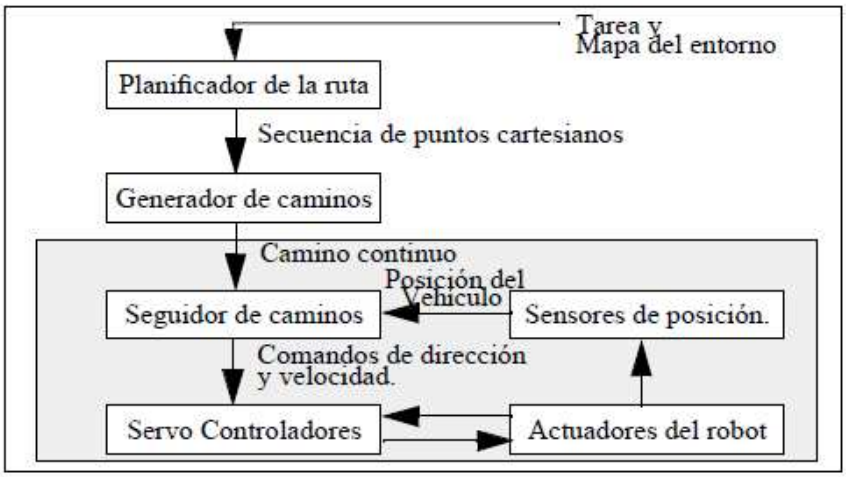


Figura 1.16 Estructura de Control de navegación básica para un robot móvil, tomado de [13]

La estructura anterior responde para un sistema en el que el entorno prácticamente es conocido, de darse el caso de que el modelo del entorno posea ciertas imperfecciones y se tenga cierto grado de incertidumbre, la estructura a seguir sería la de la Figura 1.17.

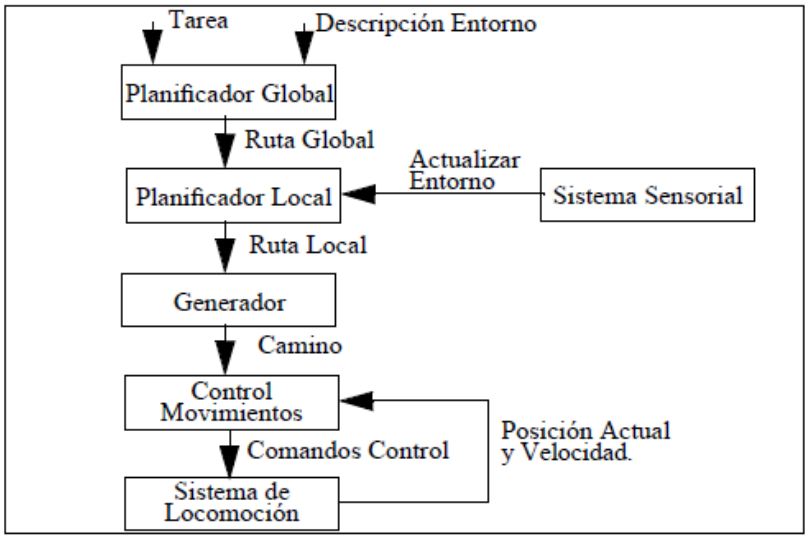


Figura 1.17 Navegador implantado en el robot móvil Blanche AT&T, tomado de [13]

La clave de este esquema es que el robot podría adaptarse a varios entornos inclusive si no se tiene un conocimiento exhaustivo del entorno. Para esto es necesario aplicar lo que se conoce con el nombre de planificación global y planificación local.

- *Planificación global:* Es una aproximación al camino final que va a seguir el robot móvil, consiste en construir o planificar la ruta que lleva al robot a cada una de las submetas. Se la puede llevar a cabo antes de que el robot empiece a ejecutar alguna tarea. Si el entorno es totalmente conocido, es innecesario llevar a cabo la planificación global para la navegación.
- *Planificación local:* Aquí se toma en cuenta los detalles del entorno local al vehículo, mismos que son proporcionados por los sensores externos del robot, aquí se determina la ruta real a seguir. Esta planificación se la realiza en tiempo de ejecución.

Se conozca o no el entorno, la navegación estratégica, va a estar basada en la realización de una manera secuencial y continua de las operaciones de percepción, planificación, generación y seguimiento. Para esto, es necesario conocer el mínimo error posible en la posición actual del robot ya que de este depende la realimentación a realizarse, siendo esta la base de la próxima acción de control.

Mediante el uso de la odometría del vehículo se puede realizar esta acción, pero debido a la naturaleza del método y a las características de los sensores utilizados, la estimación efectuada se ve afectada por errores acumulativos (Watanabe y Yuta, 1.990). Cuando dichos errores alcanzan niveles indeseables se hace necesario eliminarlos mediante la utilización de algoritmos de estimación de la posición basados en referencias externas (González J., 1.993). La navegación estratégica tiene sus limitaciones en entornos dinámicos no conocidos, ya que requiere un completo conocimiento de la dinámica de los posibles obstáculos móviles, además de una adecuada actualización del mapa de entorno [13].

Por otra parte, se podrían utilizar sensores como: ultrasónicos, infrarrojos, táctiles, etc., para que reaccionen en el entorno dinámico, pero aquí estaría perdiendo importancia la planificación y seguimiento de caminos. En base a esto como se observa en la Figura 1.18, se podría aplicar una arquitectura descompuesta en módulos cada uno especializado en una tarea específica, llamadas comportamientos.

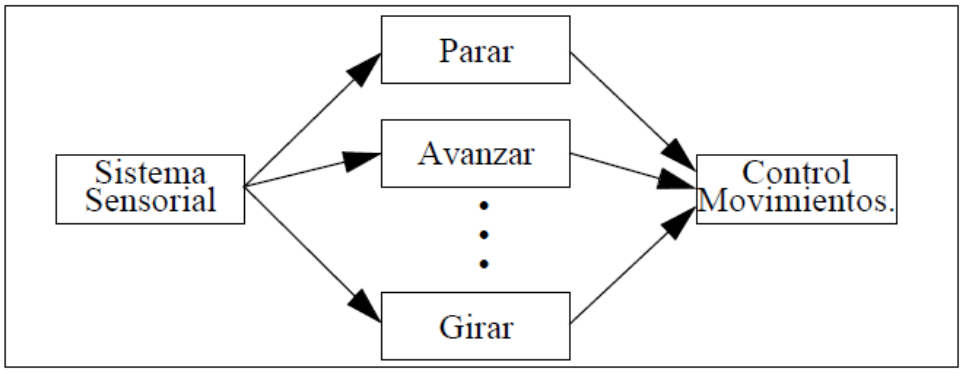


Figura 1.18 Navegación reactiva, tomado de [13]

En este sistema de navegación, según la información proporcionada por los sensores, se activan uno o más comportamientos, siendo el comportamiento final la suma de cada acción simple. Este tipo de navegación ha sido aplicada en múltiples aplicaciones para entornos desconocidos, inclusive dinámicos y sin colisionar con los obstáculos, pero con dificultad pueden seguir un plan establecido, lo que es necesario en misiones reales.

1.5.2 PLANIFICACIÓN DE LA TRAYECTORIA

Planificar es prever y decidir hoy las acciones que nos pueden llevar desde el presente hasta un futuro deseable, no se trata de hacer predicciones acerca del futuro sino de tomar las decisiones pertinentes para que ese futuro ocurra [12].

La idea general en la planificación es encontrar una trayectoria segura capaz de conducir al robot móvil desde un punto de partida hasta un punto de llegada.

El concepto de ruta segura implica el cálculo de un camino al menos continuo en posición, que sea libre de obstáculos. En virtud de esta ruta, el generador construirá las referencias que se le entregan al control de movimientos. Por ello, en la planificación de esta ruta se obvian las características cinemáticas y dinámicas del vehículo, ya que el cómputo de una referencia adecuada que cumpla con estos atributos es tarea del generador de caminos. Por tanto, la ruta al tan sólo asegurar continuidad en posición, supone que únicamente los robots móviles omnidireccionales puedan seguir una referencia de tales características [13].

1.5.2.1 Formalización del problema de la Planificación

El entorno en donde el robot realizará su tarea se lo puede considerar como un subconjunto de configuraciones en las cuales en cualquier instante de tiempo puede encontrarse el robot, igualmente se tendría un subconjunto inalcanzable que llegan a ser los obstáculos.

Se define una configuración q de un robot como un vector cuyas componentes proporcionan información completa sobre el estado actual del mismo. Un robot es un objeto rígido al cual se le puede asociar un sistema de coordenadas móvil. La localización del vehículo en un determinado instante de tiempo queda definida por la relación existente entre el sistema de coordenadas global F_g en virtud del cual está definido todo el entorno de trabajo y su sistema de coordenadas locales asociado F_r [13].

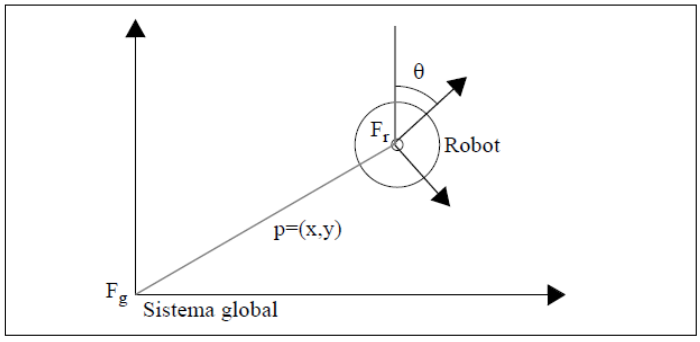


Figura 1.19 Sistema de coordenadas global, y sistema local asociado al robot

La expresión que proporciona el estado actual del robot se define como:

$$q = (p, \theta) = (x, y, \theta) \tag{1.1}$$

Donde p es la posición y θ la orientación.

Siendo $R(q)$ el subconjunto de C ocupado por el robot R cuando este se encuentra en q, modelando al robot de forma circular con radio ρ, $R(q)$ se define como:

$$R(q) = \{q_i \in C / \|q, q_i\| \leq \rho\} \tag{1.2}$$

Siendo C el espacio de configuraciones del robot R de todas las configuraciones q que puede tomar el robot en el entorno de trabajo.

De tratarse de un robot puntual la expresión anterior, ρ sería nulo y se tendría:

$$R(q) = \{q\} \tag{1.3}$$

Los obstáculos dentro del mapa de entorno se los puede definir como un conjunto de objetos rígidos B, distribuidos en el espacio de configuraciones C.

$$B = \{b_1, b_2, \dots, b_q\} \tag{1.4}$$

Sea $\boldsymbol{b_i(q)}$ el conjunto de configuraciones ocupadas por un obstáculo, se puede definir al subconjunto de C que especifican el espacio libre de obstáculos por:

$$\boldsymbol{C_l} = \left\{ \boldsymbol{q} \in \frac{\boldsymbol{C}}{\boldsymbol{R(q)}} \cap \left(\bigcup_{\boldsymbol{i=1}}^{\boldsymbol{q}} \boldsymbol{b_i(q)} \right) = \emptyset \right\} \tag{1.5}$$

En base a estas expresiones, se puede decir que una ruta $\boldsymbol{Q_r}$ es una sucesión de posturas pertenecientes a $\boldsymbol{C_l}$ tal que conecta la posición inicial $\boldsymbol{q_a}$ con la postura final $\boldsymbol{q_f}$, se tiene que:

$$\boldsymbol{Q_r} = \{\boldsymbol{q_a}, \dots, \boldsymbol{q_f} / \boldsymbol{q_i} \in \boldsymbol{C_l}\} \tag{1.6}$$

Por lo que implica la construcción de una ruta definida como:

$$\boldsymbol{\tau}: [\mathbf{0}, \mathbf{1}] \rightarrow \boldsymbol{C_l} \tag{1.7}$$

Donde:

$$\boldsymbol{\tau(0) = q_a} \qquad \boldsymbol{y} \qquad \boldsymbol{\tau(1) = q_f} \tag{1.8}$$

Suponiendo que se trata de un robot omnidireccional, teniendo en cuenta el concepto de continuidad, se tiene:

$$\boldsymbol{\lim_{s \rightarrow s_0} \| \tau(s), \tau(s_0) \| = 0} \tag{1.9}$$

1.5.2.2 Métodos de Planificación

Se fundamentan en una primera fase de construcción de algún tipo de grafo sobre el espacio libre, según la información poseída del entorno, para posteriormente emplear un algoritmo de búsqueda en grafos que encuentra el camino óptimo según cierta función de coste [13].

1.5.2.2.1 Grafos de Visibilidad

Este es uno de los métodos más rápidos de planificación de movimientos, se aplica principalmente en espacios de configuración bidimensional cuando los obstáculos con poligonales [14].

Para la generación del grafo este método introduce el término de visibilidad, según el cual define dos puntos del entorno como *visibles* si y solo si se pueden unir mediante un segmento rectilíneo que no intersecte ningún obstáculo. En otras palabras, el segmento definido debe yacer en el espacio libre del entorno C_l [13].

Se consideran como nodo del grafo a la posición inicial, la posición final y todos los vértices de los obstáculos, siendo el grafo el resultado de la unión de los nodos visibles, tal como se muestra en la Figura 1.20.

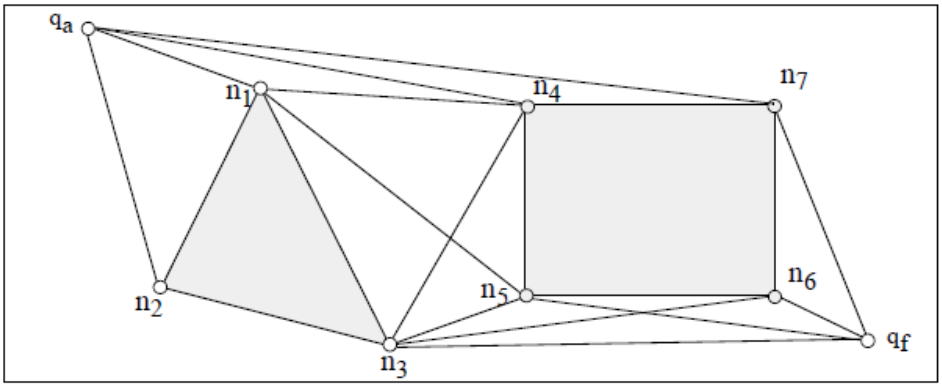


Figura 1.20 Grafo de visibilidad en un entorno de dos obstáculos, tomado de [13]

A través de un algoritmo de búsqueda de grafos se escoge la ruta más óptima que una la posición inicial con la final.

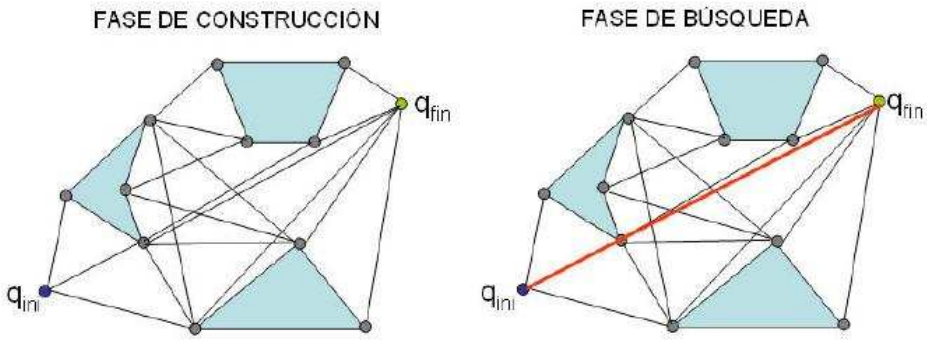


Figura 1.21 Grafo de visibilidad y ruta óptima, tomado de [14]

Aunque en principio el método está desarrollado para entornos totalmente conocidos, existe una versión denominada LNAV (Rao y otros, 1988) capaz de efectuar una planificación local a medida que se realiza la tarea de navegación. Este algoritmo, que parte de una determinada posición, determina los nodos visibles desde el punto actual. Elige el más cercano de los nodos visibles, según distancia euclídea a la posición final, para desplazarse posteriormente al nodo seleccionado y marcarlo como visitado. Desde esta nueva posición se vuelve a iterar el proceso hasta llegar a la posición final (éxito), o bien no existen más nodos sin visitar (fracaso) [13].

En grafos de visibilidad, para la búsqueda de la ruta óptima que lleve al vehículo desde una posición A hacia una posición B, se tienen logaritmos especializados para encontrar esta ruta, por ejemplo en la Figura 1.22, se tiene el modelado del entorno libre de obstáculos a través del uso de dos cadenas de segmentos.

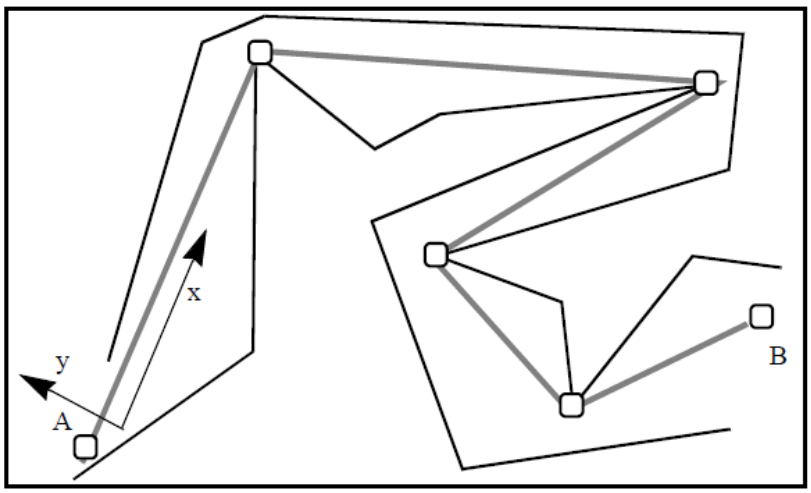


Figura 1.22 Planificación con el espacio libre de obstáculos modelado mediante cadenas, tomado de [13]

Este método se restringe a esquemas de entornos muy concretos. Aquí se puede ver como la ruta óptima se basa en la unión de los nodos de las zonas convexas tal que dos nodos consecutivos son visibles.

El uso de métodos de planificación basados en grafos de visibilidad está muy extendido, debido a que se pueden construir algoritmos a bajo coste computacional que resuelvan el referido problema. Sin embargo, utilizar como

nodos los vértices de los obstáculos implica que no son inmediatamente aplicables en la práctica, ya que un robot móvil real no consiste en un punto [13].

1.5.2.2.2 Diagramas de Voronoi

Los Diagramas de Voronoi se encuentran entre las más importantes estructuras en geometría computacional, este diagrama codifica la información de proximidad entre elementos [15].

Este método, contrario al mencionado anteriormente, considera a la ruta lo más alejada posible de los obstáculos. En un espacio de dos dimensiones, todos los puntos que son equidistantes de 2 objetos (obstáculos) son considerados parte del Diagrama generalizado de Voronoi.

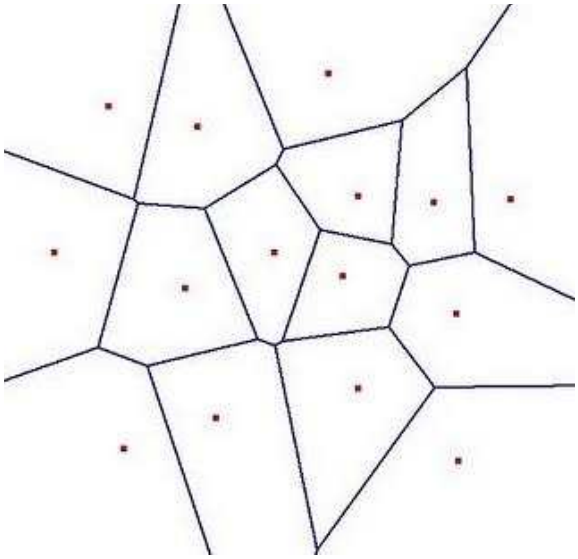


Figura 1.23 Diagrama de Voronoi en un Mapa de entorno donde se consideran a los obstáculos como puntos, tomado de [18]

1.5.2.2.3 Roadmap Probabilístico (PRM)

Consiste en generar un número n de configuraciones libres de colisión de forma aleatoria y uniforme en toda el área de trabajo.

Si n es suficientemente grande, el espacio de trabajo se cubre totalmente y el roadmap obtenido o grafo generado está completamente conectado, en caso contrario éste estará formado por componentes inconexos lo que indicará que no se ha capturado eficientemente la conectividad de dicho espacio [16].

En la Figura 1.24 se muestra la generación de las configuraciones aleatorias válidas junto con los puntos de partida y llegada del robot.

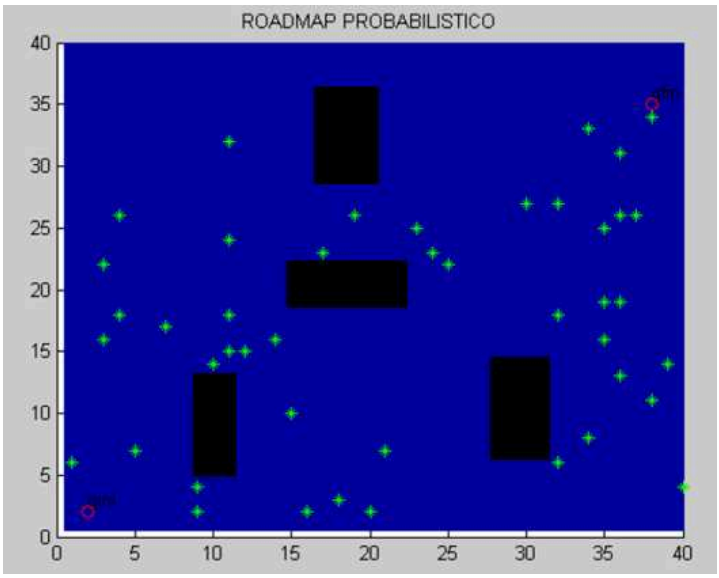


Figura 1.24 Generación de puntos aleatorios libres de colisiones, tomado de [16]

Posteriormente se prosigue a conectar cada uno de los nodos con sus nodos más cercanos según una métrica que depende del número de objetos en el entorno de trabajo. Y finalmente aplicar un algoritmo que obtenga la ruta más óptima, en este caso el algoritmo A*, mismo que será explicado más adelante.

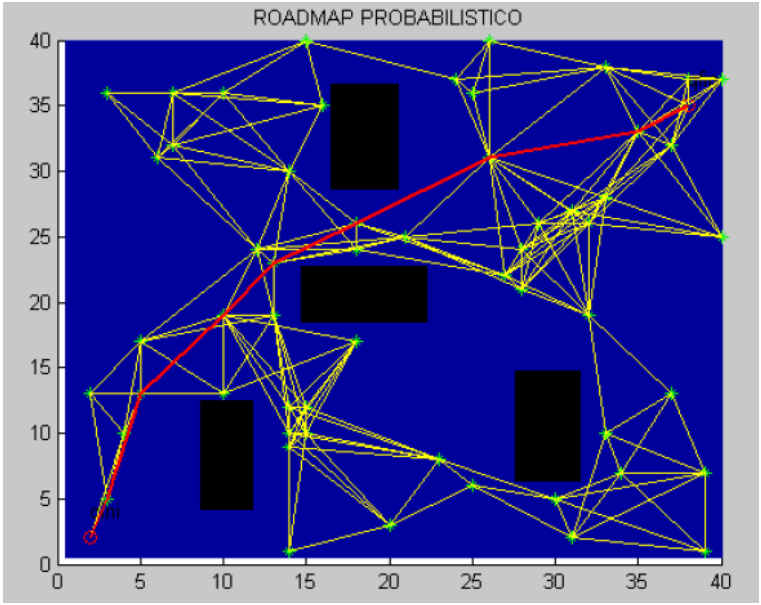


Figura 1.25 Enlaces válidos en la etapa de conexión y camino entre q_{ini} y q_{fin} mediante el algoritmo A*, tomado de [16]

El método de planificación de caminos empleando roadmap probabilístico presenta una serie de ventajas e inconvenientes, los más singulares son [16]:

Ventajas:

- No requiere información estructural del espacio de las configuraciones, basta con saber si una configuración determinada *q* presenta colisión.
- Es aplicable a cualquier número y tipos de grado de libertad

Inconvenientes:

- No se puede asegurar cuando no es posible encontrar una solución.
- Problemas con la elección de nodos, número de nodos y distancia entre ellos.
- Dificultad para saber si los nodos seleccionados dan una visión completa del espacio libre.
- Suele dar problemas en zonas de pasillos y en general en zonas con alta densidad de obstáculos.

1.5.2.2.4 Modelado del Espacio Libre

Igualmente en este método los obstáculos se los representa como polígonos. La planificación se lleva a cabo a través de los CRG, cilindros rectilíneos generalizados, y al igual que Voronoi, con el uso de los CRG se pretende que el robot se mueva lo más alejado de los obstáculos. La ruta será una configuración de CRG interconectados, tal que la configuración inicial o de partida se encuentre en el primer cilindro de la sucesión y la configuración final en el último cilindro.

La construcción de un CRG se realiza a partir de las aristas de los distintos obstáculos que se encuentran en el entorno. Para que un par de aristas $^{1}a_i$ y $^{2}a_j$ pertenecientes a los obstáculos b_1 y b_2 respectivamente puedan formar un cilindro generalizado, deben cumplir las siguientes condiciones [13]:

i. La arista $^{1}a_i$ está contenida en una recta que divide al plano en 2 regiones. La arista $^{2}a_j$ debe yacer por completo en la región opuesta en la que se encuentra situada b_1. Este es un criterio simétrico.

ii. El producto escalar de los vectores normales con dirección hacia el exterior del obstáculo que contiene cada arista debe resultar negativo.

Si se cumplen estas 2 condiciones, quiere decir que las 2 aristas se encuentran enfrentadas y por ende se puede construir un CRG.

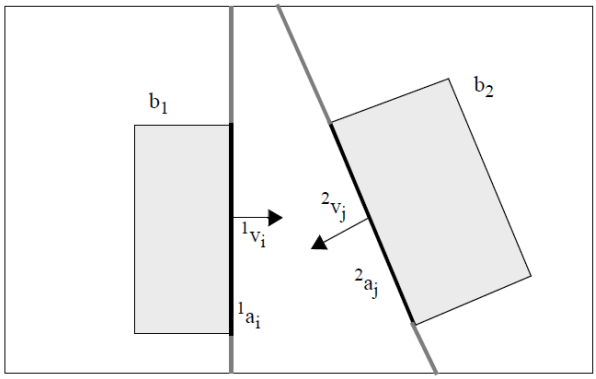


Figura 1.26 Condiciones que deben cumplir dos aristas para construir un CRG, tomado de [13]

El proceso para construir un CRG será el siguiente:

- Cálculo del eje del CRG, se define como la bisectriz del ángulo α formado por el corte de las rectas que contienen las aristas $^{1}a_i$ y $^{2}a_j$ que cumplen con las condiciones antes mencionadas.
- Por ambos lados de dichas aristas se construyen rectas paralelas al eje, con origen en los vértices de las aristas implicadas y con extremo señalado por la proyección del primer obstáculo que corta el eje.

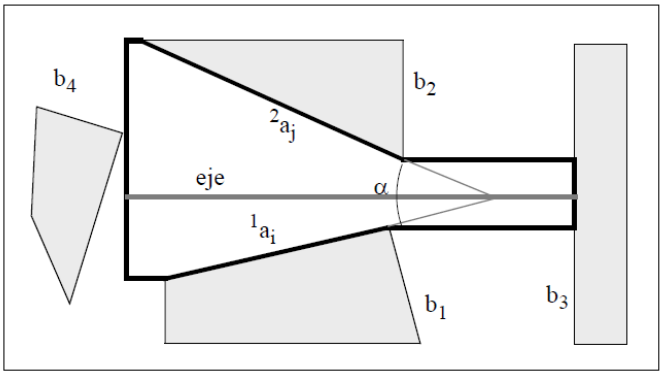


Figura 1.27 Construcción de un CRG, tomado de [13]

Repitiendo este proceso se construye una red CRG en el entorno del robot que modela el espacio libre del mismo. El robot navegará por el eje del cilindro, en el cual se encuentran anotadas para cada punto el rango de orientaciones admisibles. El paso de un CRG a otro se produce siempre y cuando sus ejes intersecten y la intersección del rango de orientaciones admisibles en el punto de corte de ambos ejes no sea nulo [13].

1.5.2.2.5 Descomposición en celdas

Se basa en una descomposición en celdas del espacio libre. La ruta desde la posición inicial q_a hasta la final q_f consiste en un sucesión de celdas que no presente discontinuidades, tal que la primera celda contenga a q_a y la última a q_f.

En este método no se encuentra una sucesión de segmentos, sino una sucesión de celdas, por lo que es necesaria la construcción de un grafo de conectividad encargado de definir la ruta además de la descomposición de celdas.

La descomposición en celdas implica construir celdas con determinada forma geométrica tal que calcular un camino entre dos configuraciones distintas pertenecientes a la celda resulte fácil. Así como comprobar si dos celdas son adyacentes sea lo más simple posible y la unión de todas las celdas debe corresponder exactamente al espacio libre.

Una vez que se tiene la descomposición en celdas, la construcción del grafo de conectividad se basa en que los nodos van a ser cada una de las celdas, y existe un arco entre dos celdas si y solo si estas son adyacentes.

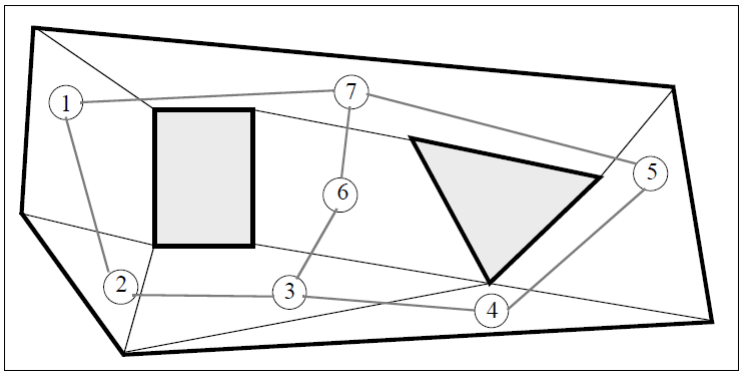


Figura 1.28 Descomposición en celdas y grafo de conectividad, tomado de [13]

Una vez especificado el grafo de conectividad, sólo queda emplear un algoritmo de búsqueda en grafos, para la detección de la celda que contiene la postura a la cual se desea llegar, tomando como partida la que contiene la postura inicial [13].

Dentro de los métodos basados en descomposición en celdas, el método más sencillo es el de descomposición trapezoidal. Se construyen rectas paralelas al eje Y a partir de los vértices de cada elemento del entorno, estas rectas quedan delimitadas por el corte con las líneas de los elementos del entorno. Como se muestra en la Figura 1.29.

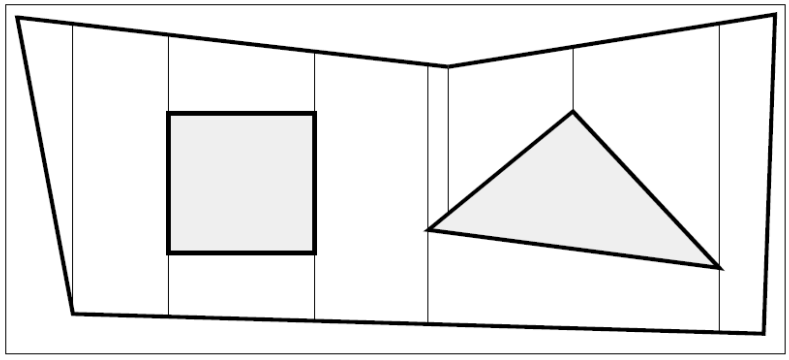


Figura 1.29 Descomposición trapezoidal del espacio libre, tomado de [13]

El grafo de conectividad se construye a través de la unión de los puntos medios de las rectas definidas.

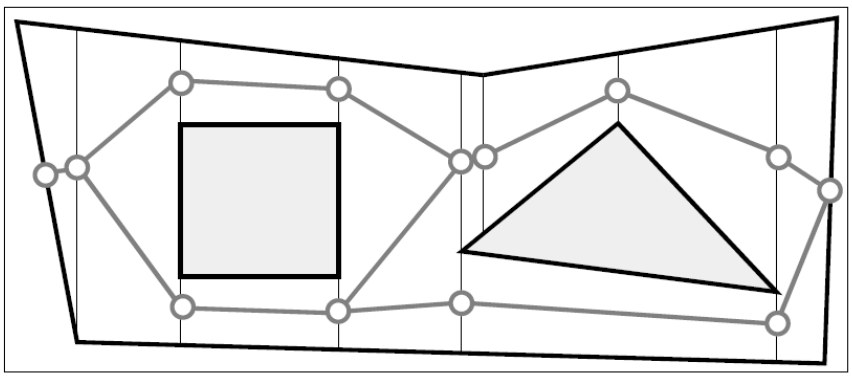


Figura 1.30 Grafo de conectividad de una descomposición trapezoidal, tomado de [13]

Este tipo de enfoque se presta a muchas variantes, por ejemplo la utilización de varios niveles de resolución para una búsqueda jerarquizada (Kambhampati y

Davis, 1986) o bien el uso de celdas en tres dimensiones para la planificación de caminos en espacios tridimensionales (Stentz, 1990) [13].

1.5.2.2.6 Campos Potenciales

Este método está basado en técnicas reactivas de planificación, esta técnica se centra en la planificación local en entornos desconocidos.

Esta teoría considera al robot como una partícula bajo la influencia de un campo potencial artificial. La función potencial *U* en un punto *p* del espacio euclídeo, consiste en la composición de un potencial atractivo $\boldsymbol{U_a(p)}$ que atrae el robot a la posición destino y un potencial repulsivo $\boldsymbol{U_r(p)}$ que lo hace alejarse de los obstáculos. Tal que:

$$\boldsymbol{U(p) = U_a(p) + U_r(p)} \tag{1.10}$$

El potencial artificial $\boldsymbol{U(p)}$ influye en la fuerza artificial $\boldsymbol{F(p)}$, tal que:

$$\boldsymbol{F(p) = -\nabla U(p)} \tag{1.11}$$

Así mismo la fuerza $\boldsymbol{F(p)}$ está compuesta por una fuerza de repulsión y una fuerza de atracción:

$$\boldsymbol{F(p) = F_a(p) + F_r(p)} \tag{1.12}$$

Así, la navegación basada en campos potenciales se basa en llevar a cabo las siguientes acciones [13]:

i. Calcular el potencial $\boldsymbol{U(p)}$ que actúa sobre el vehículo en la posición actual *p* según la información recabada de los sensores.
ii. Determinar el vector fuerza artificial $\boldsymbol{F(p)}$ según la expresión (1.12)
iii. En virtud del vector calculado construir las consignas adecuadas para que los actuadores del vehículo hagan que éste se mueva según el sentido, dirección y aceleración dadas por $\boldsymbol{F(p)}$.

La iteración del ciclo anterior constituye una navegación reactiva basada en campos potenciales. El potencial de atracción debe ser función de la distancia

euclídea a la posición destino, mientras más cerca este el robot, el potencial debe disminuir su influencia. Por otro lado, el potencial repulsivo debe solo influir el momento en que el robot se encuentre demasiado cerca de los obstáculos. En la posición destino, es necesario que la suma de los dos potenciales sea nula.

En caso de conocer todo el entorno de trabajo, se puede construir una ruta desde $\boldsymbol{p_i}$, la posición actual, a la próxima posición a alcanzar $\boldsymbol{p_{i+1}}$, resulta:

$$\boldsymbol{p_{i+1} = p_i + \delta_i \mathbf{J}(U(p))} \tag{1.13}$$

Donde $\boldsymbol{\delta_i}$ es un factor de escalado que define la longitud del segmento entre $\boldsymbol{p_i}$ y $\boldsymbol{p_{i+1}}$ tal que dicho segmente sea libre de obstáculos, y $\mathbf{J}(\boldsymbol{U(p)})$ representa el jacobiano de la función potencial en *p*.

El problema que existe en este tipo de métodos son los mínimos locales, que son lugares donde el potencial resulta nulo pero no se trata de la posición final. Evitar estos problemas implica definir ciertas funciones potenciales que eviten la aparición de mínimos locales, por ejemplo modelados mediante círculos. Otra solución es utilizar un algoritmo de búsqueda de grafos, se divide al entorno en celdas y cada una contiene su potencial. Un algoritmo utilizable es $\boldsymbol{A^*}$ usando la función potencial como función de coste.

Para la implementación del presente proyecto se seleccionó a los diagramas de Voronoi como el método de planeación de trayectorias para ser desarrollado, debido a que en comparación a los otros métodos mencionados, este posee menos desventajas y además presenta la mayor seguridad en la ruta del robot al considerar a la misma lo más alejada de los obstáculos, razones por lo cual se detalla este método en el subcapítulo a continuación.

1.5.3 DIAGRAMAS DE VORONOI

Los diagramas de Voronoi se definen como una proyección del espacio libre del entorno en una red de curvas unidimensionales yacientes en dicho espacio libre. Formalmente se definen como una retracción (Janich, 1984) con preservación de la continuidad. Si el conjunto $\boldsymbol{C_l}$ define las posiciones libres de obstáculos de un entorno, la función retracción $\boldsymbol{RT}$ construye un subconjunto $\boldsymbol{C_v}$ continuo de $\boldsymbol{C_l}$ [13].

$$RT(q): C_l \rightarrow C_v / C_v \subset C_l \tag{1.14}$$

Existe un camino desde una posición inicial q_a hasta una posición final q_f, libre de obstáculos, si y solo si existe una curva continua desde $RT(q_a)$ hasta $RT(q_f)$.

La principal idea de la construcción del diagrama de Voronoi es ampliar al máximo la distancia que existe entre el robot y los obstáculos, por tanto, el diagrama resulta el lugar geométrico de las configuraciones que se encuentran a la misma distancia de los obstáculos más próximos del entorno.

1.5.3.1 Obstáculos representados como Puntos

Si se consideran a los obstáculos de un mapa de entorno como puntos, se pueden definir las siguientes propiedades [17]:

i. Dos puntos p_i y p_j son vecinos si comparten una arista. Una arista es la bisectriz perpendicular del segmento $p_i p_j$.
ii. Un vértice es un punto equidistante a tres generadores (si lo es a más de tres se tiene casos degenerados) y es la intersección de tres aristas.
iii. Una región de Voronoi es un polígono convexo o es una región no acotada
iv. Una región de Voronoi es no acotada si su punto generador pertenece a la envolvente convexa de la nube de puntos.
v. Dentro del círculo con centro en un vértice de Voronoi y que pasa por 3 puntos generadores no puede existir ningún otro punto generador.

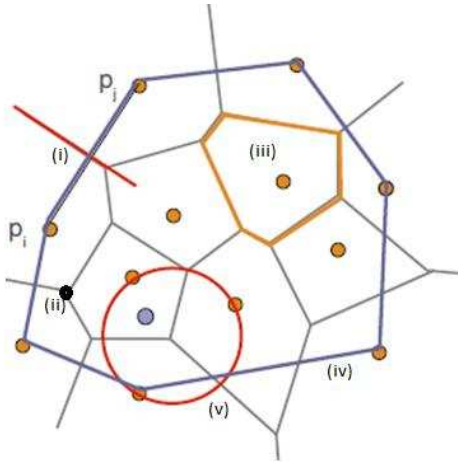


Figura 1.31 Diagrama de Voronoi y Propiedades, tomado de [17]

1.5.3.1.1 Algoritmos de Construcción: Método Divide y Vencerás

Partiendo del conjunto de puntos S, se sigue los siguientes pasos [17]:

1. Dividir el conjunto de puntos S en 2 subconjuntos S_1 y S_2 de aproximadamente el mismo tamaño.

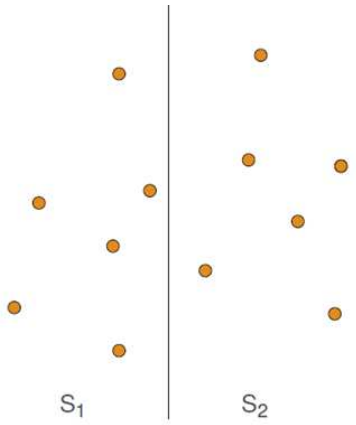


Figura 1.32 División de S en dos subconjuntos S_1 y S_2, tomado de [17]

2. Calcular recursivamente los diagramas de Voronoi Vor(S_1) y Vor(S_2). Para esto, se trazan las bisectrices que existen entre 2 puntos cercanos de cada subconjunto, se eliminan las porciones de arista y vértices que quedan dentro de la región de Voronoi obtenida perteneciente para cada punto.

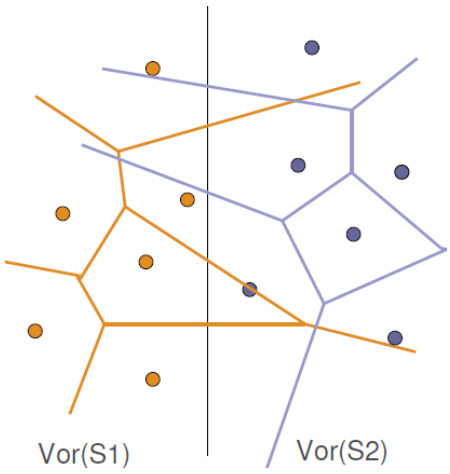


Figura 1.33 Diagramas de Voronoi Vor(S_1) y Vor(S_2), tomado de [17]

3. Unir Vor(S_1) y Vor(S_2) para obtener Vor(S). Para esto se requiere encontrar la cadena divisioria δ. Si S_1 y S_2 están separados por una línea

vertical, existe una línea poligonal monótona creciente tal que todo punto situado a la izquierda (derecha) de dicha poligonal está en la región de Voronoi de un punto de S_1 (S_2).

A partir de una línea que llega desde infinito y alcanza a la primera región de S_1 (la del punto p) y la primera de S_2 (la del punto q), se calcula la bisectriz entre p y q hasta alcanzar una arista de Voronoi, de S_1 o de S_2.

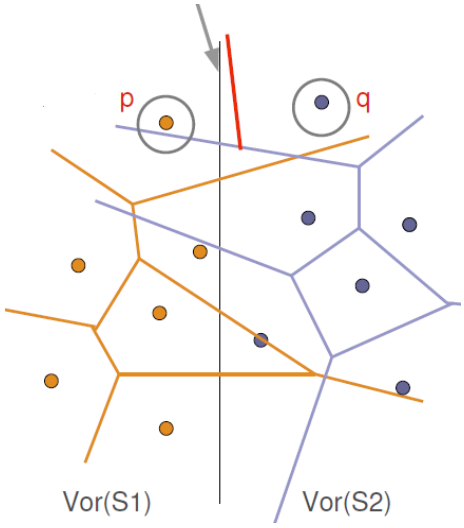


Figura 1.34 Proceso para hallar la cadena divisioria δ, tomado de [17]

Si se alcanza un eje de S_2 se actualiza el punto q con el punto vecino, si se alcanza una arista de S_1 se actualiza el punto p [17].

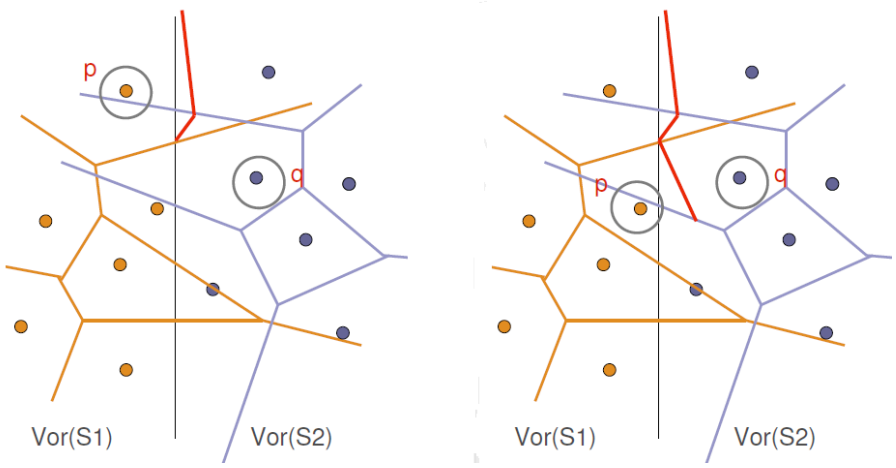


Figura 1.35 Actualización de puntos para cálculo de la arista de la cadena δ, tomado de [17]

Por último se eliminan las líneas de Vor(S_1) que queden a la derecha de δ así como las líneas de Vor(S_2) que queden a la izquierda de δ [17].

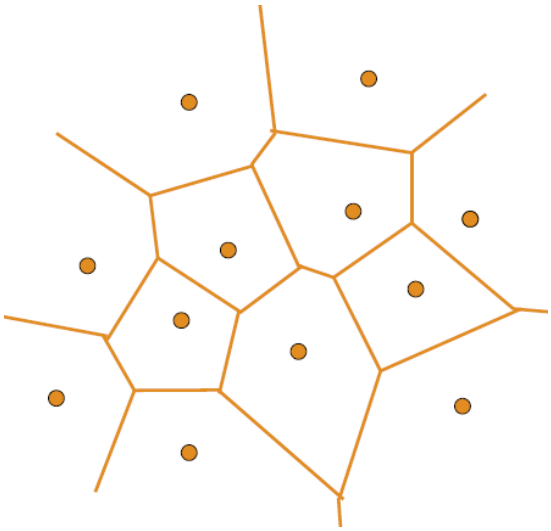


Figura 1.36 Diagrama de Voronoi del conjunto de puntos S, tomado de [17]

De esta manera, el diagrama de Voronoi llega a ser el camino más seguro para desplazarse el robot de un punto a otro de este camino.

1.5.3.2 Obstáculos representados como Polígonos

Para una representación más real se consideran a los obstáculos como polígonos, debido a que físicamente un obstáculo no es un punto. El diagrama de Voronoi estará compuesto por dos tipos de segmentos: rectas y parábolas.

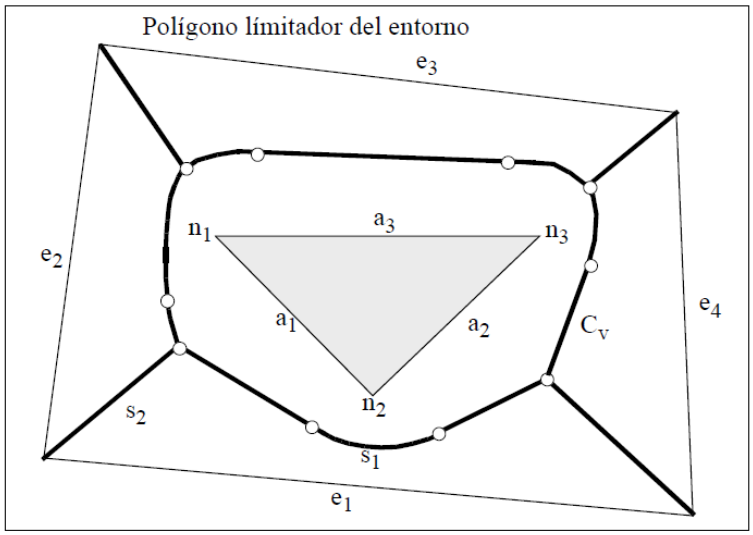


Figura 1.37 Retracción del espacio libre en un diagrama de Voronoi, tomado de [13]

El lugar geométrico de las configuraciones que se hallan a la misma distancia de dos aristas de dos obstáculos distintos será una recta, y si se trata de una arista y un vértice, será una parábola.

En la Figura 1.37 se puede observar el diagrama de Voronoi C_v representado por las líneas gruesas, el entorno se encuentra delimitado por un polígono de aristas $\{e_1, e_2, e_3, e_4, e_5\}$ y un obstáculo triangular de vértices $\{n_1, n_2, n_3\}$ y aristas $\{a_1, a_2, a_3\}$. Claramente se pueden apreciar los dos tipos de segmentos que componen el diagrama de Voronoi, el segmento s_1 (parabólico) corresponde al lugar geométrico de los puntos equidistantes entra el vértice n_2 y la arista e_1. Así como el segmento s_2 (rectilíneo) corresponde al lugar geométrico entre las aristas e_1 y e_2.

Dado un configuración *q* que no pertenece a C_v, existe un único punto *p* más cercano, perteneciente a un vértice o arista de un obstáculo. La función $RT(q)$ se define como el primer corte con C_v de la línea que une *p* con *q* [13].

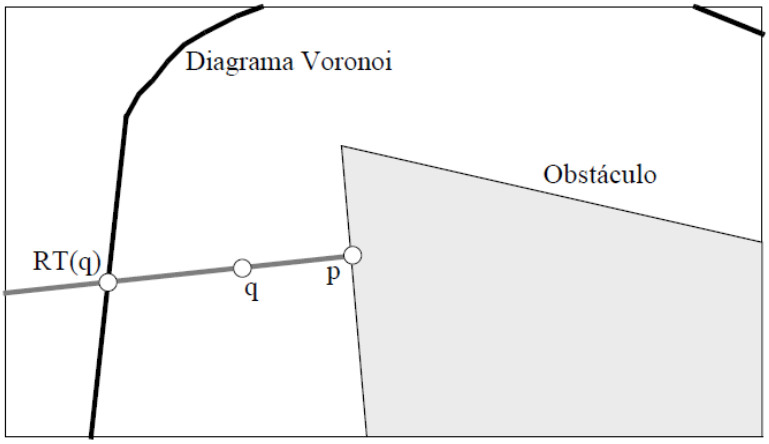


Figura 1.38 Imagen de una configuración q en el diagrama de Voronoi, tomado de [13]

Para la planificación de la ruta, el algoritmo a implementarse consiste en encontrar la secuencia de segmentos s_i del diagrama de Voronoi tal que conecten $RT(q_a)$ con $RT(q_f)$. El algoritmo se describe como [13]:

i. Calcular el diagrama de Voronoi
ii. Calcular $RT(q_a)$ y $RT(q_f)$

iii. Encontrar la secuencia de segmentos $\{s_1, \ldots, s_p\}$ tal que $RT(q_a)$ pertenece a s_1 y $RT(q_f)$ pertenezca a s_p.

iv. Si la secuencia es encontrada, devolver la ruta, caso contrario, indicará condición de error.

Al igual que los grafos de visibilidad, este método también trabaja en entornos totalmente conocidos y con obstáculos modelados mediante polígonos. Sin embargo, también existen versiones para la utilización del mismo con obstáculos inesperados (Meng, 1988) [13].

Si se requiere una información más detallada acerca de los diagramas de Voronoi se puede consultar el Anexo C de este trabajo.

1.5.4 GENERACIÓN DE CAMINOS

El camino se lo construye en base a la planificación de la ruta y debe estar libre de obstáculos, la importancia de un camino con buenas propiedades se basa en la capacidad del seguidor de caminos para ejecutar la navegación con el menor error posible. La función del generador es convertir una ruta en un camino, llevar al robot de una posición inicial a una final tal que se elimine la restricción de omnidireccionabilidad inherente a la definición de ruta.

El camino se define como la discretización de una curva continua que interpola ciertos puntos elegidos de la ruta calculada por el planificador. Por tanto, el problema de la definición de un camino con buenas propiedades pasa por la construcción de la función camino adecuada que las posea. Las características buscadas son aquellas que hacen posible el seguimiento del camino especificado según el comportamiento cinemático y dinámico del vehículo [13].

Para la generación de la ruta en este proyecto, se hace uso de un algoritmo denominado A*, mismo que se describe en el siguiente subcapítulo.

1.5.4.1 Algoritmo A*

El algoritmo de búsqueda A* se clasifica dentro de los algoritmos de búsqueda en grafos. Presentado por primera vez en 1968 por Peter E. Hart, Nils J. Nilsson y Bertram Raphael, el algoritmo A* encuentra, siempre y cuando se cumplan unas

determinadas condiciones, el camino de menor coste entre un nodo origen y uno objetivo [19].

El problema de algunos algoritmos de búsqueda es que se guían en exclusiva por la función heurística, la cual puede no indicar el camino de coste más bajo, o solo se guían por el coste real de desplazarse de un nodo a otro. Es por ello, que un buen algoritmo de búsqueda informada debería tener en cuenta ambos factores, el valor heurístico de los nodos y el coste real del recorrido.

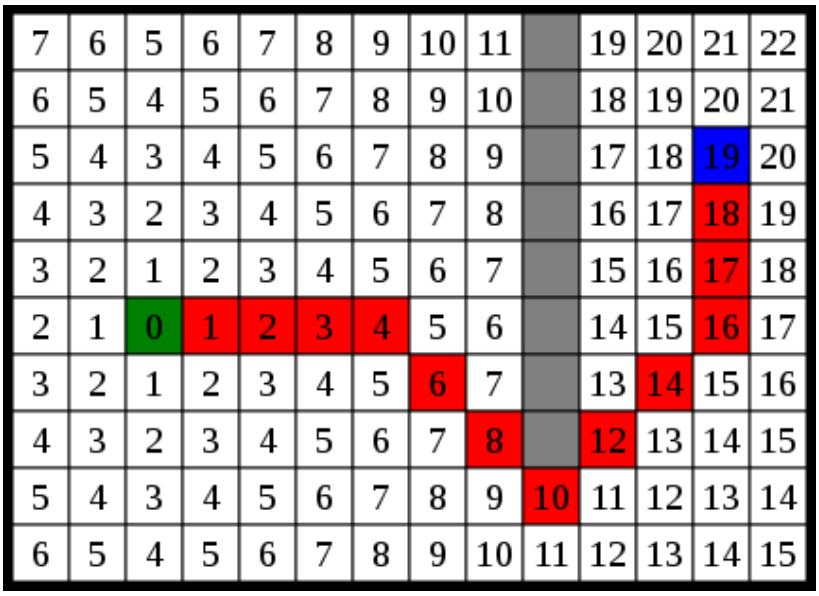


Figura 1.39 Ejemplo de aplicación del algoritmo A*, tomado de [19]

Una heurística es una función matemática $h'(n)$ definida en los nodos de un árbol de búsqueda que sirve como una estimación del coste del camino más económico de un nodo dado hasta el nodo objetivo.

El algoritmo A* utiliza una función de evaluación:

$$f(n) = g(n) + h'(n) \tag{1.15}$$

Donde:

- $h'(n)$ representa el valor heurístico del nodo a evaluar n desde el nodo actual hasta el final.
- $g(n)$ representa el coste real del camino recorrido para llegar a dicho nodo.

El algoritmo A* mantiene dos estructuras de datos auxiliares que se los va a denominar abiertos y cerrados. Para datos abiertos, se tiene implementada una especie de cola de prioridad, ordenada por el valor $f(n)$ de cada nodo. Para datos cerrados, se guarda la información de los nodos que ya han sido visitados.

En cada paso del algoritmo, se expande el nodo que está primero en abiertos, y en caso de no ser un nodo objetivo, calcula la función $f(n)$ de todos sus nodos "hijos", los inserta en abiertos y el nodo evaluado pasa a cerrados.

El algoritmo es una combinación entre búsquedas del tipo: primero en anchura con primero en profundidad. Mientras $h'(n)$ tiende a primero en profundidad, $g(n)$ tiende a primero en anchura. De esta manera, se cambia de camino de búsqueda cada vez que existen nodos más prometedores.

1.5.4.1.1 Propiedades [19]

- A* es un algoritmo completo: en caso de existir una solución, siempre dará con ella.
- Si para todo nodo n se cumple $g(n) = 0$, se trata de una búsqueda voraz.
- Si para todo nodo n se cumple $h'(n) = 0$, el algoritmo A* pasa a ser una búsqueda de costes uniforme no informada.
- Si $h'(n) = 0$ y $g(n) = 0$ la búsqueda será aleatoria.
- Para garantizar que el algoritmo sea óptimo, la función $h'(n)$ debe ser admisible, esto es, no debe sobrestimar el coste real de alcanzar el nodo objetivo. De no cumplirse esta condición, el algoritmo pasa a denominarse simplemente A y a pesar de seguir siendo completo no asegura que el resultado obtenido sea el camino de coste mínimo.
- Como una desventaja se tiene que a pesar de encontrar el camino más óptimo, se debe tener en cuenta que se desperdicia esfuerzo explorando rutas que parecieron buenas.

1.5.4.1.2 Complejidad computacional

La complejidad computacional del algoritmo está íntimamente relacionada con la calidad de la heurística que se utilice en el problema. En el peor de los casos, con

una heurística de pésima calidad, la complejidad será exponencial, mientras que en el mejor de los casos, con una buena heurística, el algoritmo se ejecutará en tiempo lineal.

1.5.4.1.3 Complejidad en memoria

El espacio requerido por A* para ser ejecutado es su mayor problema, debido a que tiene que almacenar todos los posibles siguientes nodos de cada estado, la cantidad de memoria que requerirá será exponencial con respecto al problema.

Una vez concluido el primer capítulo que abarca la teoría necesaria para el desarrollo del presente proyecto se procede a describir en el segundo capítulo el desarrollo del software, es decir la programación de los algoritmos necesarios para generar una trayectoria óptima así como la implementación de la misma en el robot Robotino®.

CAPÍTULO 2

DESARROLLO DEL SOFTWARE

2.1 INTRODUCCIÓN

Para la programación de la aplicación presentada en el presente proyecto se utilizó LabVIEW, software desarrollado por National Instruments, debido a que proporciona todas las herramientas requeridas para el desarrollo del proyecto.

LabVIEW es un entorno de programación gráfica usado para desarrollar sistemas sofisticados de medida, pruebas y control a través de íconos gráficos y cables que parecen un diagrama de flujo. Ofrece una integración con miles de dispositivos de hardware y brinda cientos de bibliotecas integradas para análisis avanzado y visualización de datos.

Los programas desarrollados en LabVIEW son denominados instrumentos virtuales (VI's) debido a su similitud con los instrumentos físicos.

Para la implementación del programa fue necesario hacer uso de tool kits adicionales como: LabVIEW Robotics Module, MathScript RT Module y Robotino LabVIEW driver.

2.1.1 LABVIEW ROBOTICS MODULE

Es un paquete de software que permite desarrollar aplicaciones de robótica, especialmente para el diseño de sistemas móviles autónomos, utilizando LabVIEW, algún otro software de National Instruments y drivers de varios dispositivos. Presenta las siguientes características generales:

- Entorno de programación gráfica de alto nivel.
- E/S integrada disponible para PCs de escritorio, sistemas en tiempo real y FPGAs.
- Paralelismo inherente para visualizar fácilmente tareas concurrentes.

- Conectividad con sensores y actuadores desde los principales proveedores incluyendo SICK, Garmin y Maxon.
- Herramientas para importar código desde otros lenguajes incluyendo C/C++ y VHDL.
- Conectividad abierta con IP de terceros inlcuyendo JAUS, cinemática inversa y entornos de simulación.

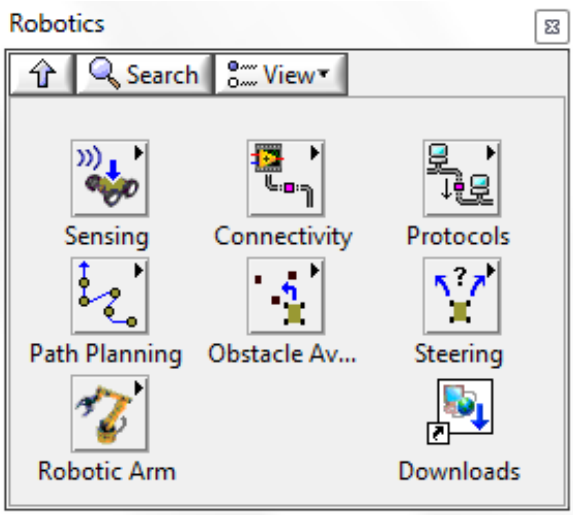


Figura 2.1 VIs de LabVIEW Robotics Module

A continuación se presenta una descripción de cada uno de los VIs que conforman el LabVIEW Robotics Module.

2.1.1.1 Conectivity VIs

Se utiliza los VIs de conectividad para trabajar con software de robótica de otros fabricantes, incluyendo sus productos.

Figura 2.2 Conectivity VIs

2.1.1.2 Obstacle Avoidance VIs

Se utiliza los VIs de evasión de obstáculos para implementar en un robot móvil la evasión de los obstáculos.

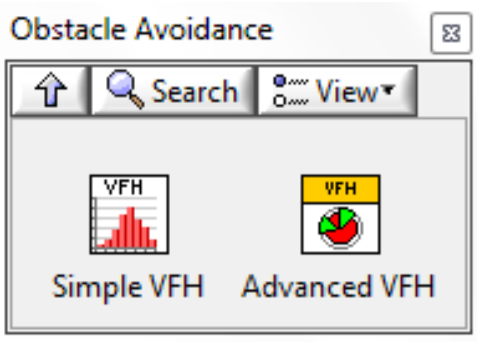


Figura 2.3 Obstacle Avoidance VIs

2.1.1.3 Path Planning VIs

Se utiliza los VIs de planeación de trayectorias para calcular una trayectoria hacia un punto de llegada en un mapa que representa el ambiente del robot.

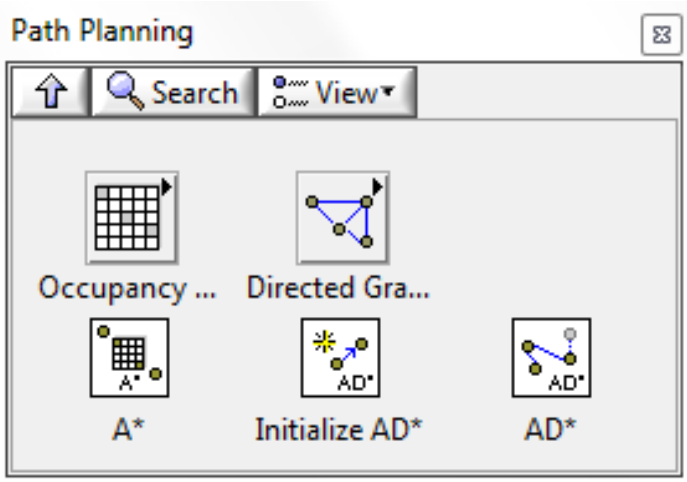


Figura 2.4 Path Planning VIs

2.1.1.4 Protocols VIs

Se utiliza los VIs de protocolos para procesar datos en protocolos de comunicación en LabVIEW, tal como datos enviados por sensores.

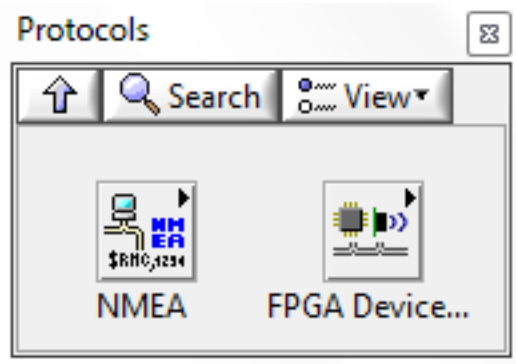


Figura 2.5 Protocols VIs

2.1.1.5 Robotic Arm VIs

Se utiliza los VIs de brazo robótico, para crear e interactuar con un brazo robotico simulado. Se puede trabajar cálculos dinámicos y cinemáticos del brazo, simular el brazo y de esta manera se obtendría el prototipo del mismo.

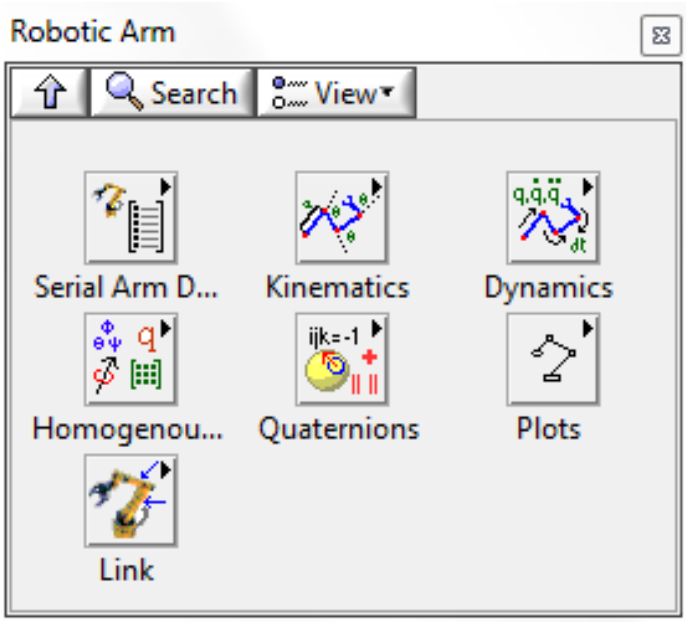


Figura 2.6 Robotic Arm VIs

2.1.1.6 Sensing VIs

Se usa los VIs de sensar para configurar, controlar y recuperar datos de instrumentos comúnmente utilizados en sistemas robóticos, tal como dispositivos seriales y USB.

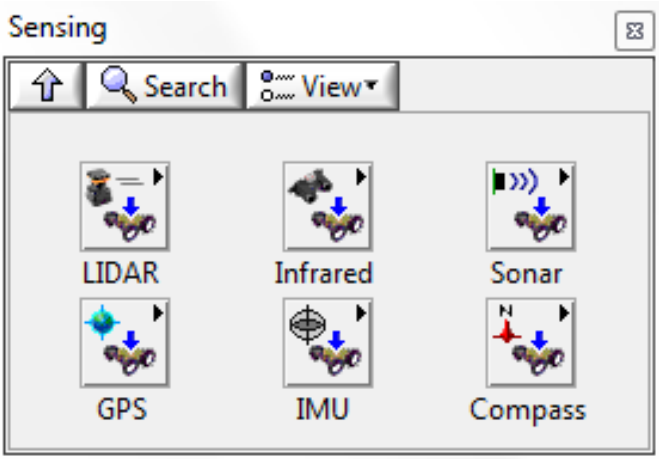


Figura 2.7 Sensing VIs

2.1.1.7 Steering VIs

Se utiliza los VIs de dirección para crear un sistema robot vehículo que consiste en una estructura integrada con dirección definida y ruedas. Se puede calcular y convertir velocidad del robot y rueda y conectar a motores que manejan las ruedas para implementar el control del motor.

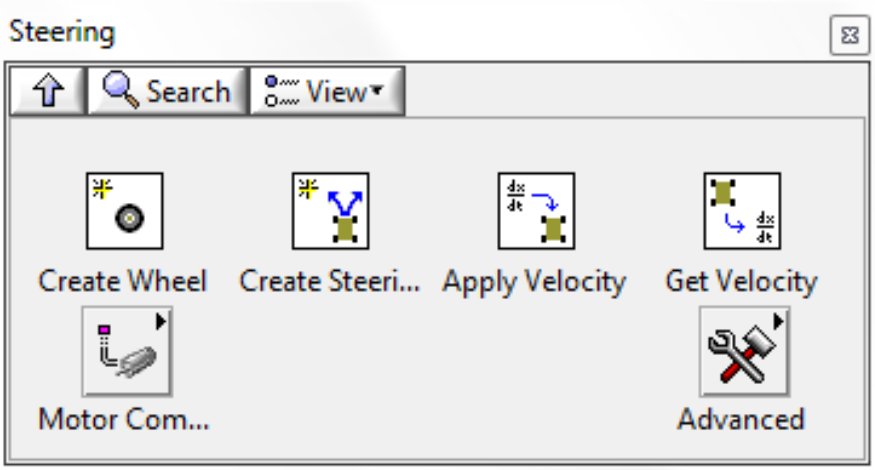


Figura 2.8 Steering VIs

2.1.2 MATHSCRIPT RT MODULE

Con LabVIEW se puede escoger la sintaxis más efectiva para desarrollar algoritmos, explorar señales, procesar conceptos o analizar resultados. Se puede combinar el paradigma de programación gráfica de LabVIEW con LabVIEW

MathScript que viene a ser un lenguaje de programación textual orientada a la Matemática.

Se puede trabajar a través de dos interfaces, LabVIEW MathScript Interactive Window o a través del MathScript Node.

2.1.2.1 LabVIEW MathScript Interactive Window

Se usa esta ventana para editar y ejecutar comandos matemáticos, crear códigos y observar representaciones numéricas y gráficas de variables. La ventana genera salidas y mantiene un historial de comandos utilizados, lista de variables definidas y un display de variables que se pueden seleccionar.

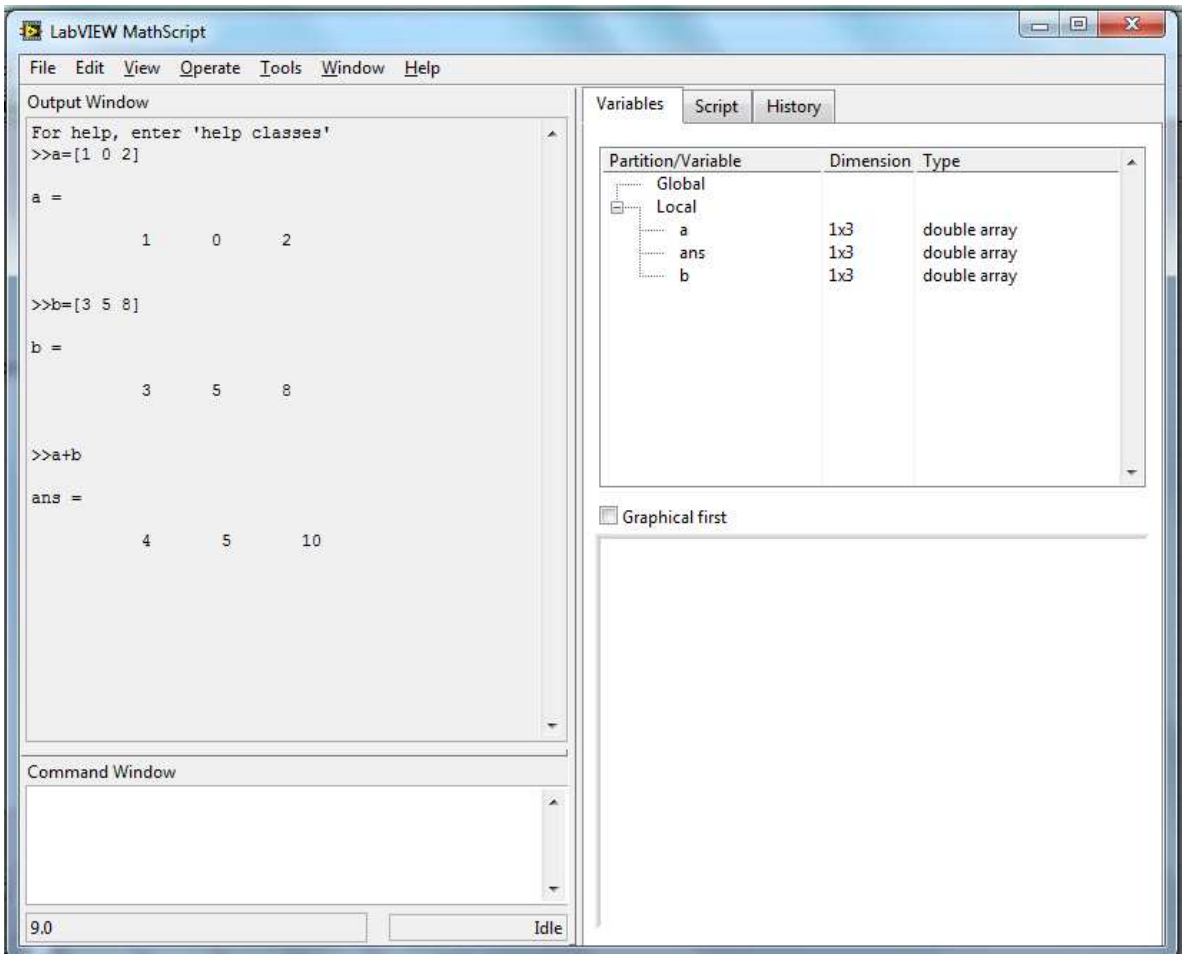


Figura 2.9 MathScript Window

Como se puede observar en la Figura 2.9 el MathScript Window es similar a trabajar con una ventana de Matlab.

2.1.2.2 MathScript Node

Se utiliza esta interfaz de programación para insertar algoritmos textuales (textos de archivo .m) en un VI y de esta manera se introduce este tipo de lenguaje en el ambiente de programación gráfica del LabVIEW.

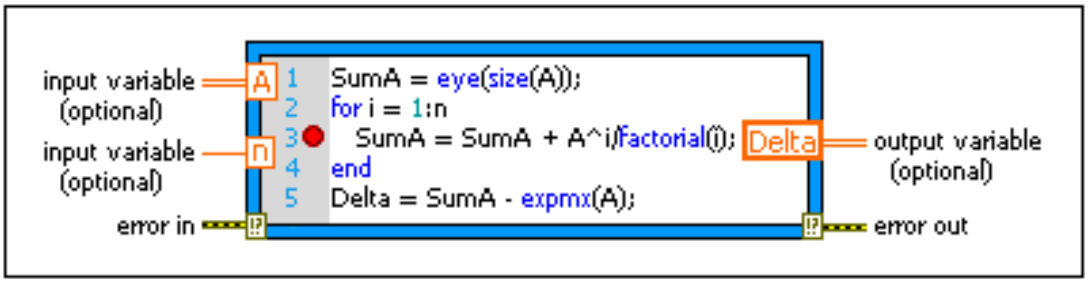


Figura 2.10 MathScript Node (Región rectangular azul)

Existe una gran variedad de funciones textuales pertenecientes al MathScript RT Module, algunas de ellas fueron utilizadas para el desarrollo del programa de este proyecto, mismas que serán detalladas más adelante.

2.1.3 ROBOTINO LABVIEW DRIVER

La plataforma móvil Robotino® no pertenece a hardware de National Instruments, por lo que es necesario hacer uso de una librería exclusiva para la programación del mismo en LabVIEW.

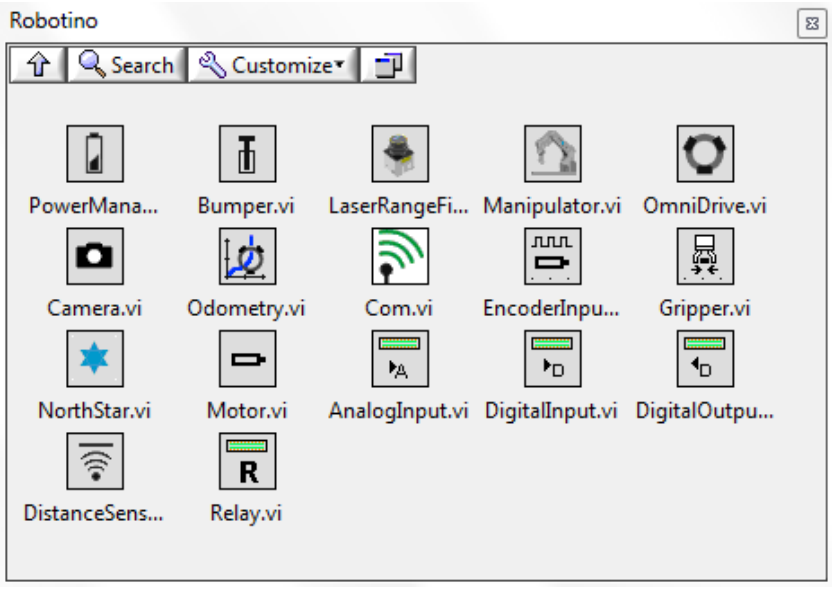


Figura 2.11 Librería de Robotino para LabVIEW

Aquí se encuentran los distintos VIs necesarios para la programación de cada uno de los componentes de Hardware de Robotino®. Así mismo, no todos los subVIs son utilizados, por lo que aquellos empleados en el programa se detallan posteriormente.

2.2 DESARROLLO DEL PROGRAMA

El desarrollo del programa tiene varias etapas, como primer paso se tiene la adquisición de datos, es decir el ingreso del mapa de entorno del robot al LabVIEW, después se tiene una etapa de procesamiento de estos datos donde se obtienen los puntos que conforman los bordes de los obstáculos pertenecientes al mapa de entorno. Se halla el diagrama de Voronoi en base a los puntos hallados y seguido se procede a discriminar los puntos del diagrama generado que no corresponden a lo requerido, por último, en base al diagrama de Voronoi final se obtiene la trayectoria óptima a través del algoritmo A* y la misma es descargada en la plataforma móvil Robotino® de Festo.

Figura 2.12 Diagrama de bloques de la programación

A continuación se presenta en detalle la programación en cada etapa así como una descripción de los algoritmos implementados.

2.2.1 MAPA DE ENTORNO

Este proyecto se aplica para mapas de entorno conocidos, por lo que los obstáculos llegan a ser los datos a ser ingresados al programa y luego analizados para hallar el diagrama de Voronoi correspondiente al mapa de entorno del cual se dispone.

Los obstáculos son representados como polígonos, teniendo presente que de ser un obstáculo irregular se realizará la aproximación a un polígono de máximo 5 vértices.

Se ingresan entonces las coordenadas de los puntos de los vértices correspondientes a cada polígono (obstáculo), incluyendo los límites del mapa en un array. Se ingresan los obstáculos representados como coordenadas de puntos debido a que más adelante se hará uso de la función *voronoi* perteneciente al MathScript Node y la misma trabaja solo con puntos.

Hasta aquí se ha ingresado los vértices de los polígonos, a continuación se rotan todos los puntos ingresados un cierto ángulo para lo cual se creó el subVI *Rotar*.

2.2.1.1 SubVI Rotar

Se hace uso de este subVI debido a que en la obtención de trayectorias, realizada más adelante a través del algoritmo A*, se generaban grandes problemas el momento en que se dispone cierta cantidad de puntos cuya componente en y era la misma en todos ellos, es el caso del borde del mapa de entorno, por lo que al rotar un ángulo despreciable los puntos se elimina esta restricción de tener la misma componente en y en los puntos de los obstáculos.

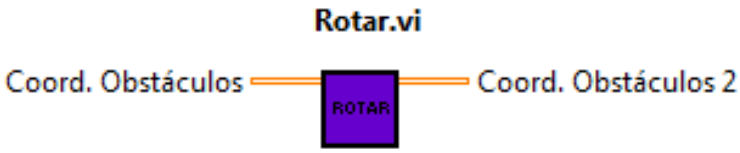


Figura 2.13 SubVI Rotar

El dato de entrada al subVI son los datos iniciales, es decir los vértices de los obstáculos ingresados inicialmente. A la salida se obtiene igualmente un array con las coordenadas de los puntos ya rotados, es decir, simplemente se cambió la referencia de los puntos.

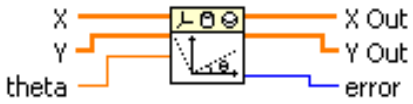


Figura 2.14 Función 2D Cartesian Coordinate Rotation

Para la rotación, se utilizó la función *2D Cartesian Coordinate Rotation* vista en la Figura 2.14, en la cual se ingresa una matriz con las coordenadas en x de los puntos, otra con las coordenadas en y y el ángulo de rotación θ en radianes, a la salida se obtienen 2 arrays con las coordenadas en x y en y respectivamente ya rotadas el ángulo ingresado en sentido contrario a las manecillas del reloj.

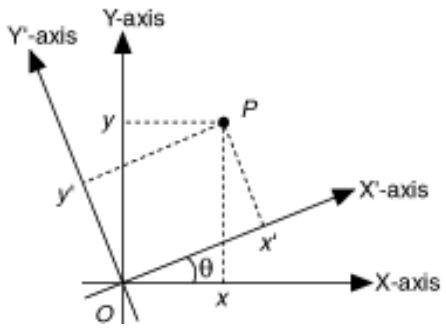


Figura 2.15 Detalles de la rotación

Una vez realizado este proceso, es necesario disponer de más puntos pertenecientes al contorno de los obstáculos para de esta manera tener más datos con los cuales poder hallar un apropiado Diagrama de Voronoi. Para esto se creó el subVI *Coordenadas_Obstáculos.*

2.2.1.2 SubVI Coordenadas_Obstáculos

En este subVI ingresan como datos el array en el cual se ingresaron las coordenadas de los vértices de los obstáculos ya rotados y el número de obstáculos.

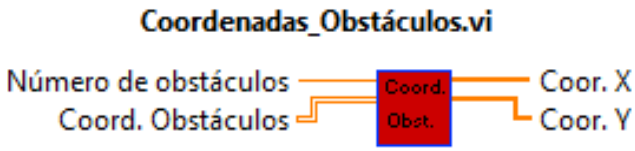


Figura 2.16 SubVI Coordenadas_Obstáculos

La finalidad de este subVI es obtener más puntos pertenecientes a los obstáculos, como se muestra en la Figura 2.16 a la salida se obtienen las coordenadas en x y en y de estos puntos generados. Para esto se utiliza la función *linramp* del MathScript Node.

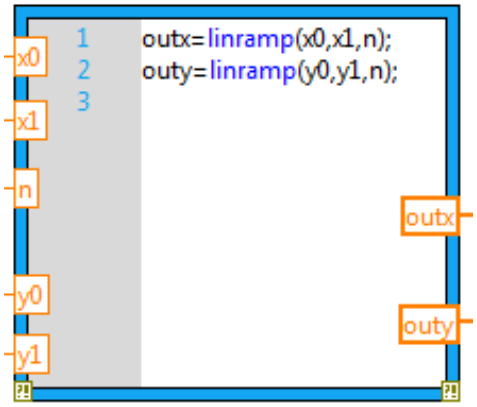


Figura 2.17 MathScript Node utilizando la función linramp

En la Figura 2.17 se puede observar uno de los MathScript Node pertenecientes al subVI, el objetivo es ir tomando 2 de los vértices correspondientes a cada arista de cada polígono representados por $P_0(x_0, y_0)$ y $P_1(x_1, y_1)$. La función *linramp* lo que hace es proporcionar un vector que contiene *n* valores dentro de los rangos asignados. Por ejemplo, *outx* es un vector con *n* valores comprendidos entre x_0 y x_1. Y de esta manera se obtienen todos los puntos requeridos para continuar con la programación.

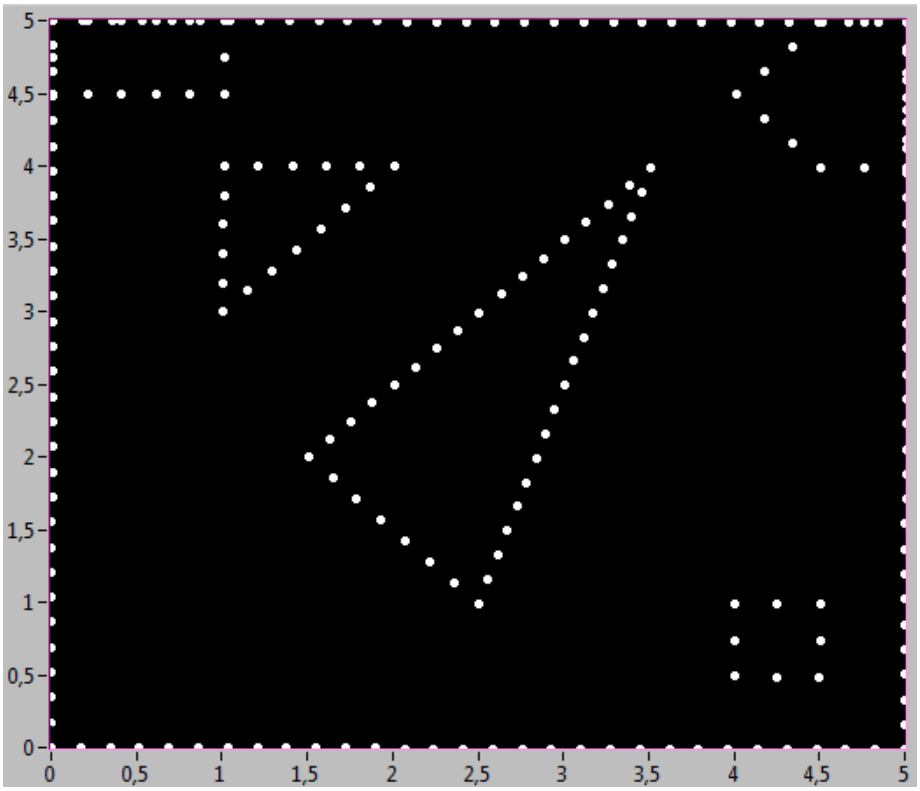


Figura 2.18 Puntos pertenecientes al contorno de los ángulos rotando puntos

Como se puede observar en la Figura 2.18, aparentemente no existe una rotación de los puntos del mapa de entorno debido a que el ángulo de rotación es mínimo y se lo va a considerar despreciable.

Todo este proceso que comprende el ingreso de los vértices y la generación de los puntos adicionales pertenece a una etapa inicial, una vez concluida se da paso a la siguiente etapa de la programación.

2.2.2 DIAGRAMA DE VORONOI

Una vez que se dispone de un vector para coordenadas de los puntos en x y otro para y, se ingresan los mismos en un MathScript Node y se hace uso de la función *voronoi.*

Esta función entrega 2 matrices vx y vy, siendo vx una matriz de 2x*n* de números reales, donde *n* es el número de puntos de las coordenadas en x del diagrama de Voronoi. Tiene 2 filas debido a que entrega dos valores: $x(k)$ y $x(k+1)$. Lo mismo ocurre para vy. LabVIEW define al diagrama de Voronoi como todas las rectas conformadas por $(x(k), y(k))$ a $(x(k+1), y(k+1))$ mismos que pertenecen a vx y vy.

Se debe tomar en cuenta, que esta función que proporciona el MathScript Node solo halla voronoi para datos de entrada representados como puntos.

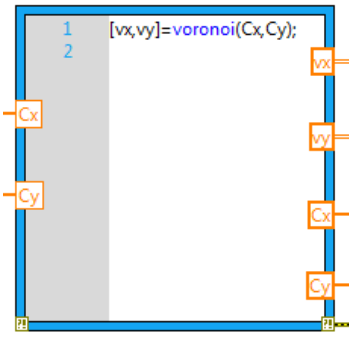


Figura 2.19 MathScript Node programado para hallar Diagrama de Voronoi

A partir de estas dos matrices vx y vy, se hace uso de 2 subVIs para hallar el diagrama de voronoi del mapa de entorno ingresado.

2.2.2.1 SubVI Inside

Anteriormente se mencionó que el algoritmo *voronoi* que proporciona el MathScript Node entrega los puntos de las rectas del diagrama de Voronoi en base a datos ingresados como puntos.

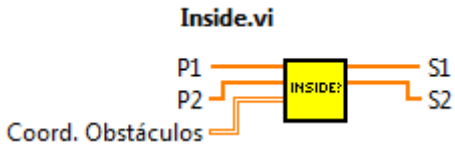


Figura 2.20 SubVI Inside

Con el objetivo de realizar una primera discriminación de puntos, es decir, descartar los puntos que se encuentran fuera de los límites del mapa de entorno, se creó el subVI Inside.

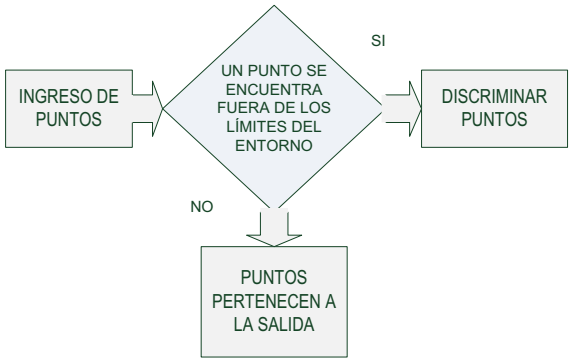


Figura 2.21 Diagrama de bloques del subVI Inside

En la entrada del subVI se ingresan los arrays de los puntos P_1 y P_2 pertenecientes a las rectas que conforman el diagrama de Voronoi inicial así como las coordenadas de los obstáculos rotadas. A la salida se obtienen solamente aquellos puntos P_1 y P_2 que se encuentra dentro de los límites del entorno, mientras que los que se encuentran fuera simplemente son descartados. Para ello se hace uso de la función *is_inpolygon* del MathScriptNode misma que se detalla más adelante en el posterior subVI.

Como se muestra en la Figura 2.22 se puede apreciar claramente que una vez realizada la primera discriminación, el diagrama de Voronoi está compuesto solo por puntos que se encuentran dentro de los límites del entorno.

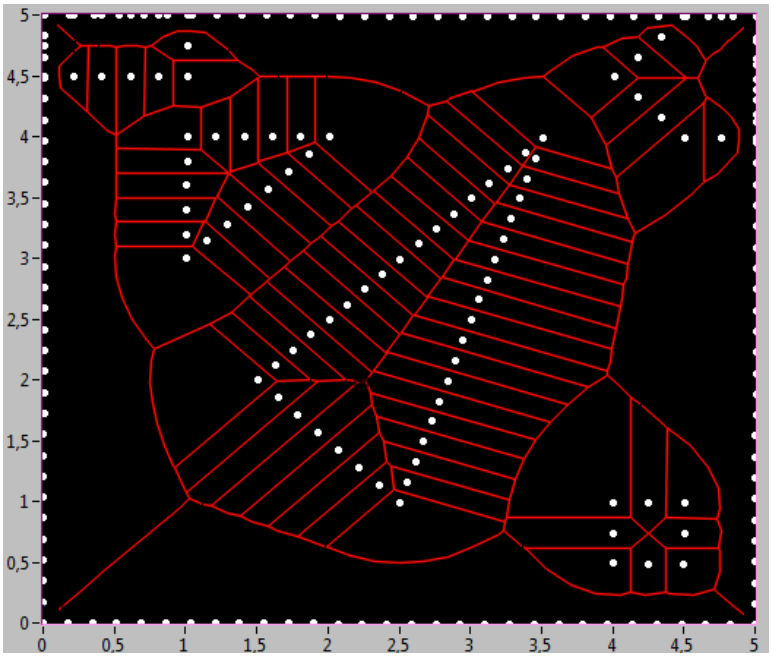


Figura 2.22 Diagrama de Voronoi después de de utilizar el subVI Inside

Para la siguiente etapa, es necesario discriminar los puntos del diagrama que pertenecen a los bordes o se hallan dentro de los obstáculos.

2.2.2.2 SubVI Discrim_puntos

El diagrama de Voronoi obtenido por el momento, no es el que se requiere para obtener la ruta más segura para el robot, claramente se puede observar que se considera una ruta segura a rectas que pertenecen a los obstáculos, por esta razón es necesario discriminar estos puntos de estas rectas. Para esto se desarrolló un subVI denominado *Discrim_puntos*.

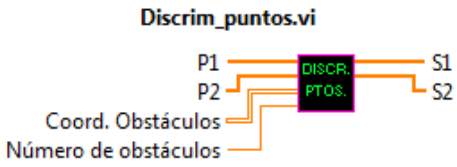


Figura 2.23 SubVI Discrim_puntos

En este subVI ingresan las salidas del subVI *Inside*, es decir los puntos del diagrama de Voronoi que pertenecen al mapa de entorno y no se hallan fuera de los límites del mismo. La lógica empleada para la programación de este subVI se puede observar en la Figura 2.24.

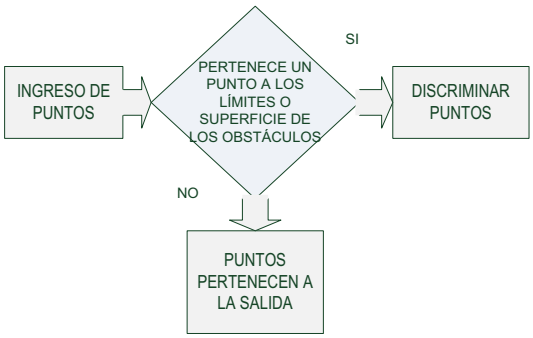


Figura 2.24 Diagrama de la programación del subVI Discrimin_puntos

Para determinar si un punto es parte del contorno y/o de la superficie de uno de los polígonos se utilizó la función *is_inpolygon* del MathScript Node. Esta función requiere de 4 entradas:

- x_1, especifica un vector de números reales que representan las coordenadas en x de los puntos que se van a analizar.
- y_1, especifica un vector de números reales que representan las coordenadas en y de estos puntos, se debe tener en cuenta que los vectores x_1 y y_1 deben ser de la misma dimensión.
- Px, especifica un vector de números reales que representan las coordenadas en x de los vértices del polígono.
- Py, especifica un vector de números reales que representan las coordenadas en y de los vértices del polígono, la dimensión de este debe ser la misma que de Px.

Así mismo proporciona de 2 salidas dependiendo de la aplicación requerida:

- a, da un vector de números lógicos del mismo tamaño que x_1, si el n punto de (x_1, y_1) se encuentra en la superficie o en el contorno del polígono, entonces el n elemento de a es 1. Caso contrario, el n elemento de a es 0.
- b, proporciona un vector de números lógicos del mismo tamaño que x_1, si el n punto de (x_1, y_1) pertenece al contorno del polígono, entonces el n elemento de b es 1. Caso contrario, el n elemento de b es 0.

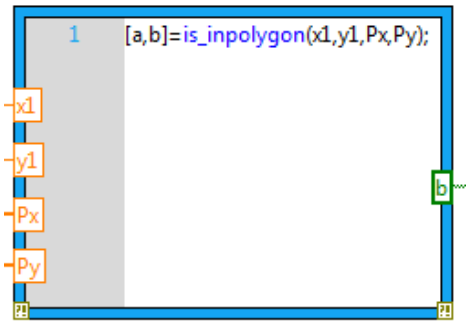


Figura 2.25 MathScript Node utilizando la función is_inpolygon

En el ejemplo presentado en la Figura 2.25 el único dato de salida que se necesita es b, debido a que el MathScript Node del ejemplo es el caso para discriminar los puntos del contorno del mapa de entorno, mas no de los obstáculos en sí.

A partir de esta lógica, se van almacenando en su array respectivo, aquellos puntos que no se hallan en los contornos y/o superficies de los polígonos (obstáculos) y estos vectores llegan a constituir los puntos pertenecientes a las rectas que en conjunto constituyen el diagrama de Voronoi del mapa de entorno dado.

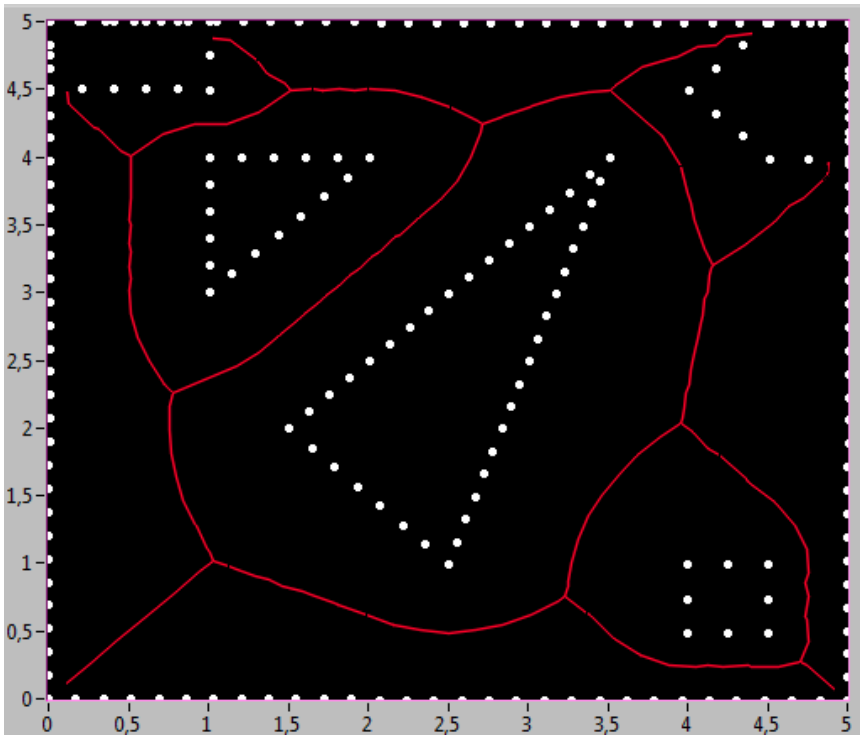


Figura 2.26 Diagrama de Voronoi del Mapa de entorno actual

Una vez concluida esta etapa, se obtiene el diagrama de Voronoi final (líneas de color rojo), tal cual se muestra en la Figura 2.26, éste representa la ruta más segura por la cual el robot va a desplazarse dentro de todo el ambiente.

A continuación es necesario determinar la trayectoria más óptima en base a un punto de llegada y salida, pertenecientes al diagrama de Voronoi final.

2.2.3 GENERACIÓN DE LA TRAYECTORIA

Para la obtención de la trayectoria se sigue el esquema basado en la Figura 2.27.

Figura 2.27 Diagrama de programación para la generación de la trayectoria

2.2.3.1 Creación del Mapa

A partir de este momento se hace uso del NI LabVIEW Robotics Module. Para crear un Mapa se usa uno de los subVI perteneciente a la paleta Robotics>Path Planning>Directed Graph Map>Create Directed Graph Map.

Figura 2.28 Create Directed Graph Map

Esta función permite generar un mapa del entorno del robot que consiste de puntos irregularmente espaciados, denominados también como *nodos*. Para la utilización de este subVI la única configuración necesaria es el map reference out el cual es la referencia para los otros subVI pertenecientes a *Path Planning*, el ingreso de nodos se lo hace posteriormente.

2.2.3.2 Ingreso de Nodos al Mapa

Una vez que se ha creado un mapa, se deben ingresar los nodos que van a pertenecer a una de las posibles rutas del robot, es decir, es necesario ingresar

los puntos de las rectas del diagrama de Voronoi final que se obtuvo anteriormente. Para esto es necesario primero darles a los puntos un “nombre”, en otras palabras transformarlo a string, debido a que es el formato con el cual trabaja el mapa creado. Se lo realiza a través del subVI *Nombre nodos*.

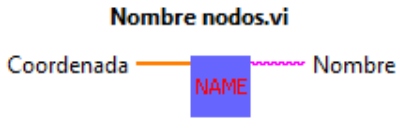


Figura 2.29 SubVI Nombre nodos

Una vez que se le ha asignado un nombre a cada punto perteneciente al diagrama de Voronoi, se ingresa los nodos al mapa a través de la función que se encuentra en la paleta Robotics>Path Planning>Directed Graph Map>Add Node.

Figura 2.30 Add Node

Como se observa claramente en la Figura 2.30, esta función añade un único punto o nodo al mapa en las coordenadas que se especifiquen. Como salida proporciona la referencia del mapa de salida la cual servirá para las configuraciones posteriores.

Si el mapa ya contiene un nodo con el mismo nombre, este VI no añade ese nuevo nodo al mapa.

2.2.3.3 Ingreso de la Distancia Euclídea

El diagrama de Voronoi se encuentra conformado por segmentos resultante de la unión de 2 puntos. Es necesario conocer la distancia entre los mismos para poder determinar el “coste” de la trayectoria, misma que requerirá el algoritmo A* para obtener la trayectoria más óptima.

Sea $P_1(x_1, y_1)$ y $P_2(x_2, y_2)$ dos de los puntos que conforman una de las rectas del diagrama, la distancia euclídea entre los mismos viene determinada por la ecuación:

$$d.e. = \sqrt{((x_2 - x_1)^2 + (y_2 - y_1)^2} \quad (2.1)$$

Se ingresa los 2 puntos pertenecientes a cada recta al mapa a través de la función que se encuentra en la paleta Robotics>Path Planning>Directed Graph Map>Enqueue Change to Edge Cost.

Como se muestra en la Figura 2.31 cada punto de cada segmento representa un nodo de partida y un nodo de llegada, se ingresa 2 veces la recta, es decir, el nodo de partida pasa a ser de llegada y viceversa, para poder tener un sentido bidireccional el momento del cálculo de la trayectoria por parte del algoritmo A*. *Cost* no es más que la distancia euclídea entre estos 2 puntos.

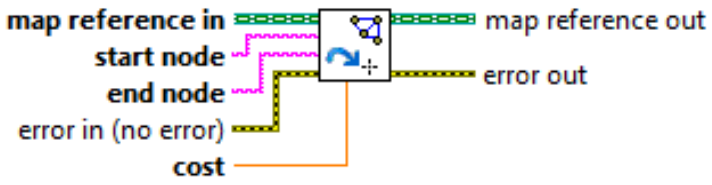


Figura 2.31 Enqueue Change to Edge Cost

2.2.3.4 Ingreso de los nodos de partida y llegada

Los nodos tanto de partida como de llegada van a ser controlados por cursores, mismos que son inicializados en una posición aleatoria. Una vez generado el diagrama de Voronoi final, se posicionará a los cursores acorde a donde se quiera localizar la referencia de partida y llegada de la trayectoria.

Para ingresar estos nodos se hace uso de la función localizada en la paleta Robotics>Path Planning>Directed Graph Map>Get Node Reference. Igualmente es necesario darle un "nombre" al nodo a través del subVI *Nombre nodos* antes de ingresar los nodos a esta función.

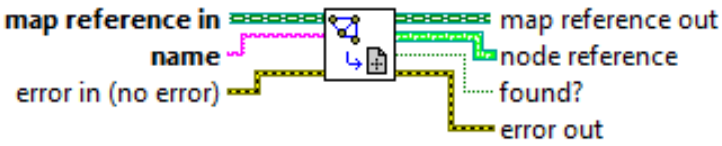


Figura 2.32 Get Node reference

Como se observa en la Figura 2.32, se obtiene un nodo de referencia el mismo que será de utilidad el instante de utilizar el algoritmo A*. Es necesario obtener 2 nodos de referencia, el inicio y el final de la trayectoria.

2.2.3.5 Algoritmo A*

En los puntos anteriores, se creó un mapa, se ingresaron los nodos pertenecientes a este, y se obtuvieron las referencias de coste de los segmentos así como los puntos de partida y llegada. Finalizado todos estos puntos, se cuenta con los datos necesarios para que el algoritmo A* obtenga la trayectoria más óptima es decir la de menor coste.

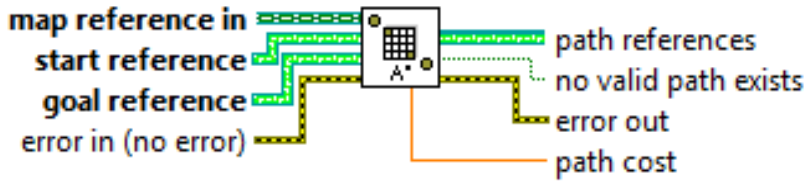


Figura 2.33 Algoritmo A*

En la Figura 2.33 se puede observar que a esta función ingresan la referencia del mapa, misma que contiene la información de los nodos y el coste de cada segmento, la referencia del nodo de partida y del nodo de llegada. A la salida proporciona una referencia de la trayectoria basada en el menor coste. Se debe tener en cuenta que esta función se utiliza para mapas de referencia estática, es decir, donde el entorno no va a cambiar.

2.2.3.6 Obtener nodos de la trayectoria

Como el algoritmo A* genera una referencia, para disponer de los nodos de la trayectoria, tanto de sus nombres como de sus coordenadas, se utiliza la función localizada en la paleta Robotics>Path Planning>Directed Graph Map>Get Nodes in Path.

Figura 2.34 Get Nodes in Path

De esta manera concluye la generación de la trayectoria, con un array que contiene los puntos de la misma los cuales pueden ser analizados y/o tratados acorde a la aplicación requerida, de igual manera como en un inicio se creo el mapa, es necesario cerrarlo.

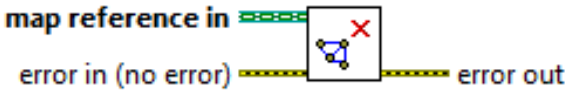


Figura 2.35 Close Directed Graph Map

En la Figura 2.36 se puede observar que la trayectoria más óptima desde los puntos de partida y llegada, indicados en el mapa, es la que se muestra de color verde.

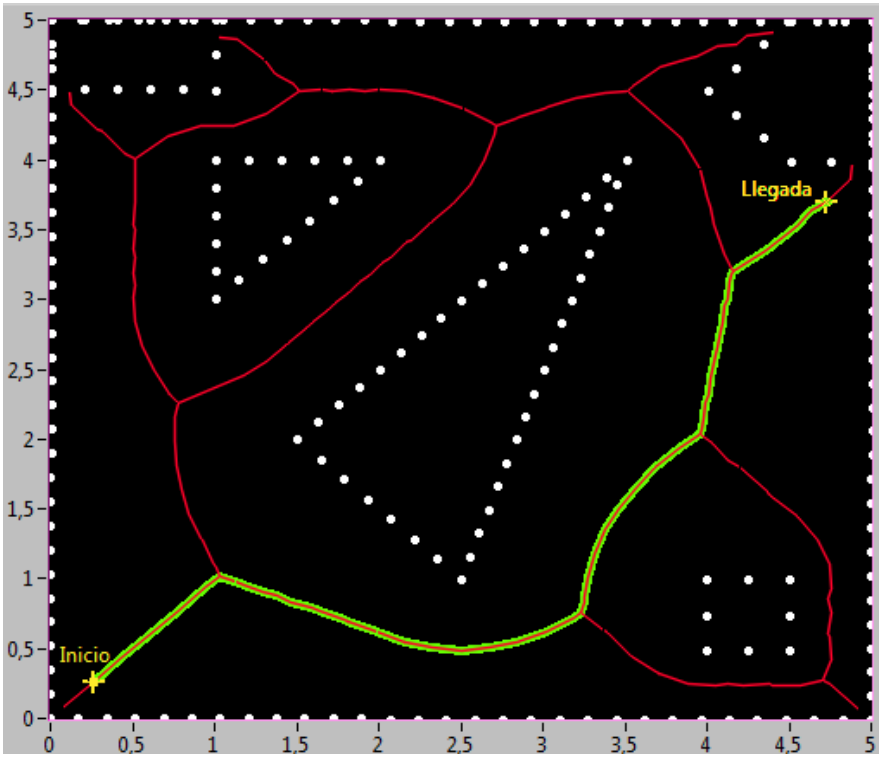


Figura 2.36 Trayectoria óptima acorde al mapa de entorno basada en diagramas de Voronoi

2.2.4 PROGRAMACIÓN DE ROBOTINO®

Una vez que se obtiene la trayectoria óptima a seguir, es necesario descargarla en la plataforma móvil Robotino® para que físicamente el robot siga esta trayectoria y de esta manera comprobar los algoritmos implementados.

Los datos con los que se dispone están en un array, mismo que contiene los puntos pertenecientes a las rectas que conforman la trayectoria a seguir por el robot.

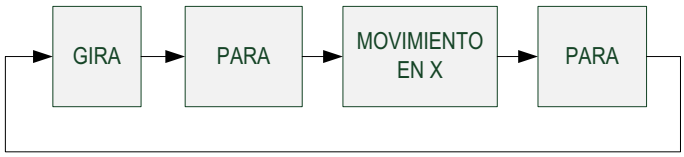


Figura 2.37 Lógica de movimientos de Robotino

Para que el robot pueda seguir la trayectoria, se utilizó la lógica de movimientos presentada en la Figura 2.37.

Como primer paso es necesario que el robot gire hasta tomar la posición de la recta, una vez que el robot se encuentra orientado de frente, se mueve la distancia necesaria para completar el seguimiento de la recta y el robot vuelve a girar para tomar posición de la siguiente recta a seguir. Se repite la misma lógica hasta recorrer todas las rectas que conforman la trayectoria.

2.2.4.1 Programación para "Girar"

Para que el robot cumpla con el primer paso que es girar sobre su propio eje, es necesario conocer el ángulo y sentido de giro.

Para obtener estos datos, se utiliza una lógica de cambio de referencias, es decir, es necesario desplazar y rotar los ejes cada vez que se va a seguir una nueva recta, para de esta manera poder hallar fácilmente el ángulo de giro.

Con el objeto de desplazar los ejes, se hace uso de la función *2D Cartesian Coordinate Shift.*

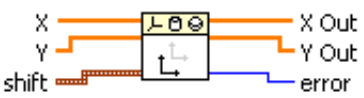


Figura 2.38 Función 2D Cartesian Coordinate Shift

Como se observa en la Figura 2.38, a la entrada de esta función se ingresa una matriz con las coordenadas en x de los puntos a ser desplazados, otra con las

coordenadas en y y un cluster compuesto por dx y dy, siendo éstos, los valores de desplazamiento en x y en y respectivamente. A la salida se obtienen 2 arrays con las coordenadas de los puntos ya desplazados.

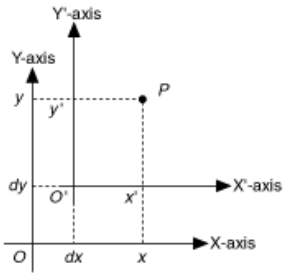


Figura 2.39 Detalles del desplazamiento de ejes

El objetivo es hacer que con el desplazamiento, el primer punto de la recta sea $P_1(0,0)$, por lo que las distancias a desplazar en x como en y serán las coordenadas de este primer punto antes del desplazamiento.

Para rotar los ejes, se hace uso de la función *2D Cartesian Coordinate Rotation,* misma que fue descrita anteriormente. El ángulo de giro para la rotación, será el último ángulo de giro que dio el robot si el sentido de giro fue antihorario, o la diferencia entre 2π radianes menos éste último ángulo si el sentido fue horario.

Una vez descrita la lógica del cambio de ejes, en base a la Figura 2.40 se llegó a que las condiciones de programación tanto para el sentido de giro como para el ángulo de rotación son:

- Si x' es de valor *positivo*, el ángulo de giro será α.
- Si x' es de valor *negativo*, el ángulo de giro será 180º- α.
- Si y' es de valor *positivo*, el sentido de giro será anti horario.
- Si y' es de valor *negativo*, el sentido de giro será horario.

Siendo $P_2(x',y')$ el segundo punto que pertenece a la recta a seguir, y el primer punto $P_1(0,0)$. Para hallar α, se utiliza la ecuación:

$$\alpha = \left|\tan^{-1}\left(\frac{y'-0}{x'-0}\right)\right|$$

Téngase en cuenta que es necesario tener el ángulo de giro en grados, por lo que hay que hacer las conversiones del caso siempre que se requiera.

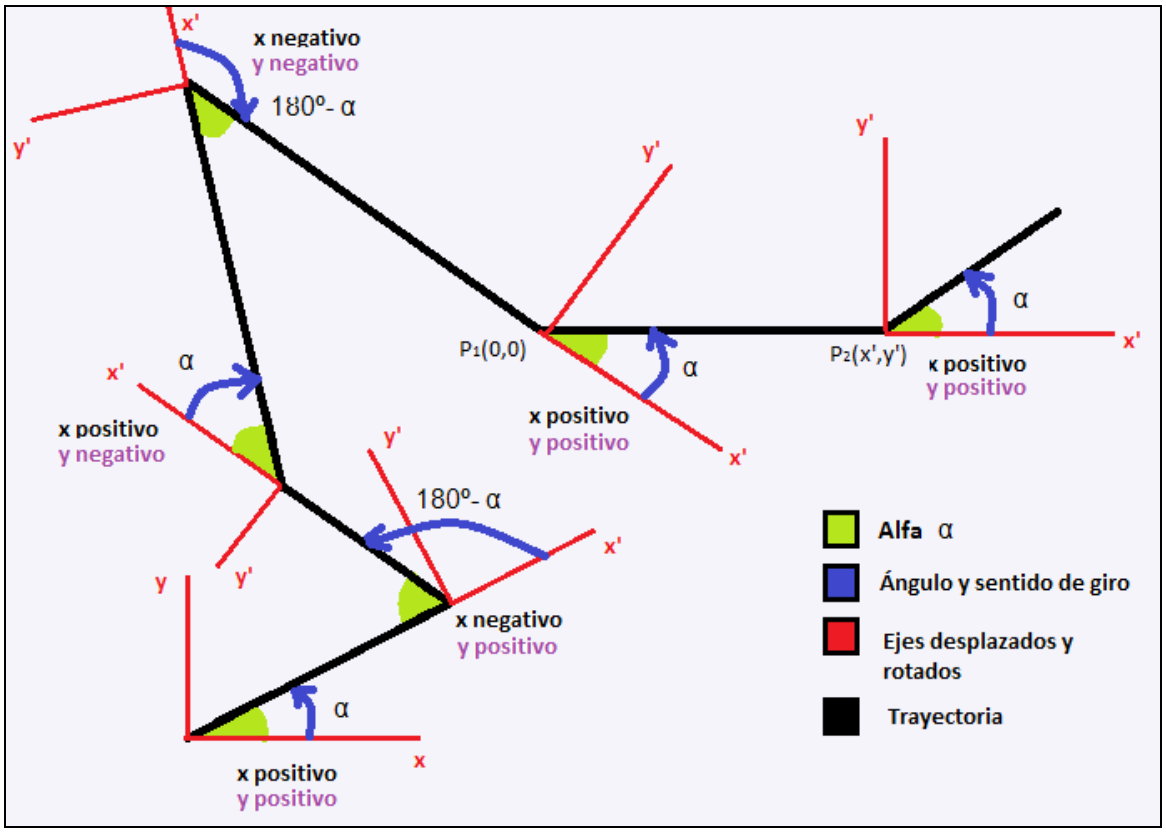


Figura 2.40 Lógica para hallar ángulos y sentido de giro

Una vez que se dispone de los ángulos y el sentido de giro, se procede a enviar este dato al robot para que gire los valores y en el sentido de giro deseado.

Para programar el hardware del Robotino® a través del LabVIEW, se hace uso de una librería exclusiva, misma que contiene todos los VI's necesarios para programar al robot según la aplicación requerida.

Antes que nada es necesario establecer una comunicación con Robotino®, para lo cual se utiliza el *Com.vi.*

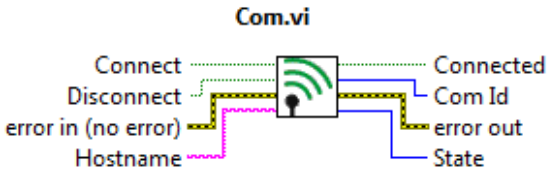


Figura 2.41 Com.vi

Como se puede apreciar en la Figura 2.41, para establecer una comunicación con el robot es necesario ingresar la dirección IP del robot (*hostname*) y dar un valor de verdadero a *connect* para habilitar la comunicación. Como salida proporciona el *Com Id* que es la señal requerida por otros VI's para establecer la comunicación con Robotino. Si se ha establecido una conexión entre el robot y la PC, *Com Id* será un valor igual o mayor que cero, caso contrario si no se ha establecido conexión, el valor por default es -1.

Una vez configurada la comunicación, se puede configurar los demás VI's requeridos. En este caso se quiere que el robot gire sobre su propio eje, un ángulo y en un sentido dado, para esto se usa el *OmniDrive.vi.*

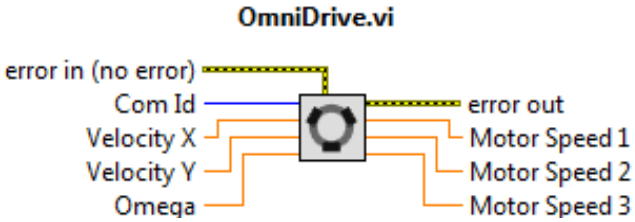


Figura 2.42 OmniDrive.vi

En la Figura 2.42 se puede notar que este VI permite configurar, una velocidad en x y/o una velocidad en y dadas en milímetros por segundo, así como la velocidad angular (Omega). Para este caso, es necesario simplemente configurar a *Omega* con una velocidad dada en grados por segundo. A partir de la velocidad asignada, así como la conexión *del Com Id* de este VI con el *Com Id* que proporciona el *Com.vi*, se obtiene a la salida 3 valores que corresponden a las velocidades en RPM de cada motor para que en conjunto se obtenga la velocidad asignada a *Omega*.

A continuación se configura a los 3 motores del Robotino, para esto se utiliza al *Motor.vi*. En la Figura 2.43 se puede apreciar que este VI dispone en sus entradas del *Com Id* necesario para la comunicación, la velocidad *Speed* en la cual ingresa la velocidad asignada por el *OmniDrive.vi,* el número del motor (0,1 ó 2), además dispone de un freno el cuál cuando toma el valor de verdadero, para a los motores.

Como salidas se tiene la velocidad actual, la corriente del motor y la posición actual, ésta última proporcionada por los encoders y entrega un número que representa la suma de los incrementos según el desplazamiento del motor.

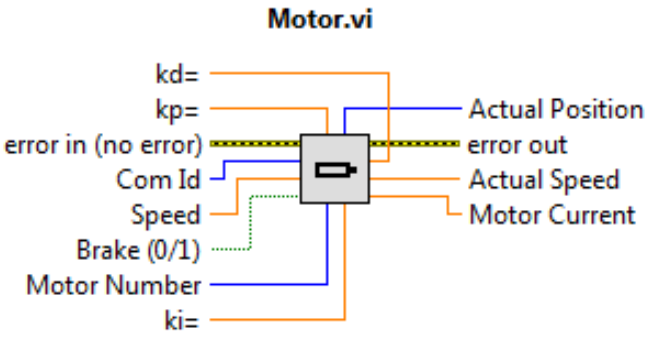


Figura 2.43 Motor.vi

Es importante mencionar que en el *Motor.vi* viene un PID incorporado el cual en teoría puede ser calibrado, pero debido a que éste ocasionó serios problemas en el movimiento de Robotino, el PID fue excluido de este VI a pesar de ser una librería que ya vino establecida.

Finalizada la configuración de los motores, en base a la posición actual se comparan con relaciones previamente calculadas de cuanto deberá girar el robot según el ángulo y el sentido de giro determinados anteriormente. Si se requiere que el sentido de giro sea anti horario, el valor de *Omega* del *OmniDrive.vi* deberá ser positivo, para que el robot gire en sentido horario el valor de *Omega* será negativo.

Una vez que el robot ha girado el ángulo requerido es necesario frenar los motores, para continuar con la lógica de la programación.

2.2.4.2 Programación para "Parar"

Cada vez que el robot ha alcanzado los valores establecidos de desplazamiento, ya sea de giro o de movimiento en x, es necesario detener al robot para que pueda seguir con el ciclo de movimientos para completar la trayectoria.

Aquí simplemente se configura al *Motor.vi* con una velocidad de cero y asignando un valor de verdadero a los frenos de cada motor.

2.2.4.3 Programación para “Mover en x”

La distancia a recorrer se calcula a través de la distancia euclídea. Así mismo es necesario configurar el *OmniDrive.vi* con una velocidad en x, dada en milímetros por segundo y a la salida se tendrá los datos de las velocidades para cada motor.

Se configuran los motores de la misma manera que se lo hizo en el caso anterior, teniendo en cuenta que hay que desactivar los frenos del motor dando un valor de falso a los mismos y así mismo en base a la señal de los incrementos de los encoders se hace que el robot recorra la distancia requerida.

Como se mencionó en un inicio, se repite este mismo ciclo (girar, parar, mover en x, parar) hasta que el robot haya terminado de recorrer todas las rectas que conforman la trayectoria. Concluidos todos los desplazamientos, es necesario finalizar la conexión de Robotino® asignando un valor de verdadero a *Disconnect* del *Com.vi.*

Con esto concluye el segundo capítulo, mismo que abarca toda la programación del proyecto en LabVIEW. En el capítulo siguiente, se presentan las pruebas y resultados obtenidos de todo el desarrollo del proyecto.

CAPÍTULO 3

PRUEBAS Y RESULTADOS

En este capítulo se detalla el manejo y funcionamiento del proyecto, así como las pruebas realizadas y los resultados obtenidos.

3.1 CONEXIÓN CON ROBOTINO®

Para encender al Robotino® se debe presionar el pulsador On/Off hasta que el LED se encienda, el display se enciende, aparecen dos barras que cruzan todo el ancho de la pantalla, esto es señal de que el PC del Robotino® está arrancando. Tras unos 30 segundos, el display muestra: el canal al que se encuentra conectado el Robotino®, su dirección IP: 172.26.201.2, el nombre de la red: Robotino.005.106 y en la última línea muestra una barra indicando el estado de carga de las baterías así como la versión del robot: v2.0 (Figura 3.1). El Robotino® se halla ahora preparado para funcionar.

Figura 3.1 Display del Robotino®

Si no se acciona ninguna tecla durante 10 segundos, se apaga la iluminación del display para mantener el consumo de corriente lo más bajo posible durante el funcionamiento. Para reactivar el display se pulsa una de las teclas de flecha, no se debe presionar la tecla de Enter para activar el display, con el fin de evitar un arranque no deseado de, por ejemplo, la ejecución de un programa de demostración.

Para configurar la conexión de Robotino a través de una WLAN, se siguen los siguientes pasos:

1. Activar la WLAN, la configuración de la WLAN debe permitir la asignación de una dirección IP para la WLAN. Solo entonces puede establecerse una conexión entre la red y el punto de acceso en el Robotino®.
2. Después de poner en marcha el Robotino® y esperar el proceso de arranque, se toma nota de la dirección IP que se muestra en el display, en este caso la dirección es: 172.26.201.2.
3. Permitir que la WLAN busque dispositivos inalámbricos dentro del alcance, entonces aparecerá una red con el nombre Robotino.005.106 en la lista de redes disponibles, tal como se muestra en la Figura 3.2.

Figura 3.2 Redes inalámbricas disponibles a través de la WLAN

4. Establecer la conexión con la red Robotino.005.106 si ello no ha sido realizado a través del software de red.

Para una conexión correcta, es indispensable verificar que se tengan los siguientes ajustes en la red:

- Asignar automáticamente una clave de red (SSID).
- Obtener una dirección IP automáticamente.

Ambos ajustes deben estar activos para poder establecer una conexión con el Robotino®.

Si se desea verificar la conexión WLAN, se lo puede hacer a través del MS-DOS, siguiendo los pasos:

1. Abrir la ventana de comandos MS-DOS, la cual se encuentra en Inicio>Programas>Accesorios>Símbolo del sistema. O alternativamente se puede escribir la orden *cmd* en Inicio>Ejecutar.
2. Escribir en la ventana que se ha abierto, la orden: *ping 172.26.201.2*, seguida de la tecla *Enter.*
3. De haberse establecido una conexión WLAN, se recibirán los mensajes mostrados en la Figura 3.3.

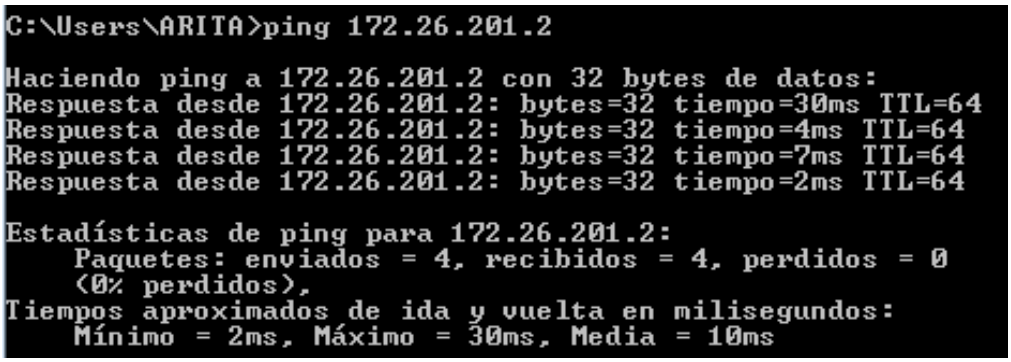


Figura 3.3 Conexión establecida con Robotino

4. Si no existe conexión con el Robotino, aparecerán los mensajes mostrados en la Figura 3.4.

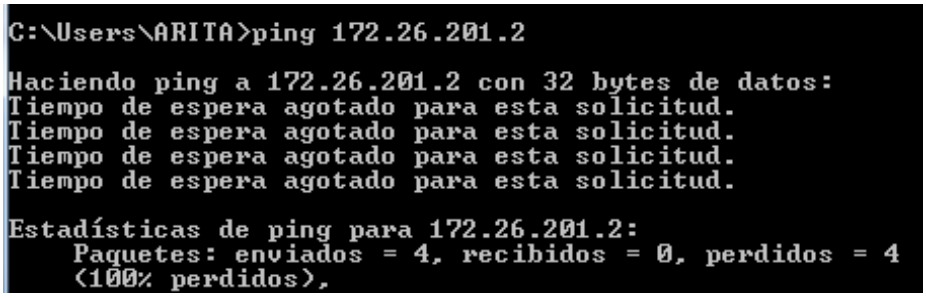


Figura 3.4 Conexión no establecida con Robotino

Cabe mencionar que Robotino® presenta dos modalidades de funcionamiento en cuanto a conexión: *AP* (Access Point) y *Client*. A través del modo *AP* se pueden

controlar de 1 hasta 4 Robotinos independientemente de manera segura. El modo Client por su parte, permite trabajar con cualquier número de Robotinos en una red, asignando una dirección IP diferente para cada robot. En el caso de este proyecto, el modo a utilizarse es *AP* debido a que se trabaja con un solo Robotino®.

Una vez que se ha establecido la conexión con el Robotino®, se procede a hacer uso de la HMI desarrollada en LabVIEW, como información adicional es importante mencionar que el suelo por el que debe desplazarse el Robotino® debe ser plano y bien nivelado, de esta manera, pueden ejecutarse correctamente los movimientos deseados. El suelo debe estar seco y limpio para evitar cualquier daño a los componentes mecánicos y electrónicos. Además es importante asegurarse que las baterías se encuentren cargadas completamente para un correcto funcionamiento del robot.

3.2 FUNCIONAMIENTO DE LA HMI

Al iniciar la HMI se mostrará una ventana de presentación tal como se muestra en la Figura 3.5.

Figura 3.5 Ventana de presentación de la HMI

Se procede a dar clic en INICIAR para empezar con el funcionamiento de la HMI, aparecerá una ventana como se muestra en la Figura 3.6.

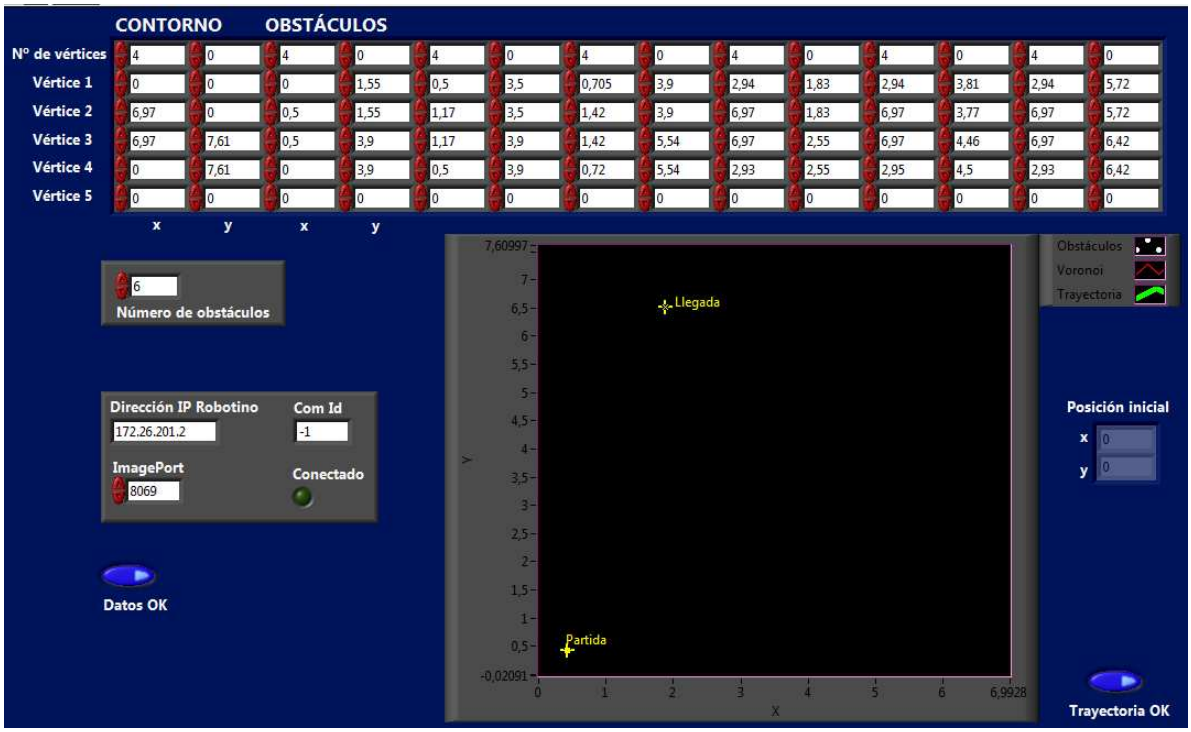


Figura 3.6 Ventana de trabajo de la HMI

3.2.1 INGRESO DE DATOS

Una vez que se ingresa a la HMI desarrollada en LabVIEW, se debe ingresar todos los datos necesarios para el funcionamiento de la misma. Éstos se detallan a continuación.

3.2.1.1 Ingreso de Obstáculos

Se debe seleccionar el número de obstáculos que contiene el mapa de entorno, aparecerá un array con el número de filas y columnas acorde al número de obstáculos tal como se muestra en la Figura 3.7. Anteriormente se mencionó que los datos que se ingresan son las coordenadas de los vértices de los obstáculos representados por polígonos de máximo 5 vértices. Téngase en cuenta que las coordenadas deben ser ingresadas en metros.

A la primera y segunda columna le corresponden los datos necesarios para el ingreso del contorno, por ejemplo de darse un entorno cuadrangular de 5x5 [m],

es decir un polígono de 4 vértices, en la primera fila, primera columna se ingresa el número de vértices es decir 4, posteriormente a partir de la segunda fila se ingresa de vértice en vértice las coordenadas en x y en y en la primera y segunda columna respectivamente. Como no posee un quinto vértice, sencillamente no se llena ninguna coordenada.

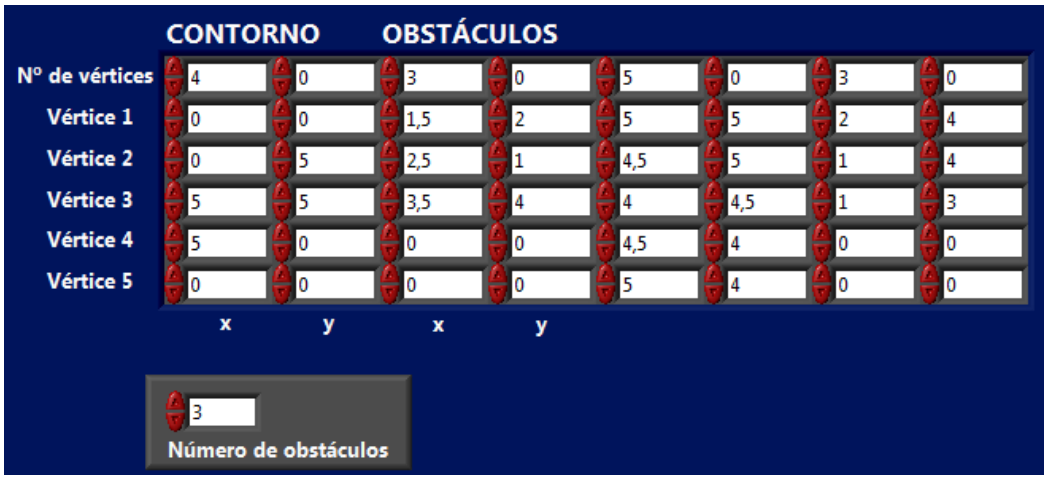


Figura 3.7 Ejemplo de ingreso de obstáculos en el HMI

A partir de la tercera columna se ingresan los datos pertenecientes a los obstáculos, por ejemplo, la tercera y cuarta columna le pertenece al primer obstáculo y los datos se ingresan de manera similar que el contorno; como se muestra en la Figura 3.7, el primer obstáculo es un polígono triangular cuyas coordenadas de los vértices son (1,5; 2), (2,5; 1) y (3,5; 4).

De manera similar, se ingresan los obstáculos del mapa de entorno restantes hasta completar todas las coordenadas de los vértices. Otro aspecto a ser tomado en cuenta es que en la generación de la trayectoria a seguir, no se toma en cuenta el diámetro del robot para seguir trayectorias entre 2 obstáculos, por lo que la HMI tiene la restricción de ingreso de obstáculos siempre y cuando la distancia entre los mismos sea superior al diámetro del robot, es decir 36 cm.

3.2.1.2 Conexión con Robotino®

Después de ingresar los obstáculos se recomienda verificar los datos necesarios para la conexión con Robotino®.

Como se muestra en la Figura 3.8, los datos mostrados son los que ya están configurados por default, pero es necesario verificar los mismos para establecer una conexión exitosa con Robotino®. En la dirección IP del robot debe estar aquella dirección que proporciona el display del Robotino® una vez que ha sido encendido y se debe verificar que el *Com Id* tenga el valor de -1. Además se tiene el *ImagePort,* puerto que permite el envío y recepción de datos, éste puede a veces ocasionar problemas, así que se recomienda cambiar de número de puerto si se presentan problemas en la comunicación.

Figura 3.8 Configuración de datos para conexión con Robotino®

Una vez ingresadas todas las coordenadas de los obstáculos y verificado los datos de conexión del Robotino® se procede a dar clic en *Datos OK* para proceder al siguiente paso que es obtener el diagrama de Voronoi del mapa de entorno ingresado y la generación de la trayectoria.

3.2.2 GENERACIÓN DE LA TRAYECTORIA

Después de haber dado clic en *Datos OK*, el programa tarda pocos segundos para generar el diagrama de Voronoi en base al entorno ingresado, así como genera una trayectoria en base a puntos de partida y llegada aleatorios, como se muestra en la Figura 3.9.

Para cambiar los puntos de partida y llegada acorde a lo deseado, se desplaza a estos puntos con el cursor sobre cualquier punto del diagrama de Voronoi obtenido, tómese en cuenta que por la cantidad de datos ingresados, este desplazamiento de los puntos puede tardar un poco, así como la obtención de la trayectoria.

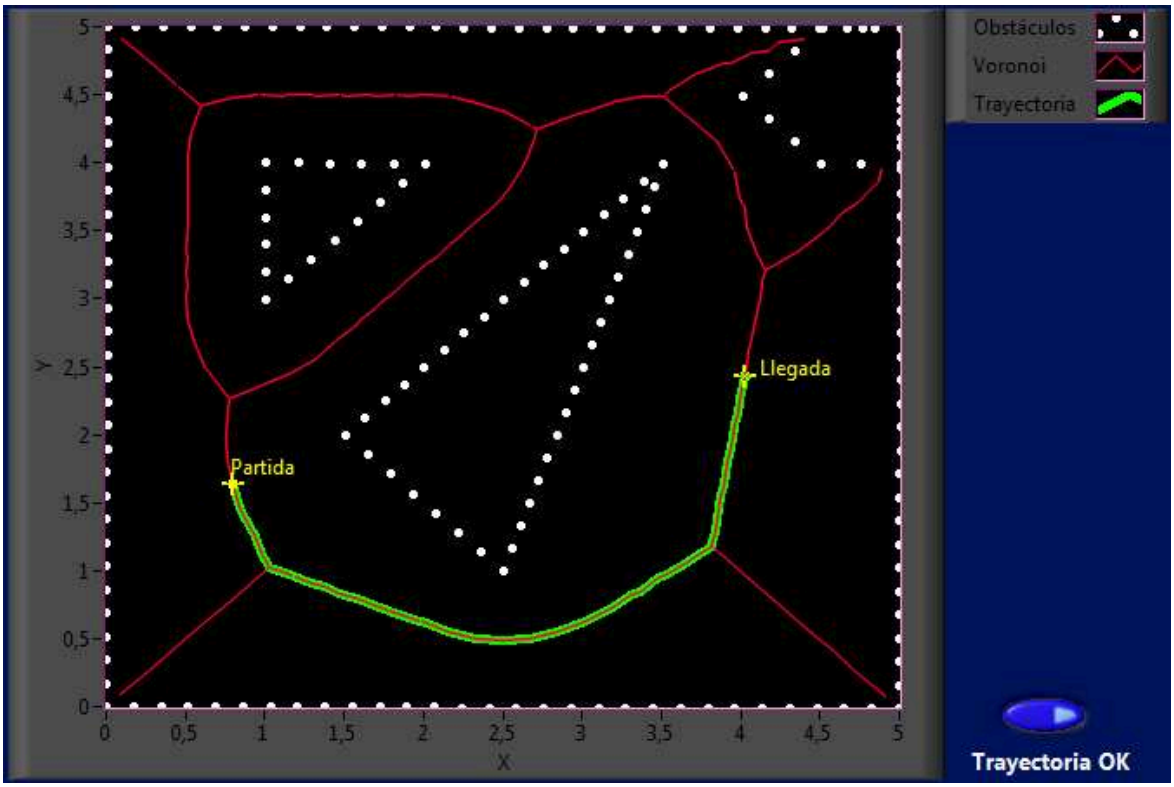


Figura 3.9 Diagrama de Voronoi y trayectoria generada por la HMI

Una vez seleccionados los puntos de partida y llegada deseados y se haya generado la trayectoria a seguir por el robot, se da clic en *Trayectoria OK*, para seguir con el siguiente paso, es decir el seguimiento de la trayectoria por el Robotino®.

3.2.3 SEGUIMIENTO DE LA TRAYECTORIA POR ROBOTINO®

Inmediatamente después de haber sido seleccionada una trayectoria a seguir, Robotino® seguirá la misma.

Debe considerarse el hecho de que la posición inicial o de partida del robot es conocida, así como la orientación del mismo.

Se considera al punto de partida como la referencia inicial respecto a los demás puntos, es decir será el punto $P(0,0)$ del sistema de coordenadas por lo que la posición del Robotino deberá ser: su eje sobre el punto de partida y con orientación como si fuese a desplazarse en línea recta sobre el eje x, tal como se muestra en la Figura 3.10.

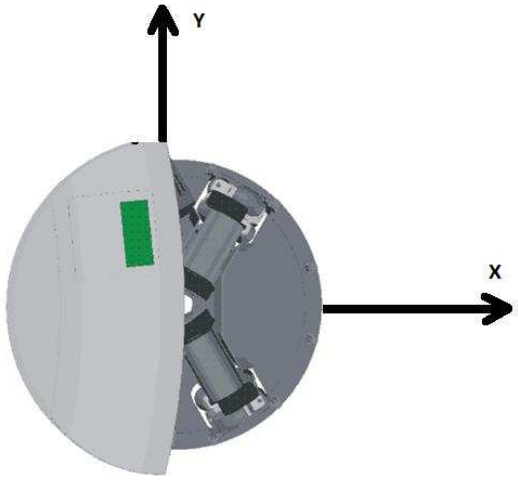


Figura 3.10 Orientación inicial de Robotino

Una vez que el Robotino® se desplaza la trayectoria indicada, se procede a la desconexión automática del mismo, con esto, se da por finalizado el movimiento. En el siguiente subcapítulo se detallan pruebas de varias trayectorias a seguir, así como los errores de posición cometidos y los resultados obtenidos.

3.3 PRUEBAS Y RESULTADOS

En el presente subcapítulo, se presentan tres pruebas con trayectorias y mapas de entorno diferentes, así como los resultados alcanzados.

3.3.1 PRUEBA 1

El mapa de entorno ingresado, contiene 5 obstáculos de varias formas y tamaños, el diagrama de Voronoi y la trayectoria obtenida a seguir por el Robotino® se muestra en la Figura 3.11.

Es importante mencionar, que el mapa de entorno ingresado no es real, la idea de esta prueba es comprobar que el robot se desplace no sólo en la trayectoria generada sino además, los mismos puntos y distancias correspondientes.

Se colocó un marcador en el eje del robot y como se observa en la Figura 3.12, el robot siguió al menos la forma correspondiente a la trayectoria requerida.

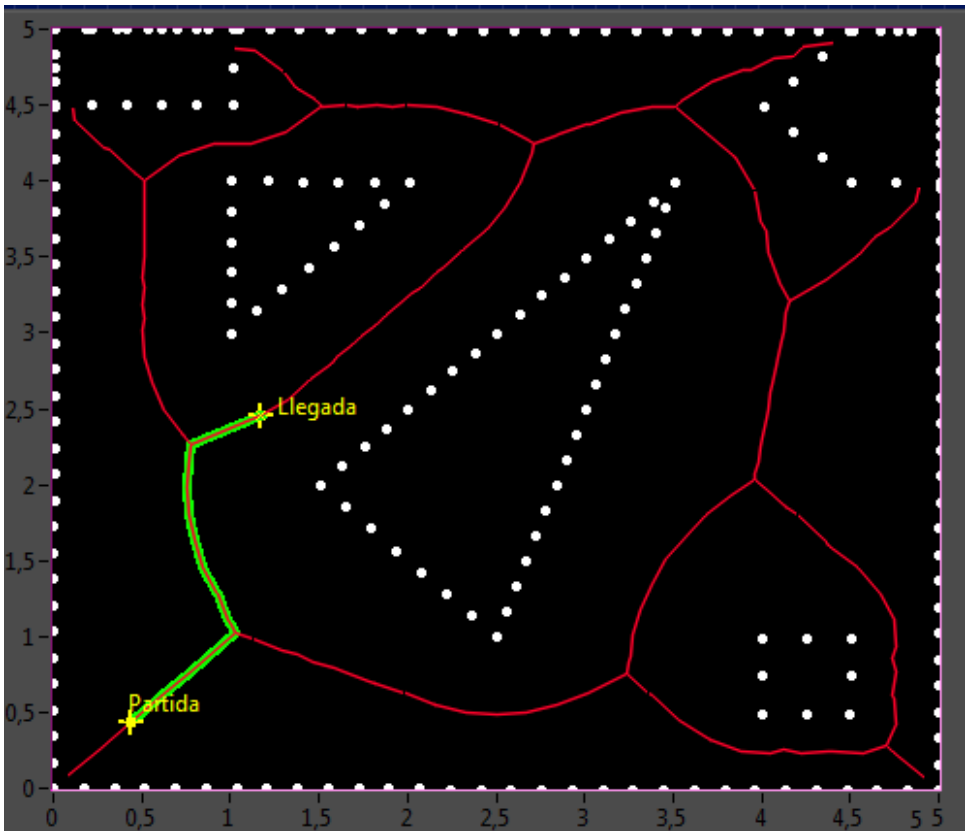


Figura 3.11 Trayectoria 1

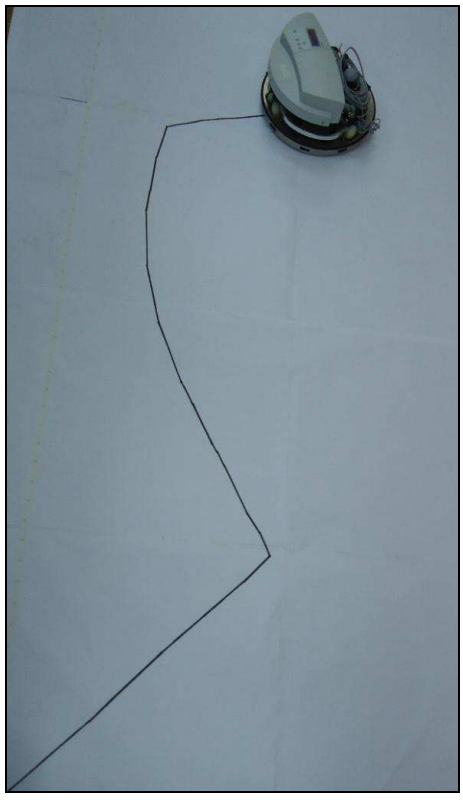


Figura 3.12 Trayectoria seguida por el robot en la primera prueba

Para análisis de errores, se tomaron 6 puntos a lo largo de la trayectoria, tanto teóricos como prácticos y se obtuvieron los resultados mostrados en la Tabla 3.1.

Téngase en cuenta que los valores de los puntos mostrados en la Tabla 3.1 y en las demás pruebas a realizarse fueron tomados considerando al punto de partida del Robotino® como referencia, en otras palabras, el punto de partida se lo considera como $P(0,0)$.

Tabla 3.1 Resultados prueba 1

	DATOS PRÁCTICOS		DATOS TEÓRICOS		ERRORES	
	X [m]	Y [m]	X [m]	Y [m]	X [%]	Y [%]
Punto 1	0,35	0,35	0,35	0,34	0	2,9
Punto 2	0,60	0,60	0,60	0,59	0	1,7
Punto 3	0,55	0,675	0,56	0,69	1,8	2,2
Punto 4	0,34	1,12	0,36	1,20	5,5	6,7
Punto 5	0,33	1,84	0,35	1,82	5,7	1,1
Punto 6	0,72	2,05	0,74	2,03	2,7	1

Adicionalmente se muestra en la Figura 3.13 la gráfica de la trayectoria seguida por el robot en base a los puntos obtenidos prácticamente.

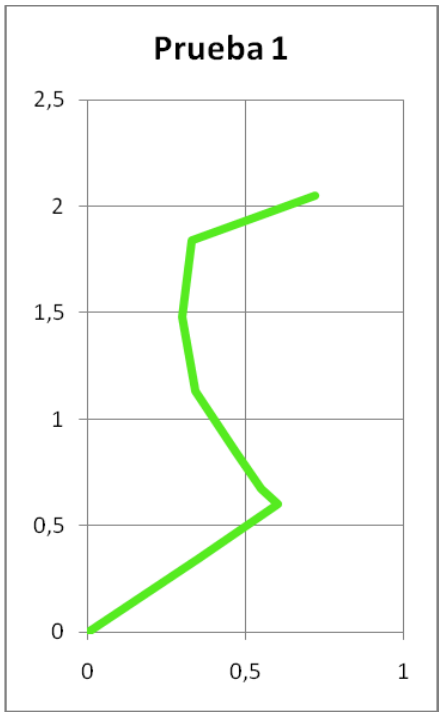


Figura 3.13 Trayectoria 1 en la práctica

Como se puede observar en la Tabla 3.1 los errores cometidos tanto en el eje x como en el eje y no sobrepasan el 7%, la mayoría de ellos no llegan ni al 5% de error por lo que se puede decir que el error de posición cometido en el seguimiento de esta trayectoria es relativamente pequeño y se puede considerar a los resultados obtenidos como satisfactorios.

Se puede justificar los errores cometidos no solo por posibles fallos en el programa o por la respuesta del hardware del robot a través de posibles problemas en la comunicación WLAN, sino también a errores producidos en las medidas dependiendo de la exactitud en que el robot fue colocado en su posición y orientación inicial, además se debe considerar la manera en que se tomaron las medidas respecto a un sistema de referencia.

3.3.2 PRUEBA 2

El mapa de entorno ingresado, es exactamente el mismo de la prueba anterior, con 5 obstáculos de varias formas y tamaños, por ende, el diagrama de Voronoi generado será el mismo.

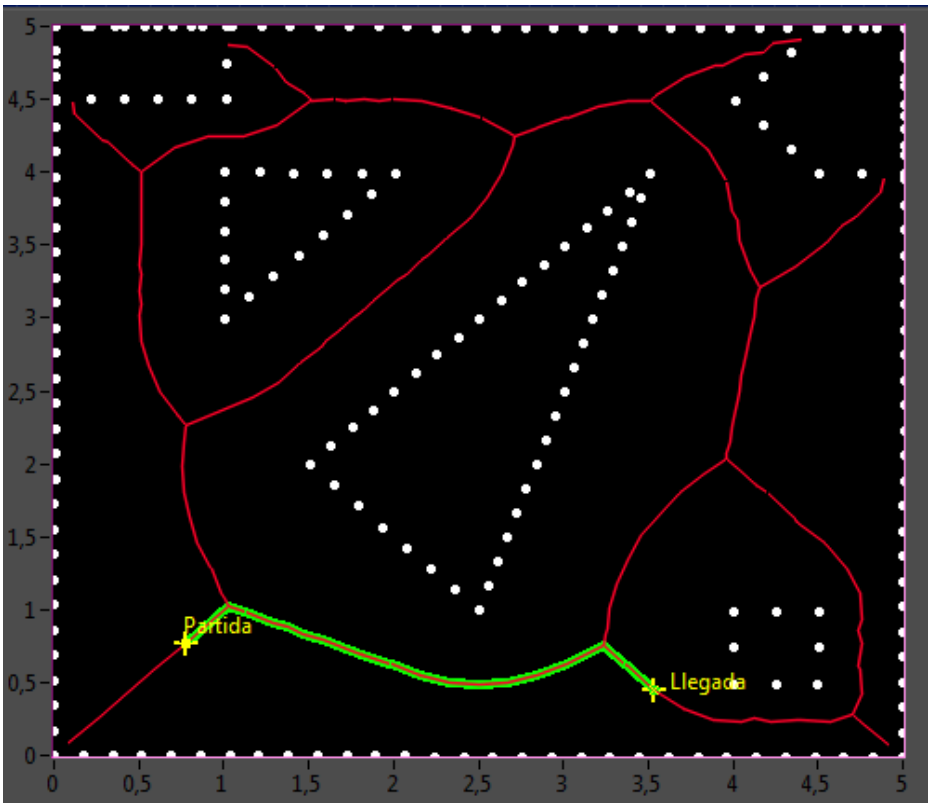


Figura 3.14 Trayectoria 2

A diferencia de la prueba anterior, en esta prueba se escogen distintos puntos de partida y llegada, por lo que la trayectoria a seguir por el Robotino® tiene una forma diferente a la presentada en la *prueba 1*. Se puede observar esta trayectoria en la Figura 3.14.

De igual manera, en la Figura 3.15 se puede observar la trayectoria seguida por el Robotino® y si se la compara a ésta con la generada en LabVIEW (Figura 3.14) se puede notar la similitud en cuanto a la forma de ambas trayectorias.

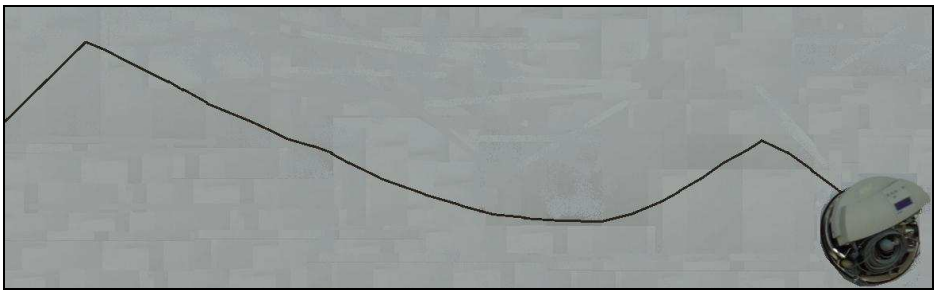


Figura 3.15 Trayectoria 2

Para análisis de errores en el seguimiento de la trayectoria por parte del Robotino®, se tomaron igualmente 6 puntos a lo largo de esta trayectoria, tanto teóricos como prácticos y se obtuvieron los resultados que se muestran en la Tabla 3.2.

Tabla 3.2 Resultados prueba 2

	DATOS PRÁCTICOS		DATOS TEÓRICOS		ERRORES	
	X [m]	Y [m]	X [m]	Y [m]	X [%]	Y [%]
Punto 1	0,25	0,25	0,25	0,25	0	0
Punto 2	0,615	0,10	0,61	0,10	0,82	0
Punto 3	1,20	-0,16	1,20	-0,15	0	6,7
Punto 4	1,90	-0,295	1,89	-0,27	5,3	9,3
Punto 5	2,38	-0,06	2,41	-0,05	1,2	20
Punto 6	2,75	-0,32	2,76	-0,33	0,36	3

Además se muestra en la Figura 3.16 la trayectoria seguida por el robot en base a los puntos obtenidos prácticamente.

Como se puede observar en la Figura 3.16, la trayectoria simulada en Excel es similar a la que teóricamente es generada en el LabVIEW, se debe tomar en

cuenta que la trayectoria de Excel no cuenta con todos los puntos que componen la trayectoria pero sí la mayoría, es por esta razón que la semejanza entre trayectorias no es precisa.

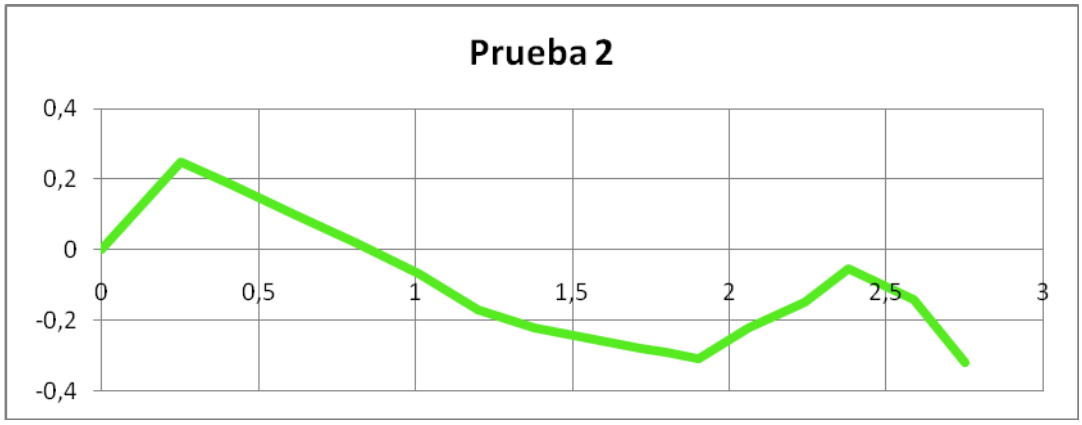


Figura 3.16 Trayectoria 2 en la práctica

En cuanto a los errores generados, se puede observar en la Tabla 3.2 que el error máximo cometido y de gran valor es del 20%, pero téngase en cuenta que es en el *punto 5* en el que la coordenada en x tiene un valor de 0,05 [m] teóricamente y en la práctica se obtuvo 0,06 [m], es decir 1 [cm] de error, lo cual se considera despreciable debido a que no significa que el robot se movió 6 [cm] en lugar de 5 [cm] sino que después de girar cierto ángulo y haber recorrido cierta distancia llego a esas coordenadas, como por ejemplo en este mismo punto en la coordenada en y se tienen 3 [cm] de diferencia entre valor práctico y teórico, pero las coordenadas al ser de alto valor generan un error del 1,2%. En cuanto a los demás errores obtenidos, se los considera bajos por lo que el resultado de la trayectoria seguida se considera satisfactorio.

Así mismo se justifica los errores cometidos debido a la inexactitud en la que fue colocado el robot en la posición de partida, ocasionando desde ya un error, así como la toma de datos respecto a un sistema de referencia elegido.

3.3.3 PRUEBA 3

El mapa de entorno ingresado para esta prueba es el del laboratorio de Microprocesadores, el cual comprende un contorno rectangular de 6,97x7,6 [m], posee 6 obstáculos de forma rectangular mismos que son de diferentes tamaños,

el diagrama de Voronoi y trayectoria generada por la HMI a seguir por el Robotino® en base a los puntos de partida y llegada deseados se muestran en la Figura 3.17.

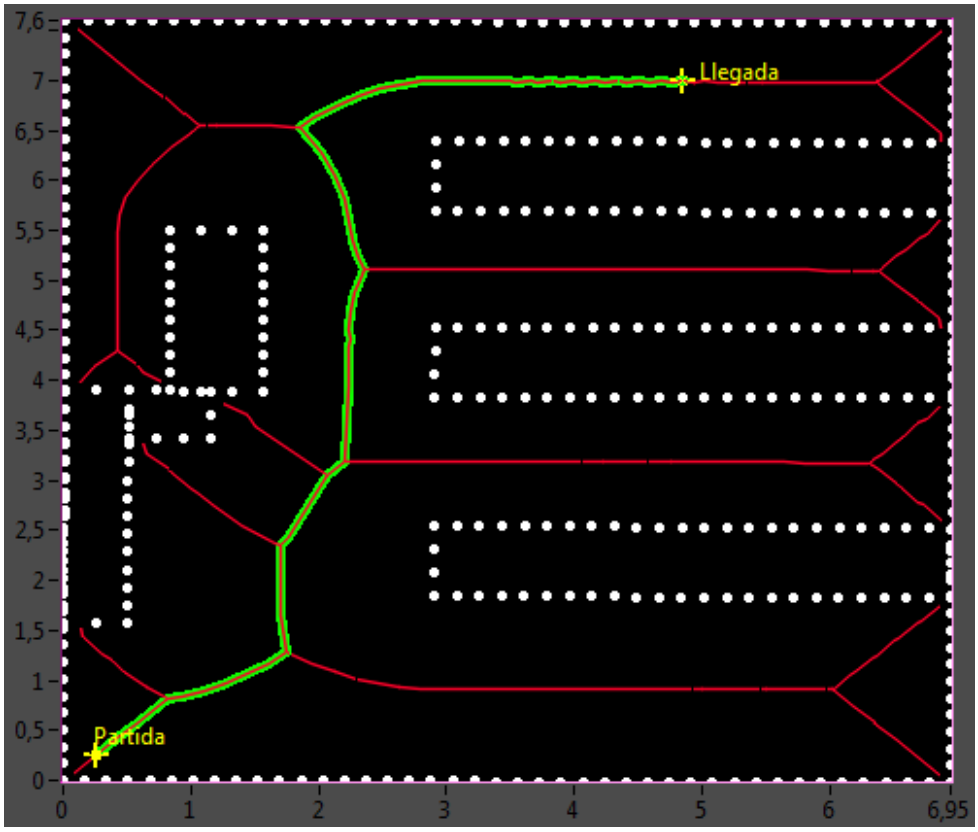


Figura 3.17 Trayectoria 3

Para análisis de errores, se tomaron 9 puntos a lo largo de la trayectoria, tanto teóricos como prácticos y se obtuvieron los resultados mostrados en la Tabla 3.3.

Tabla 3.3 Resultados prueba 3

	DATOS PRÁCTICOS		DATOS TEÓRICOS		ERRORES	
	X [m]	Y [m]	X [m]	Y [m]	X [%]	Y [%]
Punto 1	0,81	0,79	0,806	0,80	0,5	1,3
Punto 2	1,76	1,26	1,77	1,285	0,6	1,9
Punto 3	1,71	2,39	1,725	2,406	0,9	0,7
Punto 4	2,17	3,13	2,14	3,17	1,4	1,3
Punto 5	2,18	4,13	2,197	4,16	0,8	0,7
Punto 6	2,32	5,06	2,309	5,077	0,5	0,3
Punto 7	1,74	6,44	1,874	6,523	7,1	1,3
Punto 8	2,84	6,94	2,82	7,0	0,7	0,9
Punto 9	4,71	7,05	4,86	7,0	3,1	0,7

Se puede observar en la Figura 3.18 la trayectoria real seguida por el robot en base a los puntos obtenidos prácticamente.

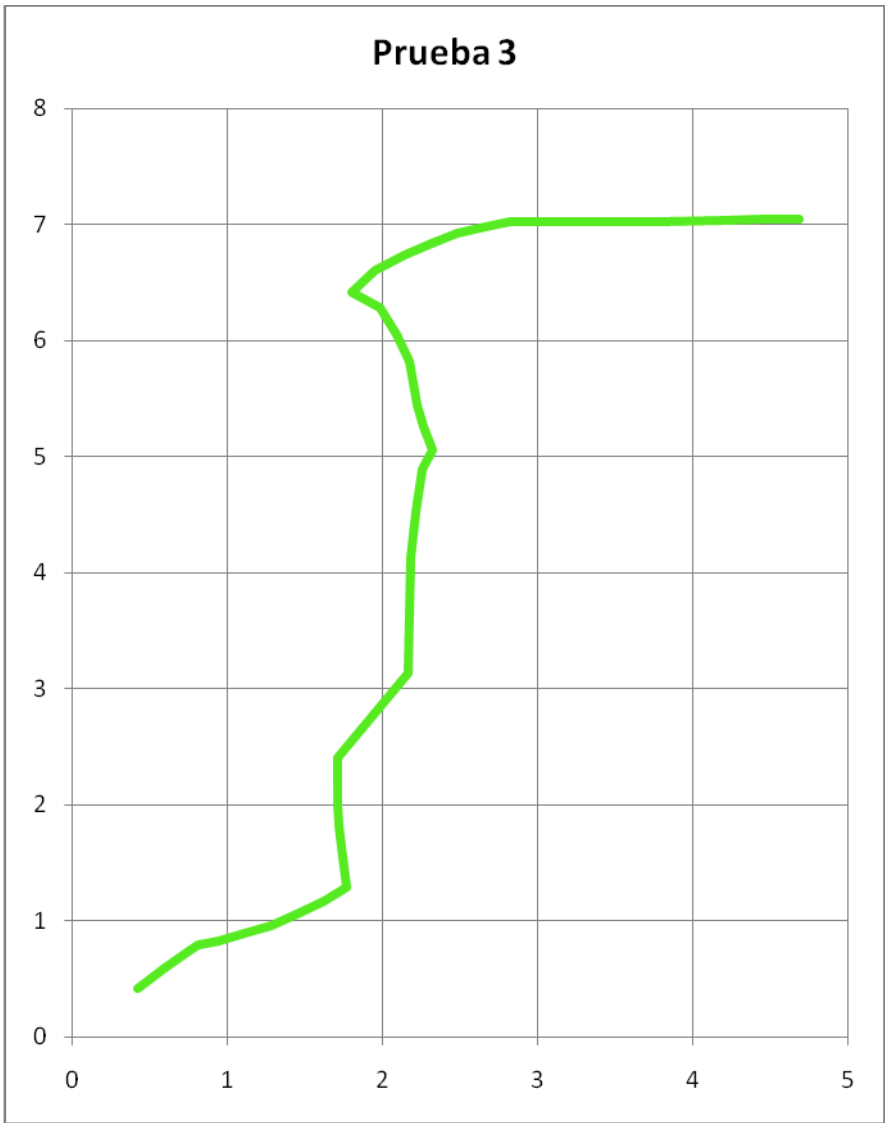


Figura 3.18 Trayectoria 3 en la práctica

Al igual que en las pruebas anteriores, el gráfico obtenido en Excel no cuenta con todos los puntos de la trayectoria seguida por el robot, es por esto que visualmente no presenta una similitud exacta en cuanto a la forma de la trayectoria, pero si muy parecida.

En cuanto a los errores aparecidos se puede observar en la Tabla 3.3 que son bastante pequeños la mayoría de ellos alrededor del 1%, solo en un punto se tiene un error de 7% pero éste no produce un error en los puntos posteriores de la

trayectoria por lo que los resultados obtenidos en esta prueba y en las anteriores se los considera satisfactorios.

Así mismo, para la justificación de errores, se debe mencionar que el laboratorio en la realidad no es perfectamente rectangular, por lo que desde un inicio ya se produce cierto error en cuanto al sistema de referencia. Además de los errores producidos en cuanto a las limitaciones de hardware del robot así como la exactitud en la que fue colocado el robot en su posición inicial.

A partir de estas 3 pruebas realizadas se puede concluir exitosamente este capítulo, dando paso al último capítulo, en el cual se presentan las conclusiones y recomendaciones en el transcurso del desarrollo del presente proyecto.

CAPÍTULO 4

CONCLUSIONES Y RECOMENDACIONES

Después de estudiar los métodos de planeación de trayectorias, desarrollar uno de ellos, diagramas de Voronoi, e implementar la ruta óptima a seguir en el robot Robotino® de Festo, se pueden citar las siguientes conclusiones y recomendaciones.

4.1 CONCLUSIONES

- La planeación y generación de trayectorias es una herramienta clave para la navegación autónoma de robots móviles debido a la necesidad de poseer una trayectoria óptima libre de obstáculos.

- Existen varios métodos para la planeación y generación de trayectorias por lo que es importante saber elegir que método se va a utilizar dependiendo de las características de la aplicación a desarrollar.

- Diagramas de Voronoi llega a ser uno de los mejores métodos de planeación de trayectorias debido a la seguridad que presenta la trayectoria al considerarse a esta lo más alejada de los obstáculos, así como presentar menos desventajas en comparación a otros métodos existentes.

- LabVIEW es un software que permite desarrollar proyectos en una gran variedad de áreas a través de la utilización de toolkits adicionales. Durante el desarrollo de esta aplicación proporcionó todas las herramientas necesarias para la programación y funcionamiento de este proyecto.

- El toolkit LabVIEW Robotics Module proporcionó grandes facilidades en cuanto a la programación, puesto que posee una gran variedad de VI's que

permitieron la creación del mapa de entorno así como la generación de una trayectoria óptima.

- El algoritmo de búsqueda A* cumple una función importante dentro de la generación de la trayectoria debido a que éste es el encargado de encontrar el camino de menor coste entre los nodos de partida y llegada. Obteniendo de esta manera la trayectoria más óptima.

- Robotino® es una plataforma educativa de alta calidad y plenamente funcional con accionamiento omnidireccional. Dispone de varios elementos como sensores analógicos, sensores binarios, cámara, encoders, un controlador de altas prestaciones, entre otros, que en conjunto proporcionan al sistema la necesaria "inteligencia" para hacer de Robotino® un robot autónomo.

- Dependiendo de la aplicación, el robot móvil Robotino® permite la programación de su hardware acorde a las necesidades del proyecto que se esté desarrollando. Al poseer una gran variedad de sensores y otros elementos, se puede implementar un sin número de aplicaciones y de esta manera facilitar la formación de los estudiantes en robótica móvil.

- La comunicación vía wireless presenta grandes ventajas respecto a otros tipos de comunicación para las HMI ya que elimina el medio físico de comunicación (cable). No obstante presenta desventajas, una de ellas es la interferencia que puede presentarse al recibir y enviar datos.

4.2 RECOMENDACIONES

- Se recomienda leer el manual de usuario antes de empezar a hacer uso de la HMI para evitar posibles daños y de esta manera obtener un correcto funcionamiento del proyecto.

- Es importante ingresar con precisión las coordenadas de los obstáculos para de esta manera evitar desde ya errores iniciales, de igual manera, la

colocación del robot en el punto de partida así como la orientación del mismo debe ser lo más precisa posible, ya que de lo contrario, este error ocasiona desvíos de la trayectoria original.

- Se recomienda que las baterías del Robotino® se encuentren 100% cargadas debido a que, de no estarlo, se podrían tener problemas y fallos en el funcionamiento de los motores lo que ocasionaría desvíos en la trayectoria.

- De darse el caso de tener problemas con la comunicación, se recomienda cambiar el número de puerto del *ImagePort*. Debe tenerse en cuenta que la red del Robotino® es una red pública sin clave de acceso, por lo que cualquier persona podría conectarse a la misma y posiblemente causar problemas en el funcionamiento del robot con la HMI.

- El programa podría ser adaptado para descargase en un robot más simple con la finalidad de no subutilizar las capacidades del Robotino®, debido a que lo único que se está controlando del robot móvil son los motores y los encoders, así como el sistema de comunicación inalámbrica.

- A pesar de que diagramas de Voronoi es uno de los métodos más completos, presenta ciertas desventajas, por lo que se podría optimizar la planeación y generación de trayectorias combinando a éste método con propiedades de otro método.

- El movimiento del robot se lo realiza solo en un sentido, en x, al ser de accionamiento omnidireccional se podría hacer que el movimiento sea en x y en y, de esta manera se podría dar una continuidad más precisa en el seguimiento del camino, pero a la vez la lógica de programación y obtención de las velocidades acorde a la trayectoria se vuelve más compleja.

REFERENCIAS BIBLIOGRÁFICAS

[1] Barrientos A.; Peñin L.; y otros, “Fundamentos de Robótica”, Segunda Edición, Editorial McGra-Hill. España, 2007

[2] Torres F.; Pomares J.; Gil P.; Puerta S.; Aracil R., “Robots y Sistemas Sensoriales”, Editorial Pearson Educación SA, Madrid 2002

[3] Lara J., Loor O., “Control de una plataforma robótica bípeda”, EPN, Quito 2006

[4] Historia del arte de la Robótica, “Móviles o vehículos robot”, Abril 2009, http://robotik-jjlg.blogspot.com/2009/04/moviles-o-vehiculos-robot.html

[5] Sánchez A.; López C., “Diseño de un robot hexápodo, Hardware y Software de Control”, Escola Universitària Politécnica de Vilanova i la Geltrú, 2005

[6] González O., Bricogeek, “Hexapod Phoenix: El robot araña”, Marzo 2008, http://blog.bricogeek.com/noticias/diy/video-hexapod-phoenix-el-robot-arana/

[7] Roboserv, “Robucar”, 2010, http://www.roboserv.net/robucar

[8] CFIE de Valladolid, “Robots móviles: diseño”, Febrero 2002, http://cfievalladolid2.net/tecno/cyr_01/robotica/movil.htm

[9] Muñoz V.; Gil G.; García A., “Modelo Cinemático y Dinámico de un Robot Móvil Omnidireccional”, Universidad de Málaga, 2003

[10] Gracia L., “Modelado Cinemático y Control de Robots Móviles con ruedas”, Universidad Politécnica de Valencia, 2006

[11] Manual Robotino®, FESTO, 2007

[12] Fernández M.; Fernández D.; Valmaseda C., “Planificación de trayectorias para un Robot Móvil”, Universidad Complutense, Madrid 2009

[13] Ollero A., “Planificación de trayectorias para Robots Móviles”, Universidad de Málaga, 1995

[14] Latombe J., “Robot Motion Planning”, Kluwer Academic Pulishers, 1991

[15] Rodríguez E., “Diagramas de Voronoi”, Cinvestav-Tamaulipas, 2010

[16] Borja de los Santos J., “Planificación de Trayectorias – El algoritmo PRM”, España, 2007

[17] Ortega L., “El Diagrama de Voronoi”, Universidad de Jaen, España, 2010

[18] La Canción de Malapata, “El Diagrama de Voronoi”, Febrero 2010, http://lacanciondemalapata.blogspot.com/2010/02/el-diagrama-de-voronoi-i.html

[19] Wikipedia, “Algoritmo de búsqueda A*”, diciembre 2010, http://es.wikipedia.org/wiki/Algoritmo_de_b%C3%BAsqueda_A*

[20] De Verg M.; Cheong O.; Van Kreveld M.; Overmars M., “Computational Geometry: Algorithms and Applications”, Third Edition, Berlin, 2008

ANEXO A

MANUAL DE USUARIO

En el presente trabajo se desarrolla en LabVIEW uno de los métodos de planeación de trayectorias denominado diagramas de Voronoi para obtener la ruta más óptima acorde a un mapa de entorno, puntos de salida y llegada conocidos, descargando la misma en la plataforma educativa Robotino® de Festo vía wireless.

Todos los VI's requeridos para el funcionamiento del presente trabajo se encuentran dentro de un proyecto desarrollado en LabVIEW denominado *Planeación_Seguimiento_Trayectorias.lvproj*

A.1 REQUERIMIENTOS BÁSICOS

Para un correcto y adecuado funcionamiento del proyecto es necesario disponer de la versión LabVIEW 2010, así como tener instalados los toolkits:

- ✓ MathScript RT Module
- ✓ NI LabVIEW Robotics Module
- ✓ Robotino LabVIEW driver

Es importante que todos estos toolkits deban ser compatibles con la versión 2010 de LabVIEW.

A.2 ROBOTINO®

Robotino® es un sistema de robot móvil de alta calidad, plenamente funcional con accionamiento omnidireccional. Cuenta con 3 unidades de accionamiento que permiten movimientos en todas las direcciones. Además Robotino® se encuentra equipado con una webcam, una serie de sensores analógicos para mediciones de distancia y sensores digitales para detectar la velocidad real. Se puede conectar

actuadores y sensores adicionales en el Robotino® a través de una interfaz de entradas/salidas.

Figura A.1 Plataforma educativa Robotino® de Festo

Para programar el Robotino® en un PC se utiliza el software Robotino®View, este software es capaz de transmitir señales de manera inalámbrica al controlador del motor, así como visualizar, cambiar y evaluar valores de los sensores. La programación se la puede hacer incluso durante el funcionamiento real. Otras alternativas de programación del Robotino® son APIs Linux y C++. En el caso de este proyecto la programación se la hace a través del LabVIEW haciendo uso del Robotino LabVIEW driver.

A.2.1 NOTAS SOBRE LA SEGURIDAD Y CORRECTO FUNCIONAMIENTO

Para un uso y funcionamiento seguro del Robotino® se debe tener en cuenta las siguientes observaciones generales:

- Los alumnos sólo deben trabajar en el Robotino® bajo la supervisión de un instructor
- Deberán observarse los datos de las fichas técnicas de los componentes individuales, en particular las notas sobre seguridad.
- Las conexiones eléctricas deben establecerse o interrumpirse únicamente cuando la tensión de alimentación esté desconectada.
- Se debe utilizar bajas tensiones de hasta 24 V DC.
- Mueva siempre el Robotino® con mucho cuidado, esto es, sujetándolo por sus asas.

- No intervenga manualmente a no ser que el Robotino® esté en reposo.
- El suelo por el que debe desplazarse el Robotino® debe ser plano y bien nivelado, de esta manera, pueden ejecutarse correctamente los movimientos deseados.
- El suelo debe estar seco y limpio para evitar cualquier daño a los componentes mecánicos y electrónicos.
- Es importante asegurarse que las baterías se encuentren cargadas completamente para un correcto funcionamiento del robot.

A.2.2 CONEXIÓN CON ROBOTINO®

Robotino® dispone de un teclado de membrana y display como se muestra en la Figura A.2.

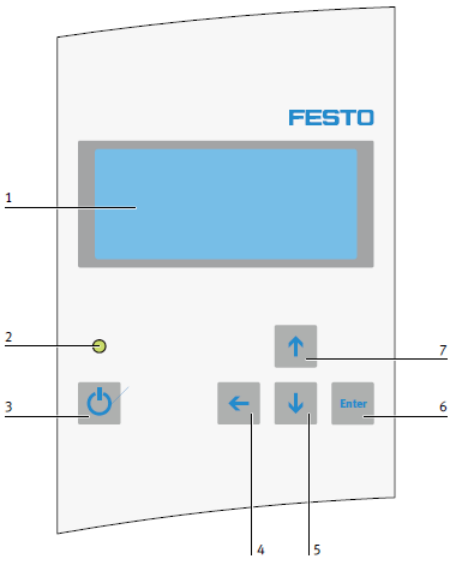


Figura A.2 Display (1), Led (2), On/Off (3), Subir un nivel el menú (4), Deslizar abajo una selección (5), Aceptar selección (6), Deslizar arriba una selección (7), tomado de [11]

Para encender al Robotino, se debe presionar el pulsador On/Off hasta que el LED se encienda. Al encenderse el display, aparecen dos barras que cruzan todo el ancho de la pantalla, esto es señal de que el PC del Robotino® está arrancando.

Después de unos 30 segundos, el display mostrará: el canal al que se encuentra conectado el Robotino®, su dirección IP: 172.26.201.2, el nombre de la red:

Robotino.005.106 y en la última línea muestra una barra indicando el estado de carga de las baterías así como la versión del robot: v2.0, como se puede observar en la Figura A.3. Esto es señal de que el Robotino® se halla preparado para funcionar.

Figura A.3 Display del Robotino®

Si no se acciona ninguna tecla durante 10 segundos, se apaga la iluminación del display para mantener el consumo de corriente lo más bajo posible durante el funcionamiento.

> **NOTA:** Para reactivar el display se pulsa una de las teclas de flecha. No se debe presionar la tecla de Enter para activar el display, con el fin de evitar un arranque no deseado de, por ejemplo, la ejecución de un programa de demostración.

A.2.2.1 Configuración de una conexión WLAN

Para configurar la conexión de Robotino a través de una WLAN, se siguen los siguientes pasos:

1. Activar la WLAN, la configuración de la WLAN debe permitir la asignación de una dirección IP para la WLAN. Solo entonces puede establecerse una conexión entre la red y el punto de acceso en el Robotino®.
2. Después de poner en marcha el Robotino® y esperar el proceso de arranque, se toma nota de la dirección IP que se muestra en el display, en este caso la dirección es: 172.26.201.2.
3. Permitir que la WLAN busque dispositivos inalámbricos dentro del alcance, entonces aparecerá una red con el nombre Robotino.005.106 en la lista de redes disponibles, tal como se muestra en la Figura A.4.
4. Establecer la conexión con la red Robotino.005.106 si ello no ha sido realizado a través del software de red.

Figura A.4 Redes inalámbricas disponibles a través de la WLAN

Para una conexión correcta, es indispensable verificar que se tengan los siguientes ajustes en la red:

- Asignar automáticamente una clave de red (SSID).
- Obtener una dirección IP automáticamente.

Ambos ajustes deben estar activos para poder establecer una conexión con el Robotino®.

A.2.2.2 Verificación de la conexión WLAN

Si se desea verificar la conexión WLAN, se lo puede hacer a través del MS-DOS, siguiendo los pasos:

1. Abrir la ventana de comandos MS-DOS, la cual se encuentra en Inicio>Programas>Accesorios>Símbolo del sistema. O alternativamente se puede escribir la orden *cmd* en Inicio>Ejecutar.
2. Escribir en la ventana que se ha abierto, la orden: *ping 172.26.201.2*, seguida de la tecla *Enter.*
3. De haberse establecido una conexión WLAN, se recibirán los mensajes mostrados en la Figura A.5.

Figura A.5 Conexión establecida con Robotino

4. Si no existe conexión con el Robotino, aparecerán los mensajes mostrados en la Figura A.6.

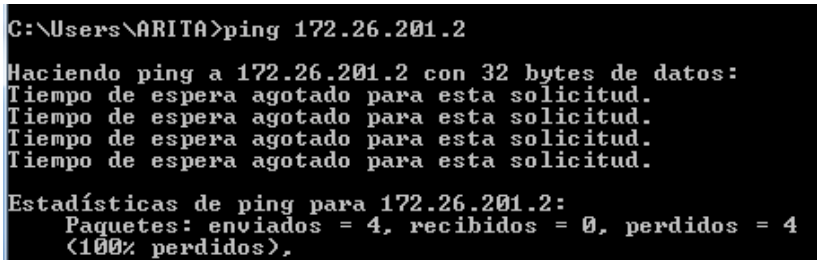


Figura A.6 Conexión no establecida con Robotino

A.2.2.3 Trabajo con varios Robotinos®

Robotino® presenta dos modalidades de funcionamiento en cuanto a conexión: *AP* (Access Point) y *Client*. A través del modo *AP* se pueden controlar de 1 hasta 4 Robotinos independientemente de manera segura con la misma dirección IP. El modo Client por su parte, permite trabajar con cualquier número de Robotinos en una red, asignando una dirección IP diferente para cada robot.

> **NOTA:** Debe verificarse que el modo a utilizarse se encuentre en *AP,* debido a que se va a trabajar con un solo Robotino®.

A.2.3 BATERÍAS

Junto con el Robotino® se suministran dos baterías adicionales. Las baterías que han sido utilizadas pueden reemplazarse y recargarse mientras se utiliza el segundo juego, esto permite que el Robotino® pueda funcionar ininterrumpidamente.

Se puede cargar las baterías directamente, es decir montadas las mismas en el Robotino® o hacerlo desmontadas del robot. Para la segunda opción, se dispone las baterías una junto a otra de forma que el terminal positivo de una batería (rojo) y el terminal negativo de la otra (negro) se hallen directamente uno junto al otro. Se conecta estos dos terminales con el cable azul suministrado. Luego se conectan las pinzas que hay en los extremos del cable de carga de la batería a los otros dos terminales de las baterías, la pinza roja en el terminal positivo libre (rojo) y la pinza negra en el terminal negativo libre (negro) como se muestra en la Figura A.7. Se inserta la clavija del cargador de baterías en el zócalo del cable. Asegurarse de que el retenedor encaje en su sitio.

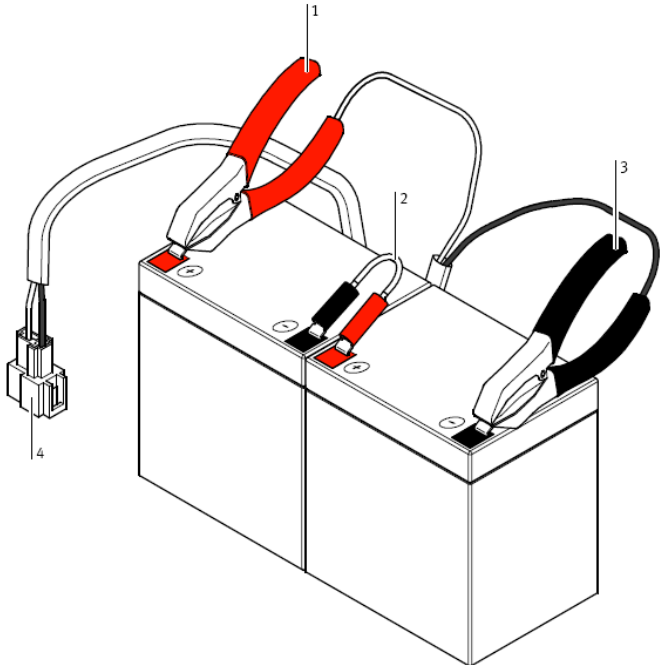


Figura A.7 Terminal positivo (rojo) (1), Cable azul (2), Terminal negativo (negro) (3), Retenedor (4)

El cargador de baterías dispone de un indicador LED que dependiendo de su color, indica el estado de carga de las baterías:

- Rojo : Tensión aplicada
- Amarillo: Cargando
- Verde: Carga finalizada

Si las baterías no están suficientemente cargadas, el funcionamiento de las unidades de accionamiento puede ser errático.

NOTA: Se debe cargar siempre ambas baterías al mismo tiempo. Si se carga una batería individualmente, quedará dañada.

A.3 DESCRIPCIÓN DE LOS ARCHIVOS DEL PROYECTO

Para poder hacer uso de los distintos VI's que componen el proyecto se debe ingresar al proyecto ***Planeación_Seguimiento_Trayectorias.lvproj,*** se cargará entonces una ventana como se muestra en la Figura A.8. En ***My Computer*** se disponen todos los VI's que componen el proyecto.

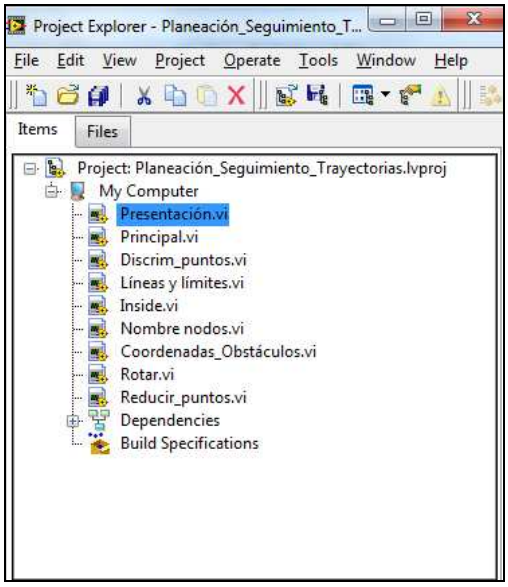


Figura A.8 Planeación_Seguimiento_Trayectorias.lvproj

A.3.1 Presentación.vi

Este VI, como su nombre lo dice, es la presentación del proyecto, su función es simplemente ser una introducción a la ejecución del proyecto. Es el primer VI a ser ejecutado.

A.3.2 Principal.vi

En este VI se dispone de toda la programación de la aplicación, se encuentra compuesto por todos los subVI's siguientes en la lista del proyecto. Aquí se adquieren los datos del entorno, se procesan estos datos a través de la obtención de más puntos del mapa de entorno, se obtiene diagrama de Voronoi en base a

puntos, se discriminan los puntos del diagrama que no pertenecen a lo requerido y en base al diagrama de Voronoi final se obtiene la trayectoria óptima la misma que es descargada en el Robotino®. Éste es el VI principal dentro del proyecto.

A.3.3 Rotar.vi

Este subVI rota los puntos de los obstáculos ingresados un ángulo despreciable, debido a que más adelante se generan errores en la obtención de la trayectoria a través del algoritmo A* cuando se dispone de cierta cantidad de puntos cuya componente en y es la misma en todos estos puntos. Al rotar, se elimina la restricción de que las componentes en y sean las mismas.

A.3.4 Coordenadas_Obstáculos.vi

Es preciso disponer de más puntos pertenecientes a los obstáculos, para lo cual se creó este subVI, mismo que en base a los vértices de los polígonos ingresados genera más puntos que pertenecen al contorno de los mismos.

A.3.5 Inside.vi

Debido a que el algoritmo ***voronoi*** que proporciona el LabVIEW trabaja con obstáculos ingresados como puntos, y en realidad lo que se tiene son obstáculos representados como polígonos es necesario realizar una primera discriminación de los puntos generados en el diagrama de Voronoi, es decir, descartar a aquellos que se encuentran fuera de los límites del mapa de entorno para lo cual se creó el subVI *Inside*.

A.3.6 Discrim_puntos.vi

Este subVI realiza una segunda discriminación de puntos del diagrama de Voronoi, aquí se eliminan los puntos que pertenecen al contorno y/o superficie de los obstáculos. Una vez realizadas estas discriminaciones, se obtiene el diagrama de Voronoi final del mapa de entorno ingresado.

A.3.7 Nombre nodos.vi

Al hacer uso del NI LabVIEW Robotics Module, se tiene la opción de crear un mapa del entorno del robot, para lo cual es necesario ingresar los puntos o nodos

del diagrama de Voronoi final. Para esto es necesario asignar un “nombre” a los puntos, es decir transformarlos a un formato string, debido a que éste es el formato con el cual trabaja el mapa creado. Esta transformación la hace el subVI *Nombre nodos*.

> **NOTA:** No se debe modificar ninguno de los VI’s del proyecto si no se tiene conocimiento de la programación y función de cada uno en el programa.

A.4 FUNCIONAMIENTO DE LA HMI

Para iniciar el funcionamiento de la HMI, se debe ingresar al archivo ejecutable *Planeación_Seguimiento_Trayectorias.exe*, se mostrará entonces, una ventana de presentación tal como se muestra en la Figura A.9.

Figura A.9 Ventana de presentación de la HMI

Se procede a dar clic en INICIAR y aparecerá una ventana como se muestra en la Figura A.10.

Para el manejo de la HMI, primero es necesario ingresar los obstáculos, seguido se seleccionan los puntos de partida y llegada del robot y se da la señal requerida por el robot para empezar el seguimiento de la trayectoria.

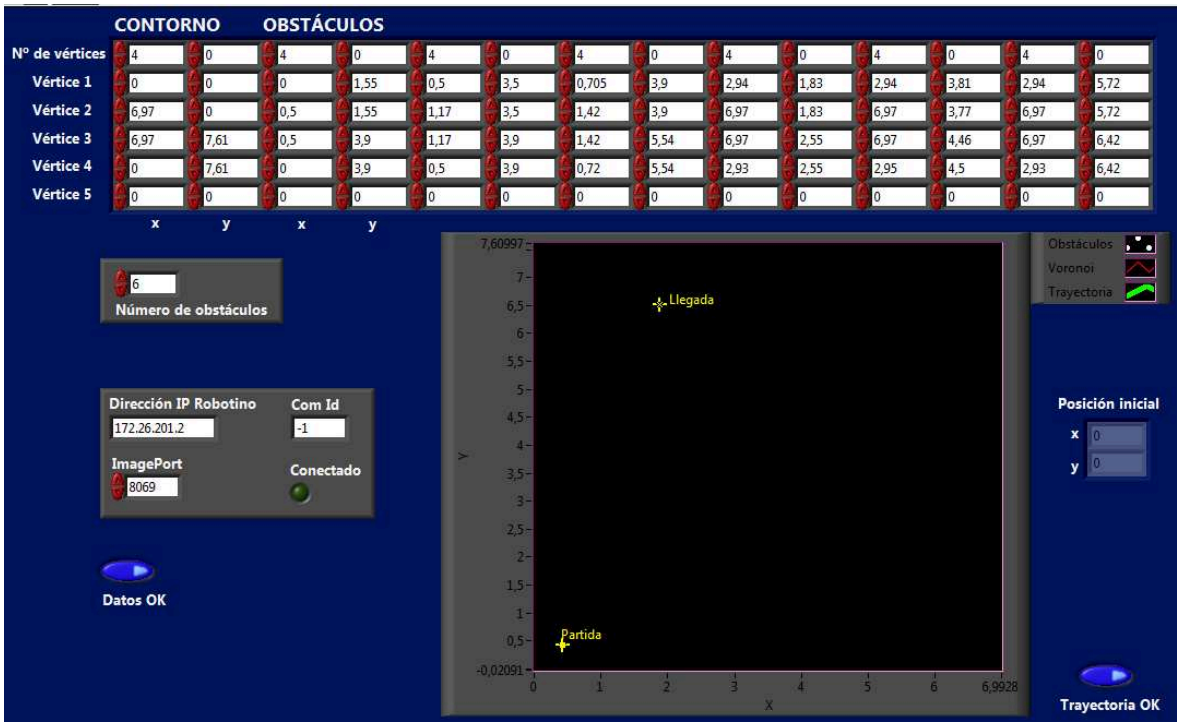


Figura A.10 Ventana de trabajo de la HMI

A.4.1 INGRESO DE DATOS

A.4.1.1 Ingreso de Obstáculos

Primero, se debe seleccionar el número de obstáculos que contiene el mapa de entorno, aparecerá un array con el número de filas y columnas acorde al número de obstáculos tal como se muestra en la Figura A.11.

> **NOTA:** Los datos a ser ingresados son las coordenadas de los vértices de los obstáculos representados por polígonos de máximo 5 vértices. Téngase en cuenta que las coordenadas deben ser ingresadas en metros.

A la primera y segunda columna le corresponden los datos necesarios para el ingreso del contorno, por ejemplo de darse un entorno cuadrangular de 5x5 [m], se tiene un polígono de 4 vértices, en la primera fila, primera columna se ingresa el número de vértices es decir 4, posteriormente a partir de la segunda fila se ingresa de vértice en vértice las coordenadas en x y en y en la primera y segunda

columna respectivamente. Al no poseer un quinto vértice, no se llena ninguna coordenada.

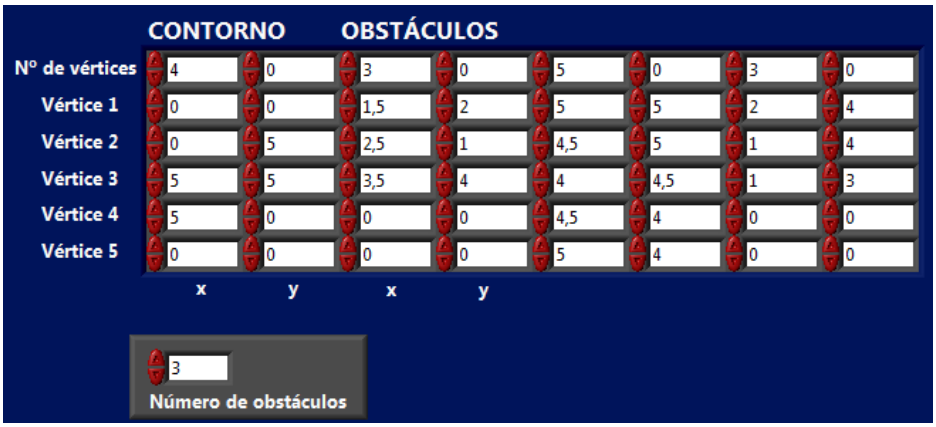


Figura A.11 Ejemplo de ingreso de obstáculos en el HMI

A partir de la tercera columna se ingresan los datos pertenecientes a los obstáculos, por ejemplo, la tercera y cuarta columna le pertenece al primer obstáculo y los datos se ingresan de manera similar que el contorno; como se muestra en la Figura A.11, el primer obstáculo es un polígono triangular cuyas coordenadas de los vértices son (1,5; 2), (2,5; 1) y (3,5; 4).

De manera similar, se ingresan los obstáculos del mapa de entorno restantes hasta completar todas las coordenadas de los vértices.

> **NOTA:** La HMI tiene la restricción de ingreso de obstáculos siempre y cuando la distancia entre los mismos sea superior al diámetro del robot, es decir 36 cm.

A.4.1.2 Conexión con Robotino®

Después de ingresar los obstáculos se recomienda verificar los datos necesarios para la conexión con Robotino® en la HMI.

> **NOTA:** Es necesario haber establecido previamente una conexión WLAN con el Robotino® como ya se explicó previamente.

Como se muestra en la Figura A.12, los datos mostrados son los que se encuentran configurados por default, pero es necesario verificar los mismos para establecer una conexión exitosa entre Robotino® y la HMI.

Se debe tener en cuenta los siguientes aspectos:

- ✓ En la dirección IP del Robotino® debe estar aquella dirección que proporciona el display del Robotino® una vez que ha sido encendido.
- ✓ Se debe verificar que el *Com Id* tenga el valor de -1, esto nos indica que no existe comunicación entre el robot y la HMI.
- ✓ El *ImagePort,* puerto que permite el envío y recepción de datos, puede a veces ocasionar problemas, así que se recomienda cambiar de número de puerto de presentarse problemas en la comunicación.

Figura A.12 Configuración de datos para conexión con Robotino®

Una vez ingresadas todas las coordenadas de los obstáculos y verificado los datos de conexión del Robotino® se procede a dar clic en *Datos OK*.

A.4.2 GENERACIÓN DE LA TRAYECTORIA

Después de dar clic en *Datos OK*, el programa tarda pocos segundos para generar el diagrama de Voronoi en base al entorno ingresado, así como genera una trayectoria en base a puntos de partida y llegada aleatorios, como se muestra en la Figura A.13. Para cambiar los puntos de partida y llegada acorde a lo deseado, se desplaza a estos puntos con el cursor sobre cualquier punto del diagrama de Voronoi obtenido.

> **NOTA:** Por la cantidad de datos ingresados, este desplazamiento de los puntos puede tardar un poco, así como la obtención de la trayectoria.

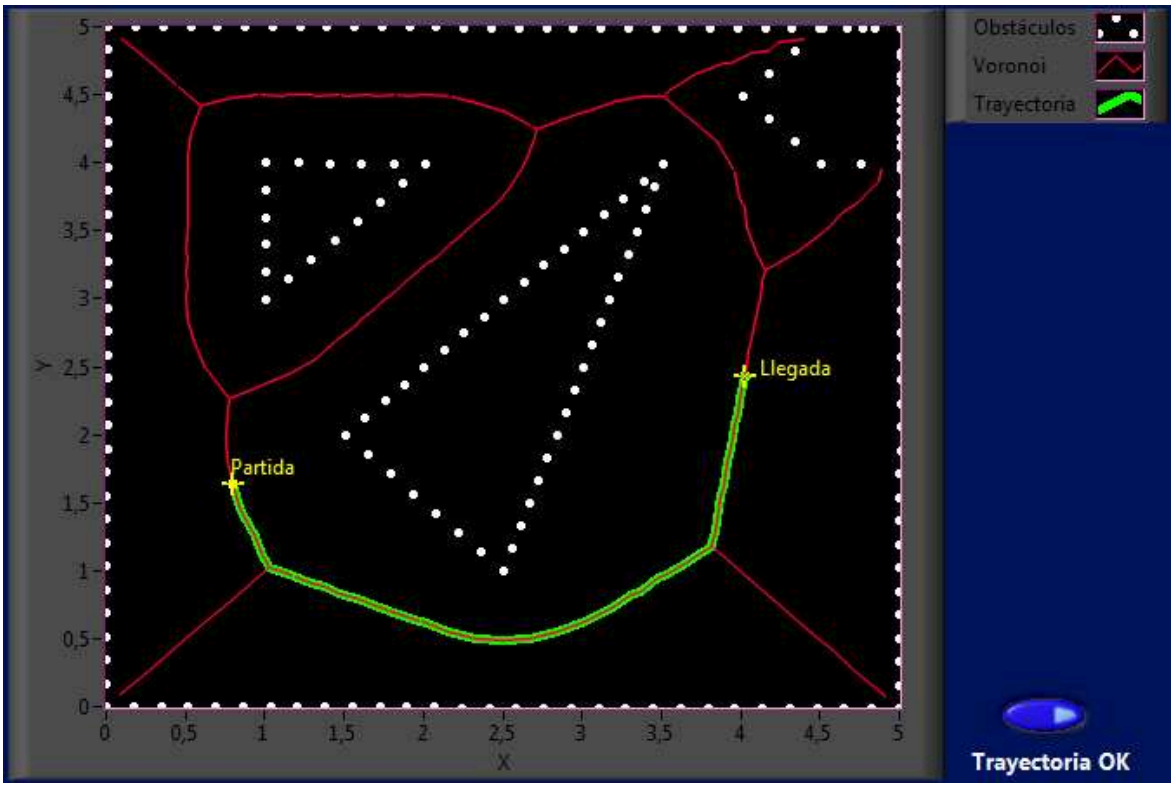


Figura A.13 Diagrama de Voronoi y trayectoria generada por la HMI

Una vez seleccionados los puntos de partida y llegada deseados y se haya generado la trayectoria a seguir por el robot, se da clic en *Trayectoria OK.*

A.4.3 SEGUIMIENTO DE LA TRAYECTORIA POR ROBOTINO®

Inmediatamente después de haber sido seleccionada una trayectoria a seguir, Robotino® seguirá la misma.

> **NOTA:** Debe considerarse el hecho de que la posición inicial o de partida del robot es conocida, así como la orientación del mismo.

Se considera al punto de partida como la referencia inicial respecto a los demás puntos, es decir será el punto $P(0,0)$ del sistema de coordenadas por lo que la posición del Robotino deberá ser: su eje sobre el punto de partida y con orientación como si fuese a desplazarse en línea recta sobre el eje x, tal como se muestra en la Figura A.14.

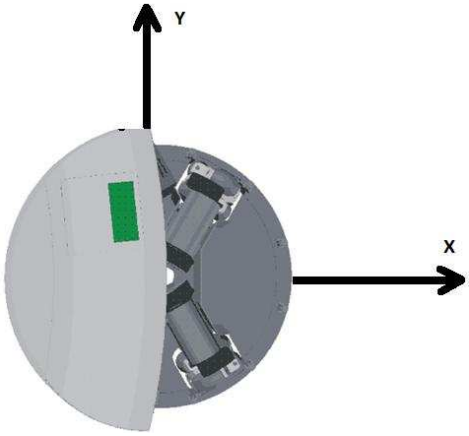


Figura A.14 Orientación inicial de Robotino

Una vez que el Robotino® se desplaza la trayectoria indicada, se procede a la desconexión automática del mismo, con esto, se da por finalizado el movimiento y la HMI volverá a la pantalla de presentación inicial.

> **IMPORTANTE:** se debe tomar en cuenta todas las precauciones y notas mencionadas en el manual de usuario para un correcto funcionamiento de la HMI y del Robotino®.

ANEXO B

ROBOTINO®

B.1 DATOS TÉCNICOS

Parámetro	Valor
Alimentación de tensión	24 V DC, 4.5 A
Entradas digitales	8
Salidas digitales	8
Entradas analógicas	8 (0 – 10 V)
Salidas por relé	2

B.1.1 DATOS DE RENDIMIENTO DEL MOTOR

Motor DC (GR 42x25)	Unidad de medida	
Tensión nominal	V DC	24
Velocidad nominal	RPM	3600
Par nominal	Ncm	3,8
Intensidad nominal	A	0,9
Par de arranque	Ncm	20
Intensidad de arranque	A	4
Velocidad sin carga	RPM	4200
Intensidad sin carga	A	0,17
Intensidad de desmagnetización	A	6,5
Momento de inercia de la masa	gcm^2	71
Peso del motor	gr.	390

B.1.2 RODILLO OMNIDIRECCIONAL

Rodillo omnidireccional, accionado (ARG 80)	
Diámetro Ø	80 mm
Máxima capacidad de carga	40 kg

B.1.3 REDUCTOR

Reductor planetario (PLG 42 S)	
De una sola etapa, Nm:	3,5
De una sola etapa, i:	4 :1 – 8 :1
2-etapas, Nm:	6
2-etapas, i:	16 :1 – 64 :1
3-etapas, Nm:	14
3-etapas, i:	100 :1 – 512 :1

B.1.4 MÓDULO DE CÁMARA

Especificaciones técnicas	
Sensor de imágenes	Color VGA CMOS
Profundidad de color	24 Bit Color verdadero
Conexión a PC	USB 1.1
Resoluciones de vídeo	160 x 120, 30 fps (SQCGA) 176 x 144, 30 fps (QCIF) 320 x 240, 30 fps (QVGA) 352 x 288, 30 fps (CIF) 640 x 480, 15 fps (VGA)
Resoluciones a imagen parada	160 x 120 (SQCGA) 176 x 144 (QCIF) 320 x 240 (QVGA) 352 x 288 (CIF) 640 x 480 (VGA) 1024 x 768 (SVGA)
Formato de captura a imagen parada	BMP, JPG

B.1.5 EJEMPLO DE UN DISPLAY

Texto del display	**Descripción**
Robotino®	
172.26.1.1	Dirección IP del Robotino®
V2.0	Versión del software

B.1.6 SENSOR DE PROXIMIDAD INDUCTIVO ANALÓGICO

Datos técnicos	
Tensión de funcionamiento	15 – 30 V DC
Tensión de salida	0 – 10 V
Tipo	SIEA-M12B-UI-S
Número de artículo	538292
Diámetro	M12
Margen de detección	0 a 6 mm
Montaje	Casi enrasable
Frecuencia de conmutación	1000 Hz
Temperatura ambiente	-25 a +70° C
Protección	IP 67
Material del cuerpo	Latón cromado
Par de apriete máximo	10 Nm
Repetibilidad	0,01 mm

B.2 INSTRUCCIONES DE FUNCIONAMIENTO Y FICHAS TÉCNICAS

Unidad	Documentos (en Inglés)
Motores	MotorGR2042(Diagrams).pdf MotorGR2042(TechData).pdf MotorGR2042(Description).pdf
Acumuladores	EN_Powerfit_S3124S(Datasheet).pdf
Sensores de distancia	EN_Distance_Sensor_gp2d120.pdf
Sensor inductivo analógico	EN / DE / ES / FR 678411_Sensor_induktiv_analog_M12.pdf
Encoder incremental	RE30(Data).pdf RE30(Description).pdf
Sensor de reflexión directa	369669_Fibre_optic_device.pdf 369682_Fibre_optic_cable_diffuse.pdf 165327_Fibre_Optic_Device_SOEG_L.pdf 165358_Fibre_Optic_Cable_Diffuse_SOEZ_RT.pdf
Sensor de colisiones	SafetyEdges.pdf

Punto de acceso WLAN	WAP-0004(Manual).pdf WAP-0004(DataSheet).pdf
Fusibles	Fuses_Robotino®_Datasheet.pdf
Webcam, Archivo de Ayuda de Windows	Webcam Live Users Guide English.chm
Unidad de control	Kontron_M_MOPSlcdSE_MOPSSE_PSTEM111.pdf

Todos los documentos mencionados se hallan en el directorio “\Doc\EN” del CD-ROM suministrado conjuntamente con Robotino®.

La información actualizada y las modificaciones hechas en la documentación técnica del Robotino® se encuentran disponibles en Internet, en la dirección:

- ✓ http://www.festo-didactic.com

ANEXO C

DIAGRAMAS DE VORONOI

7 Voronoi Diagrams

The Post Office Problem

Suppose you are on the advisory board for the planning of a supermarket chain, and there are plans to open a new branch at a certain location. To predict whether the new branch will be profitable, you must estimate the number of customers it will attract. For this you have to model the behavior of your potential customers: how do people decide where to do their shopping? A similar question arises in social geography, when studying the economic activities in a country: what is the trading area of certain cities? In a more abstract setting we have a set of

Figure 7.1
The trading areas of the capitals of the twelve provinces in the Netherlands, as predicted by the Voronoi assignment model

central places—called *sites*—that provide certain goods or services, and we want to know for each site where the people live who obtain their goods or services from that site. (In computational geometry the sites are traditionally viewed as post offices where customers want to post their letters—hence the subtitle of this chapter.) To study this question we make the following simplifying assumptions:

- the price of a particular good or service is the same at every site;
- the cost of acquiring the good or service is equal to the price plus the cost of transportation to the site;

- the cost of transportation to a site equals the Euclidean distance to the site times a fixed price per unit distance;
- consumers try to minimize the cost of acquiring the good or service.

Usually these assumptions are not completely satisfied: goods may be cheaper at some sites than at others, and the transportation cost between two points is probably not linear in the Euclidean distance between them. But the model above can give a rough approximation of the trading areas of the sites. Areas where the behavior of the people differs from that predicted by the model can be subjected to further research, to see what caused the different behavior.

Our interest lies in the geometric interpretation of the model above. The assumptions in the model induce a subdivision of the total area under consideration into regions—the trading areas of the sites—such that the people who live in the same region all go to the same site. Our assumptions imply that people simply get their goods at the nearest site—a fairly realistic situation. This means that the trading area for a given site consists of all those points for which that site is closer than any other site. Figure 7.1 gives an example. The sites in this figure are the capitals of the twelve provinces in the Netherlands.

The model where every point is assigned to the nearest site is called the *Voronoi assignment model*. The subdivision induced by this model is called the *Voronoi diagram* of the set of sites. From the Voronoi diagram we can derive all kinds of information about the trading areas of the sites and their relations. For example, if the regions of two sites have a common boundary then these two sites are likely to be in direct competition for customers that live in the boundary region.

The Voronoi diagram is a versatile geometric structure. We have described an application to social geography, but the Voronoi diagram has applications in physics, astronomy, robotics, and many more fields. It is also closely linked to another important geometric structure, the so-called Delaunay triangulation, which we shall encounter in Chapter 9. In the current chapter we shall confine ourselves to the basic properties and the construction of the Voronoi diagram of a set of point sites in the plane.

7.1 Definition and Basic Properties

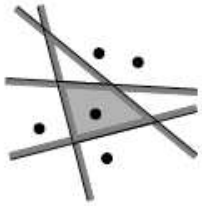
Denote the Euclidean distance between two points p and q by $\mathrm{dist}(p,q)$. In the plane we have

$$\mathrm{dist}(p,q) := \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}.$$

Let $P := \{p_1, p_2, \ldots, p_n\}$ be a set of n distinct points in the plane; these points are the sites. We define the Voronoi diagram of P as the subdivision of the plane into n cells, one for each site in P, with the property that a point q lies in the cell corresponding to a site p_i if and only if $\mathrm{dist}(q,p_i) < \mathrm{dist}(q,p_j)$ for each $p_j \in P$ with $j \neq i$. We denote the Voronoi diagram of P by $\mathrm{Vor}(P)$. Abusing the terminology slightly, we will sometimes use 'Vor(P)' or 'Voronoi diagram' to indicate only the edges and vertices of the subdivision. For example, when we

say that a Voronoi diagram is connected we mean that the union of its edges and vertices forms a connected set. The cell of $\mathrm{Vor}(P)$ that corresponds to a site p_i is denoted $\mathcal{V}(p_i)$; we call it the Voronoi cell of p_i. (In the terminology of the introduction to this chapter: $\mathcal{V}(p_i)$ is the trading area of site p_i.)

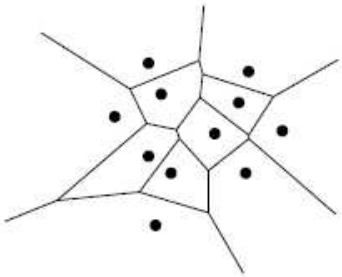
We now take a closer look at the Voronoi diagram. First we study the structure of a single Voronoi cell. For two points p and q in the plane we define the *bisector of p and q* as the perpendicular bisector of the line segment $\overline{pq}$. This bisector splits the plane into two half-planes. We denote the open half-plane that contains p by $h(p,q)$ and the open half-plane that contains q by $h(q,p)$. Notice that $r \in h(p,q)$ if and only if $\mathrm{dist}(r,p) < \mathrm{dist}(r,q)$. From this we obtain the following observation.



Observation 7.1 $\mathcal{V}(p_i) = \bigcap_{1 \leq j \leq n, j \neq i} h(p_i, p_j)$.

Thus $\mathcal{V}(p_i)$ is the intersection of $n-1$ half-planes and, hence, a (possibly unbounded) open convex polygonal region bounded by at most $n-1$ vertices and at most $n-1$ edges.

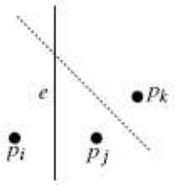
What does the complete Voronoi diagram look like? We just saw that each cell of the diagram is the intersection of a number of half-planes, so the Voronoi diagram is a planar subdivision whose edges are straight. Some edges are line segments and others are half-lines. Unless all sites are collinear there will be no edges that are full lines:



Theorem 7.2 *Let P be a set of n point sites in the plane. If all the sites are collinear then $\mathrm{Vor}(P)$ consists of $n-1$ parallel lines. Otherwise, $\mathrm{Vor}(P)$ is connected and its edges are either segments or half-lines.*

Proof. The first part of the theorem is easy to prove, so assume that not all sites in P are collinear.

We first show that the edges of $\mathrm{Vor}(P)$ are either segments or half-lines. We already know that the edges of $\mathrm{Vor}(P)$ are parts of straight lines, namely parts of the bisectors between pairs of sites. Now suppose for a contradiction that there is an edge e of $\mathrm{Vor}(P)$ that is a full line. Let e be on the boundary of the Voronoi cells $\mathcal{V}(p_i)$ and $\mathcal{V}(p_j)$. Let $p_k \in P$ be a point that is not collinear with p_i and p_j.

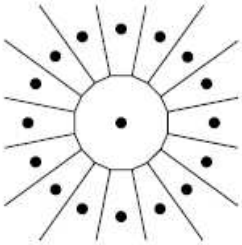


The bisector of p_j and p_k is not parallel to e and, hence, it intersects e. But then the part of e that lies in the interior of $h(p_k, p_j)$ cannot be on the boundary of $\mathcal{V}(p_j)$, because it is closer to p_k than to p_j, a contradiction.

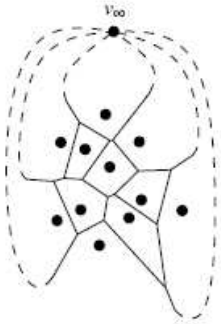
It remains to prove that $\mathrm{Vor}(P)$ is connected. If this were not the case then there would be a Voronoi cell $\mathcal{V}(p_i)$ splitting the plane into two. Because Voronoi cells are convex, $\mathcal{V}(p_i)$ would consist of a strip bounded by two parallel full lines. But we just proved that the edges of the Voronoi diagram cannot be full lines, a contradiction. □

Now that we understand the structure of the Voronoi diagram we investigate its complexity, that is, the total number of its vertices and edges. Since there are n sites and each Voronoi cell has at most $n-1$ vertices and edges, the complexity of $\mathrm{Vor}(P)$ is at most quadratic. It is not clear, however, whether $\mathrm{Vor}(P)$ can actually have quadratic complexity: it is easy to construct an example where a single Voronoi cell has linear complexity, but can it happen that many cells

have linear complexity? The following theorem shows that this is not the case and that the average number of vertices of the Voronoi cells is less than six.



Theorem 7.3 *For $n \geqslant 3$, the number of vertices in the Voronoi diagram of a set of n point sites in the plane is at most $2n-5$ and the number of edges is at most $3n-6$.*



Proof. If the sites are all collinear then the theorem immediately follows from Theorem 7.2, so assume this is not the case. We prove the theorem using *Euler's formula*, which states that for any connected planar embedded graph with m_v nodes, m_e arcs, and m_f faces the following relation holds:

$$m_v - m_e + m_f = 2.$$

We cannot apply Euler's formula directly to $\mathrm{Vor}(P)$, because $\mathrm{Vor}(P)$ has half-infinite edges and is therefore not a proper graph. To remedy the situation we add one extra vertex v_∞ "at infinity" to the set of vertices and we connect all half-infinite edges of $\mathrm{Vor}(P)$ to this vertex. We now have a connected planar graph to which we can apply Euler's formula. We obtain the following relation between n_v, the number of vertices of $\mathrm{Vor}(P)$, n_e, the number of edges of $\mathrm{Vor}(P)$, and n, the number of sites:

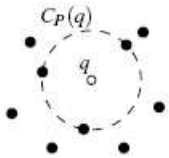
$$(n_v + 1) - n_e + n = 2. \tag{7.1}$$

Moreover, every edge in the augmented graph has exactly two vertices, so if we sum the degrees of all vertices we get twice the number of edges. Because every vertex, including v_∞, has degree at least three we get

$$2n_e \geqslant 3(n_v + 1). \tag{7.2}$$

Together with equation (7.1) this implies the theorem. □

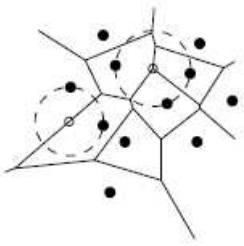
We close this section with a characterization of the edges and vertices of the Voronoi diagram. We know that the edges are parts of bisectors of pairs of sites and that the vertices are intersection points between these bisectors. There is a quadratic number of bisectors, whereas the complexity of the $\mathrm{Vor}(P)$ is only linear. Hence, not all bisectors define edges of $\mathrm{Vor}(P)$ and not all intersections are vertices of $\mathrm{Vor}(P)$. To characterize which bisectors and intersections define features of the Voronoi diagram we make the following definition. For a point q we define the *largest empty circle of q with respect to P*, denoted by $C_P(q)$, as the largest circle with q as its center that does not contain any site of P in its interior. The following theorem characterizes the vertices and edges of the Voronoi diagram.



Theorem 7.4 *For the Voronoi diagram $\mathrm{Vor}(P)$ of a set of points P the following holds:*

(i) A point q is a vertex of $\mathrm{Vor}(P)$ if and only if its largest empty circle $C_P(q)$ contains three or more sites on its boundary.

(ii) The bisector between sites p_i and p_j defines an edge of $\mathrm{Vor}(P)$ if and only if there is a point q on the bisector such that $C_P(q)$ contains both p_i and p_j on its boundary but no other site.



Proof. (i) Suppose there is a point q such that $C_P(q)$ contains three or more sites on its boundary. Let p_i, p_j, and p_k be three of those sites. Since the interior of $C_P(q)$ is empty q must be on the boundary of each of $\mathcal{V}(p_i)$, $\mathcal{V}(p_j)$, and $\mathcal{V}(p_k)$, and q must be a vertex of $\mathrm{Vor}(P)$.

On the other hand, every vertex q of $\mathrm{Vor}(P)$ is incident to at least three edges and, hence, to at least three Voronoi cells $\mathcal{V}(p_i)$, $\mathcal{V}(p_j)$, and $\mathcal{V}(p_k)$. Vertex q must be equidistant to p_i, p_j, and p_k and there cannot be another site closer to q, since otherwise $\mathcal{V}(p_i)$, $\mathcal{V}(p_j)$, and $\mathcal{V}(p_k)$ would not meet at q. Hence, the interior of the circle with p_i, p_j, and p_k on its boundary does not contain any site.

(ii) Suppose there is a point q with the property stated in the theorem. Since $C_P(q)$ does not contain any sites in its interior and p_i and p_j are on its boundary, we have $\mathrm{dist}(q,p_i) = \mathrm{dist}(q,p_j) \leqslant \mathrm{dist}(q,p_k)$ for all $1 \leqslant k \leqslant n$. It follows that q lies on an edge or vertex of $\mathrm{Vor}(P)$. The first part of the theorem implies that q cannot be a vertex of $\mathrm{Vor}(P)$. Hence, q lies on an edge of $\mathrm{Vor}(P)$, which is defined by the bisector of p_i and p_j.

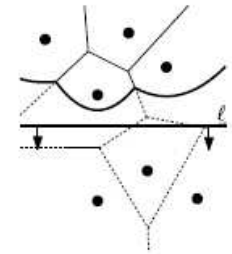
Conversely, let the bisector of p_i and p_j define a Voronoi edge. The largest empty circle of any point q in the interior of this edge must contain p_i and p_j on its boundary and no other sites. □

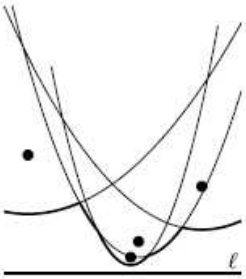
7.2 Computing the Voronoi Diagram

In the previous section we studied the structure of the Voronoi diagram. We now set out to compute it. Observation 7.1 suggests a simple way to do this: for each site p_i, compute the common intersection of the half-planes $h(p_i,p_j)$, with $j \neq i$, using the algorithm presented in Chapter 4. This way we spend $O(n\log n)$ time per Voronoi cell, leading to an $O(n^2\log n)$ algorithm to compute the whole Voronoi diagram. Can't we do better? After all, the total complexity of the Voronoi diagram is only linear. The answer is yes: the plane sweep algorithm described below—commonly known as *Fortune's algorithm* after its inventor—computes the Voronoi diagram in $O(n\log n)$ time. You may be tempted to look for an even faster algorithm, for example one that runs in linear time. This turns out to be too much to ask: the problem of sorting n real numbers is reducible to the problem of computing Voronoi diagrams, so any algorithm for computing Voronoi diagrams must take $\Omega(n\log n)$ time in the worst case. Hence, Fortune's algorithm is optimal.

The strategy in a plane sweep algorithm is to sweep a horizontal line—the *sweep line*—from top to bottom over the plane. While the sweep is performed information is maintained regarding the structure that one wants to compute. More precisely, information is maintained about the intersection of the structure with the sweep line. While the sweep line moves downwards the information does not change, except at certain special points—the *event points*.

Let's try to apply this general strategy to the computation of the Voronoi diagram of a set $P = \{p_1, p_2, \ldots, p_n\}$ of point sites in the plane. According to the plane





sweep paradigm we move a horizontal sweep line ℓ from top to bottom over the plane. The paradigm involves maintaining the intersection of the Voronoi diagram with the sweep line. Unfortunately this is not so easy, because the part of $\mathrm{Vor}(P)$ above ℓ depends not only on the sites that lie above ℓ but also on sites below ℓ. Stated differently, when the sweep line reaches the topmost vertex of the Voronoi cell $\mathcal{V}(p_i)$ it has not yet encountered the corresponding site p_i. Hence, we do not have all the information needed to compute the vertex. We are forced to apply the plane sweep paradigm in a slightly different fashion: instead of maintaining the intersection of the Voronoi diagram with the sweep line, we maintain information about the part of the Voronoi diagram of the sites above ℓ that cannot be changed by sites below ℓ.

Denote the closed half-plane above ℓ by ℓ^+. What is the part of the Voronoi diagram above ℓ that cannot be changed anymore? In other words, for which points $q \in \ell^+$ do we know for sure what their nearest site is? The distance of a point $q \in \ell^+$ to any site below ℓ is greater than the distance of q to ℓ itself. Hence, the nearest site of q cannot lie below ℓ if q is at least as near to some site $p_i \in \ell^+$ as q is to ℓ. The locus of points that are closer to some site $p_i \in \ell^+$ than to ℓ is bounded by a parabola. Hence, the locus of points that are closer to any site above ℓ than to ℓ itself is bounded by parabolic arcs. We call this sequence of parabolic arcs the *beach line*. Another way to visualize the beach line is the following. Every site p_i above the sweep line defines a complete parabola β_i. The beach line is the function that—for each x-coordinate—passes through the lowest point of all parabolas.

Observation 7.5 *The beach line is x-monotone, that is, every vertical line intersects it in exactly one point.*

It is easy to see that one parabola can contribute more than once to the beach line. We'll worry later about how many pieces there can be. Notice that the *breakpoints* between the different parabolic arcs forming the beach line lie on edges of the Voronoi diagram. This is not a coincidence: the breakpoints exactly trace out the Voronoi diagram while the sweep line moves from top to bottom. These properties of the beach line can be proved using elementary geometric arguments.

So, instead of maintaining the intersection of $\mathrm{Vor}(P)$ with ℓ we maintain the beach line as we move our sweep line ℓ. We do not maintain the beach line explicitly, since it changes continuously as ℓ moves. For the moment let's ignore the issue of how to represent the beach line until we understand where and how its combinatorial structure changes. This happens when a new parabolic arc appears on it, and when a parabolic arc shrinks to a point and disappears.

First we consider the events where a new arc appears on the beach line. One occasion where this happens is when the sweep line ℓ reaches a new site. The parabola defined by this site is at first a degenerate parabola with zero width: a vertical line segment connecting the new site to the beach line. As the sweep line continues to move downward the new parabola gets wider and wider. The part of the new parabola below the old beach line is now a part of the new beach

line. Figure 7.2 illustrates this process. We call the event where a new site is encountered a *site event*.

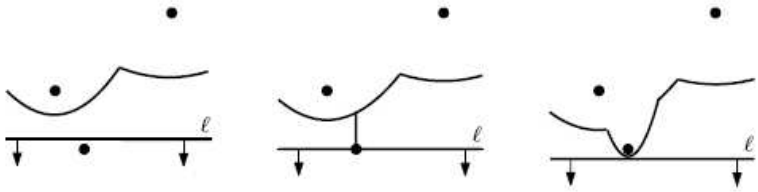
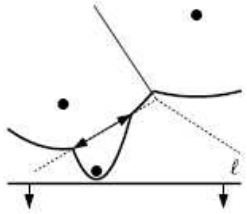


Figure 7.2
A new arc appears on the beach line because a site is encountered

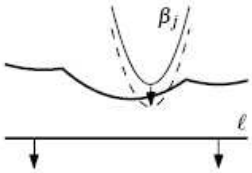
What happens to the Voronoi diagram at a site event? Recall that the breakpoints on the beach line trace out the edges of the Voronoi diagram. At a site event two new breakpoints appear, which start tracing out edges. In fact, the new breakpoints coincide at first, and then move in opposite directions to trace out the same edge. Initially, this edge is not connected to the rest of the Voronoi diagram above the sweep line. Later on—we will see shortly exactly when this will happen—the growing edge will run into another edge, and it becomes connected to the rest of the diagram.

So now we understand what happens at a site event: a new arc appears on the beach line, and a new edge of the Voronoi diagram starts to be traced out. Is it possible that a new arc appears on the beach line in any other way? The answer is no:



Lemma 7.6 *The only way in which a new arc can appear on the beach line is through a site event.*

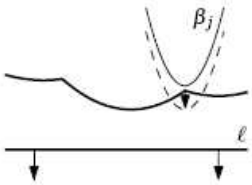
Proof. Suppose for a contradiction that an already existing parabola β_j defined by a site p_j breaks through the beach line. There are two ways in which this could happen.



The first possibility is that β_j breaks through in the middle of an arc of a parabola β_i. The moment this is about to happen, β_i and β_j are tangent, that is, they have exactly one point of intersection. Let ℓ_y denote the y-coordinate of the sweep line at the moment of tangency. If $p_j := (p_{j,x}, p_{j,y})$, then the parabola β_j is given by

$$\beta_j \;:=\; y = \frac{1}{2(p_{j,y} - \ell_y)}\left(x^2 - 2p_{j,x}x + p_{j,x}^2 + p_{j,y}^2 - \ell_y^2\right).$$

The formula for β_i is similar, of course. Using that both $p_{j,y}$ and $p_{i,y}$ are larger than ℓ_y, it is easy to show that it is impossible that β_i and β_j have only one point of intersection. Hence, a parabola β_j never breaks through in the middle of an arc of another parabola β_i.



The second possibility is that β_j appears in between two arcs. Let these arcs be part of parabolas β_i and β_k. Let q be the intersection point of β_i and β_k at which β_j is about to appear on the beach line, and assume that β_i is on the beach line left of q and β_k is on the beach line right of q, as in Figure 7.3. Then there is a circle C that passes through p_i, p_j, and p_k, the sites defining the parabolas. This circle is also tangent to the sweep line ℓ. The cyclic order

on C, starting at the point of tangency with ℓ and going clockwise, is p_i, p_j, p_k, because β_j is assumed to appear in between the arcs of β_i and β_k. Consider an infinitesimal motion of the sweep line downward while keeping the circle C tangent to ℓ; see Figure 7.3. Then C cannot have empty interior and still pass

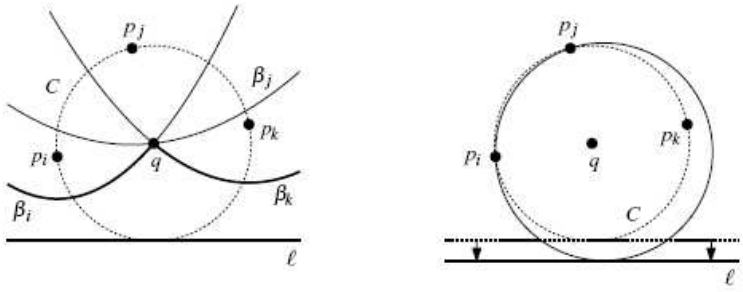


Figure 7.3
The situation when β_j would appear on the beach line, and the circle when the sweep line has proceeded

through p_j: either p_i or p_k will penetrate the interior. Therefore, in a sufficiently small neighborhood of q the parabola β_j cannot appear on the beach line when the sweep line moves downward, because either p_i or p_k will be closer to ℓ than p_j. □

An immediate consequence of the lemma is that the beach line consists of at most $2n - 1$ parabolic arcs: each site encountered gives rise to one new arc and the splitting of at most one existing arc into two, and there is no other way an arc can appear on the beach line.

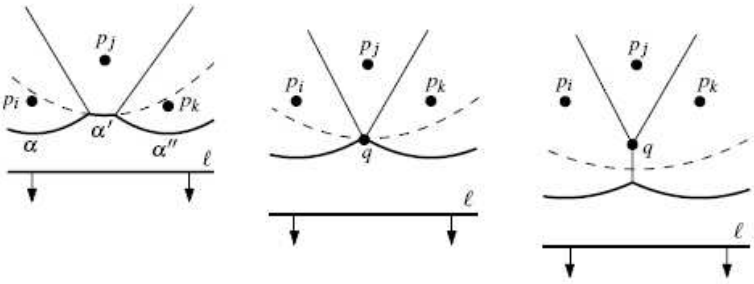


Figure 7.4
An arc disappears from the beach line

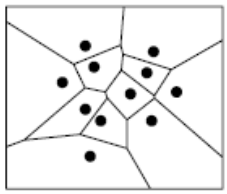
The second type of event in the plane sweep algorithm is where an existing arc of the beach line shrinks to a point and disappears, as in Figure 7.4. Let α' be the disappearing arc, and let α and α'' be the two neighboring arcs of α' before it disappears. The arcs α and α'' cannot be part of the same parabola; this possibility can be excluded in the same way as the first possibility in the proof of Lemma 7.6 was excluded. Hence, the three arcs α, α', and α'' are defined by three distinct sites p_i, p_j, and p_k. At the moment α' disappears, the parabolas defined by these three sites pass through a common point q. Point q is equidistant from ℓ and each of the three sites. Hence, there is a circle passing through p_i, p_j, and p_k with q as its center and whose lowest point lies

on ℓ. There cannot be a site in the interior of this circle: such a site would be closer to q than q is to ℓ, contradicting the fact that q is on the beach line. It follows that the point q is a vertex of the Voronoi diagram. This is not very surprising, since we observed earlier that the breakpoints on the beach line trace out the Voronoi diagram. So when an arc disappears from the beach line and two breakpoints meet, two edges of the Voronoi diagram meet as well. We call the event where the sweep line reaches the lowest point of a circle through three sites defining consecutive arcs on the beach line a *circle event*. From the above we can conclude the following lemma.

Lemma 7.7 *The only way in which an existing arc can disappear from the beach line is through a circle event.*

Now we know where and how the combinatorial structure of the beach line changes: at a site event a new arc appears, and at a circle event an existing arc drops out. We also know how this relates to the Voronoi diagram under construction: at a site event a new edge starts to grow, and at a circle event two growing edges meet to form a vertex. It remains to find the right data structures to maintain the necessary information during the sweep. Our goal is to compute the Voronoi diagram, so we need a data structure that stores the part of the Voronoi diagram computed thus far. We also need the two 'standard' data structures for any sweep line algorithm: an event queue and a structure that represents the status of the sweep line. Here the latter structure is a representation of the beach line. These data structures are implemented in the following way.

- We store the Voronoi diagram under construction in our usual data structure for subdivisions, the doubly-connected edge list. A Voronoi diagram, however, is not a true subdivision as defined in Chapter 2: it has edges that are half-lines or full lines, and these cannot be represented in a doubly-connected edge list. During the construction this is not a problem, because the representation of the beach line—described next—will make it possible to access the relevant parts of the doubly-connected edge list efficiently during its construction. But after the computation is finished we want to have a valid doubly-connected edge list. To this end we add a big bounding box to our scene, which is large enough so that it contains all vertices of the Voronoi diagram. The final subdivision we compute will then be the bounding box plus the part of the Voronoi diagram inside it.



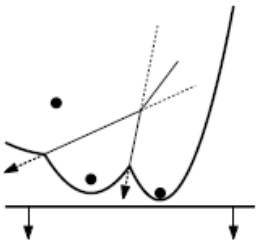
- The beach line is represented by a balanced binary search tree $\mathcal{T}$; it is the status structure. Its leaves correspond to the arcs of the beach line—which is x-monotone—in an ordered manner: the leftmost leaf represents the leftmost arc, the next leaf represents the second leftmost arc, and so on. Each leaf μ stores the site that defines the arc it represents. The internal nodes of $\mathcal{T}$ represent the breakpoints on the beach line. A breakpoint is stored at an internal node by an ordered tuple of sites $\langle p_i, p_j \rangle$, where p_i defines the parabola left of the breakpoint and p_j defines the parabola to the right. Using this representation of the beach line, we can find in $O(\log n)$

time the arc of the beach line lying above a new site. At an internal node, we simply compare the x-coordinate of the new site with the x-coordinate of the breakpoint, which can be computed from the tuple of sites and the position of the sweep line in constant time. Note that we do not explicitly store the parabolas.

In $\mathcal{T}$ we also store pointers to the other two data structures used during the sweep. Each leaf of $\mathcal{T}$, representing an arc α, stores one pointer to a node in the event queue, namely, the node that represents the circle event in which α will disappear. This pointer is **nil** if no circle event exists where α will disappear, or this circle event hasn't been detected yet. Finally, every internal node v has a pointer to a half-edge in the doubly-connected edge list of the Voronoi diagram. More precisely, v has a pointer to one of the half-edges of the edge being traced out by the breakpoint represented by v.

- The event queue Q is implemented as a priority queue, where the priority of an event is its y-coordinate. It stores the upcoming events that are already known. For a site event we simply store the site itself. For a circle event the event point that we store is the lowest point of the circle, with a pointer to the leaf in $\mathcal{T}$ that represents the arc that will disappear in the event.

All the site events are known in advance, but the circle events are not. This brings us to one final issue that we must discuss, namely the detection of circle events.



During the sweep the beach line changes its topological structure at every event. This may cause new triples of consecutive arcs to appear on the beach line and it may cause existing triples to disappear. Our algorithm will make sure that for every three consecutive arcs on the beach line that define a potential circle event, the potential event is stored in the event queue Q. There are two subtleties involved in this. First of all, there can be consecutive triples whose two breakpoints do not converge, that is, the directions in which they move are such that they will not meet in the future; this happens when the breakpoints move along two bisectors away from the intersection point. In this case the triple does not define a potential circle event. Secondly, even if a triple has converging breakpoints, the corresponding circle event need not take place: it can happen that the triple disappears (for instance due to the appearance of a new site on the beach line) before the event has taken place. In this case we call the event a *false alarm*.

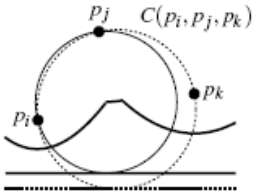
So what the algorithm does is this. At every event, it checks all the new triples of consecutive arcs that appear. For instance, at a site event we can get three new triples: one where the new arc is the left arc of the triple, one where it is the middle arc, and one where it is the right arc. When such a new triple has converging breakpoints, the event is inserted into the event queue Q. Observe that in the case of a site event, the triple with the new arc being the middle one can never cause a circle event, because the left and right arc of the triple come from the same parabola and therefore the breakpoints must diverge. Furthermore, for all disappearing triples it is checked whether they have a corresponding event in Q. If so, the event is apparently a false alarm, and

it is deleted from $\mathcal{Q}$. This can easily be done using the pointers we have from the leaves in $\mathcal{T}$ to the corresponding circle events in $\mathcal{Q}$.

Lemma 7.8 *Every Voronoi vertex is detected by means of a circle event.*

Proof. For a Voronoi vertex q, let p_i, p_j, and p_k be the three sites through which a circle $C(p_i, p_j, p_k)$ passes with no sites in the interior. By Theorem 7.4, such a circle and three sites indeed exist. For simplicity we only prove the case where no other sites lie on $C(p_i, p_j, p_k)$, and the lowest point of $C(p_i, p_j, p_k)$ is not one of the defining sites. Assume without loss of generality that from the lowest point of $C(p_i, p_j, p_k)$, the clockwise traversal of $C(p_i, p_j, p_k)$ encounters the sites p_i, p_j, p_k in this order.

We must show that just before the sweep line reaches the lowest point of $C(p_i, p_j, p_k)$, there are three consecutive arcs α, α' and α'' on the beach line defined by the sites p_i, p_j, and p_k. Only then will the circle event take place. Consider the sweep line an infinitesimal amount before it reaches the lowest point of $C(p_i, p_j, p_k)$. Since $C(p_i, p_j, p_k)$ doesn't contain any other sites inside or on it, there exists a circle through p_i and p_j that is tangent to the sweep line, and doesn't contain sites in the interior. So there are adjacent arcs on the beach line defined by p_i and p_j. Similarly, there are adjacent arcs on the beach line defined by p_j and p_k. It is easy to see that the two arcs defined by p_j are actually the same arc, and it follows that there are three consecutive arcs on the beach line defined by p_i, p_j, and p_k. Therefore, the corresponding circle event is in $\mathcal{Q}$ just before the event takes place, and the Voronoi vertex is detected. □



We can now describe the plane sweep algorithm in detail. Notice that after all events have been handled and the event queue $\mathcal{Q}$ is empty, the beach line hasn't disappeared yet. The breakpoints that are still present correspond to the half-infinite edges of the Voronoi diagram. As stated earlier, a doubly-connected edge list cannot represent half-infinite edges, so we must add a bounding box to the scene to which these edges can be attached. The overall structure of the algorithm is as follows.

Algorithm VORONOIDIAGRAM(P)
Input. A set $P := \{p_1, \ldots, p_n\}$ of point sites in the plane.
Output. The Voronoi diagram Vor(P) given inside a bounding box in a doubly-connected edge list $\mathcal{D}$.

1. Initialize the event queue $\mathcal{Q}$ with all site events, initialize an empty status structure $\mathcal{T}$ and an empty doubly-connected edge list $\mathcal{D}$.
2. **while** $\mathcal{Q}$ is not empty
3. **do** Remove the event with largest y-coordinate from $\mathcal{Q}$.
4. **if** the event is a site event, occurring at site p_i
5. **then** HANDLESITEEVENT(p_i)
6. **else** HANDLECIRCLEEVENT(γ), where γ is the leaf of $\mathcal{T}$ representing the arc that will disappear
7. The internal nodes still present in $\mathcal{T}$ correspond to the half-infinite edges of the Voronoi diagram. Compute a bounding box that contains all vertices of the Voronoi diagram in its interior, and attach the half-infinite edges to the bounding box by updating the doubly-connected edge list appropriately.

8. Traverse the half-edges of the doubly-connected edge list to add the cell records and the pointers to and from them.

The procedures to handle the events are defined as follows.

HANDLESITEEVENT(p_i)

1. If $\mathcal{T}$ is empty, insert p_i into it (so that $\mathcal{T}$ consists of a single leaf storing p_i) and return. Otherwise, continue with steps 2– 5.
2. Search in $\mathcal{T}$ for the arc α vertically above p_i. If the leaf representing α has a pointer to a circle event in $\mathcal{Q}$, then this circle event is a false alarm and it must be deleted from $\mathcal{Q}$.
3. Replace the leaf of $\mathcal{T}$ that represents α with a subtree having three leaves. The middle leaf stores the new site p_i and the other two leaves store the site p_j that was originally stored with α. Store the tuples $\langle p_j, p_i \rangle$ and $\langle p_i, p_j \rangle$ representing the new breakpoints at the two new internal nodes. Perform rebalancing operations on $\mathcal{T}$ if necessary.
4. Create new half-edge records in the Voronoi diagram structure for the edge separating $\mathcal{V}(p_i)$ and $\mathcal{V}(p_j)$, which will be traced out by the two new breakpoints.
5. Check the triple of consecutive arcs where the new arc for p_i is the left arc to see if the breakpoints converge. If so, insert the circle event into $\mathcal{Q}$ and add pointers between the node in $\mathcal{T}$ and the node in $\mathcal{Q}$. Do the same for the triple where the new arc is the right arc.

HANDLECIRCLEEVENT(γ)

1. Delete the leaf γ that represents the disappearing arc α from $\mathcal{T}$. Update the tuples representing the breakpoints at the internal nodes. Perform rebalancing operations on $\mathcal{T}$ if necessary. Delete all circle events involving α from $\mathcal{Q}$; these can be found using the pointers from the predecessor and the successor of γ in $\mathcal{T}$. (The circle event where α is the middle arc is currently being handled, and has already been deleted from $\mathcal{Q}$.)
2. Add the center of the circle causing the event as a vertex record to the doubly-connected edge list $\mathcal{D}$ storing the Voronoi diagram under construction. Create two half-edge records corresponding to the new breakpoint of the beach line. Set the pointers between them appropriately. Attach the three new records to the half-edge records that end at the vertex.
3. Check the new triple of consecutive arcs that has the former left neighbor of α as its middle arc to see if the two breakpoints of the triple converge. If so, insert the corresponding circle event into $\mathcal{Q}$. and set pointers between the new circle event in $\mathcal{Q}$ and the corresponding leaf of $\mathcal{T}$. Do the same for the triple where the former right neighbor is the middle arc.

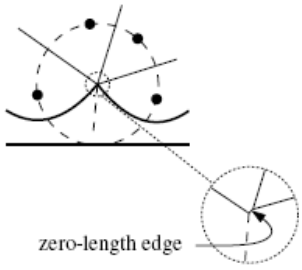
Lemma 7.9 *The algorithm runs in $O(n \log n)$ time and it uses $O(n)$ storage.*

Proof. The primitive operations on the tree $\mathcal{T}$ and the event queue $\mathcal{Q}$, such as inserting or deleting an element, take $O(\log n)$ time each. The primitive operations on the doubly-connected edge list take constant time. To handle an event we do a constant number of such primitive operations, so we spend

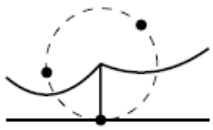
$O(\log n)$ time to process an event. Obviously, there are n site events. As for the number of circle events, we observe that every such event that is processed defines a vertex of $\mathrm{Vor}(P)$. Note that false alarms are deleted from $\mathcal{Q}$ before they are processed. They are created and deleted while processing another, real event, and the time we spend on them is subsumed under the time we spend to process this event. Hence, the number of circle events that we process is at most $2n - 5$. The time and storage bounds follow. □

Before we state the final result of this section we should say a few words about degenerate cases.

The algorithm handles the events from top to bottom, so there is a degeneracy when two or more events lie on a common horizontal line. This happens, for example, when there are two sites with the same y-coordinate. These events can be handled in any order when their x-coordinates are distinct, so we can break ties between events with the same y-coordinate but with different x-coordinates arbitrarily. However, if this happens right at the start of the algorithm, that is, if the second site event has the same y-coordinate as the first site event, then special code is needed because there is no arc above the second site yet. Now suppose there are event points that coincide. For instance, there will be several coincident circle events when there are four or more co-circular sites, such that the interior of the circle through them is empty. The center of this circle is a vertex of the Voronoi diagram. The degree of this vertex is at least four. We could write special code to handle such degenerate cases, but there is no need to do so. What will happen if we let the algorithm handle these events in arbitrary order? Instead of producing a vertex with degree four, it will just produce two vertices with degree three at the same location, with a zero length edge between them. These degenerate edges can be removed in a post-processing step, if required.

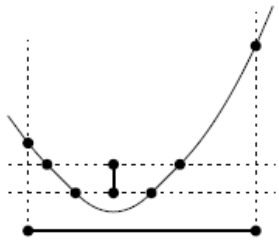


Besides these degeneracies in choosing the order of the events we may also encounter degeneracies while handling an event. This occurs when a site p_i that we process happens to be located exactly below the breakpoint between two arcs on the beach line. In this case the algorithm splits either of these two arcs and inserts the arc for p_i in between the two pieces, one of which has zero length. This piece of zero length now is the middle arc of a triple that defines a circle event. The lowest point of this circle coincides with p_i. The algorithm inserts this circle event into the event queue $\mathcal{Q}$, because there are three consecutive arcs on the beach line that define it. When this circle event is handled, a vertex of the Voronoi diagram is correctly created and the zero length arc can be deleted later. Another degeneracy occurs when three consecutive arcs on the beach line are defined by three collinear sites. Then these sites don't define a circle, nor a circle event.



We conclude that the above algorithm handles degenerate cases correctly.

Theorem 7.10 *The Voronoi diagram of a set of n point sites in the plane can be computed with a sweep line algorithm in $O(n \log n)$ time using $O(n)$ storage.*



7.3 Voronoi Diagrams of Line Segments

The Voronoi diagram can also be defined for objects other than points. The distance from a point in the plane to an object is then measured to the closest point on the object. Whereas the bisector of two points is simply a line, the bisector of two disjoint line segments has a more complex shape. It consists of up to seven parts, where each part is either a line segment or a parabolic arc. Parabolic arcs occur if the closest point of one line segment is an endpoint and the closest point of the other line segment is in its interior. In all other cases the bisector part is straight. Although bisectors and therefore the Voronoi diagram are somewhat more complex, the number of vertices, edges, and faces in the Voronoi diagram of n disjoint line segments is still only $O(n)$.

Assume for a moment that we allow the line segments to be non-crossing, that is, we allow them to share endpoints. Then a whole region of the plane can be equally close to two line segments via their common endpoint, and bisectors are not even curves anymore. To avoid the complications that arise in defining and computing Voronoi diagrams of line segments that share endpoints, we will simply assume here that all line segments are strictly disjoint. In many applications we can simply shorten the line segments very slightly to obtain disjoint line segments.

The sweep line algorithm for points can be adapted to the case of line segment sites. Let $S = \{s_1, \ldots, s_n\}$ be a set of n disjoint line segments. We call the segments of S *sites* as before, and use the terms *site endpoint* and *site interior* in the following description.

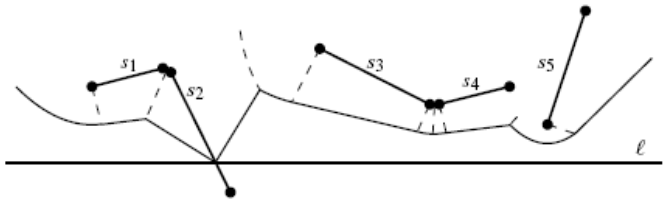


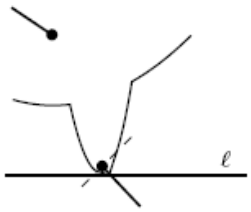
Figure 7.5
The beach line for a set of line segment sites. The breakpoints trace the dashed arcs, which include the Voronoi edges

Recall that our algorithm for point sites maintained a beach line: a piecewise parabolic x-monotone curve such that, for points on the curve, the distance to the closest site above the sweep line is equal to the distance to the sweep line. What does the beach line look like when the sites are segments? First we note that a line segment site may be partially above and partially below the sweep line. When defining the beach line, we consider only those parts of the sites that are above the sweep line. Hence, for a given position of the sweep line ℓ, the beach line consists of those points such that the distance to the closest portion of a site above ℓ is equal to the distance to ℓ. This means that the beach line now consists of parabolic arcs and straight line segments. A parabolic arc arises when that part of the beach line is closest to a site endpoint, and a straight line segment arises when that part of the beach line is closest to a site interior. Note that if a site interior intersects ℓ, then the beach line will have two straight line segments ending at the intersection—see site s_2 in Figure 7.5.

The breakpoints between parabolic arcs and straight segments on the beach line arise in several different ways. Figure 7.5 illustrates this. Assume any position ℓ of the sweep line during the downward sweep to analyze the types of breakpoint:

- If a point p is closest to two site endpoints while being equidistant from them and ℓ, then p is a breakpoint that traces a line segment (as in the point site case).
- If a point p is closest to two site interiors while being equidistant from them and ℓ, then p is a breakpoint that traces a line segment.
- If a point p is closest to a site endpoint and a site interior of different sites while being equidistant from them and ℓ, then p is a breakpoint that traces a parabolic arc.
- If a point p is closest to a site endpoint, the shortest distance is realized by a segment that is perpendicular to the line segment site, and p has the same distance from ℓ, then p is a breakpoint that traces a line segment.
- If a site interior intersects the sweep line, then the intersection is a breakpoint that traces a line segment (the site interior).

In the fourth and fifth cases, the breakpoint does not actually trace an arc of the Voronoi diagram, because only one site is involved. For the proper operation of the algorithm, dealing with such breakpoints and corresponding events is still necessary.



As in the sweep line algorithm for point sites, we again have site events and circle events. A site event occurs when the sweep line reaches a site endpoint. Obviously, site events at upper endpoints should be handled differently from site events at lower endpoints. At an upper endpoint, an arc of the beach line is split into two, and in between, four new arcs appear. The breakpoints between these four arcs are of the last two types. At a lower endpoint, the breakpoint that is the intersection of the site interior with the sweep line is replaced by two breakpoints of the fourth type, with a parabolic arc in between (for the newly discovered site endpoint).

Similarly, there are several types of circle event. They all correspond to the disappearance of an arc of the beach line, and they occur when the sweep line reaches the bottom of an empty circle that is defined by two or three sites above the sweep line. The centers of these empty circles are at locations where two consecutive breakpoints will meet. Depending on the types of the breakpoints that meet, several different cases can be distinguished and handled. If the two breakpoints are of any of the first three types, then three sites are involved. If one of the breakpoints is of the fourth type, then only two sites are involved. Breakpoints of the fifth type cannot be involved for disjoint line segments.

Notice that the Voronoi diagram that the algorithm computes is a subdivision with straight edges and parabolic arcs. Can we store this type of subdivision in a doubly-connected edge list? This is indeed possible, and the structure need not even be adapted. With each face, we store the corresponding site, so for any

half-edge $\vec{e}$ we can determine the two sites that have e on their bisector (using $IncidentFace(\vec{e})$ and $IncidentFace(Twin(\vec{e}))$). Since we can also easily find the two vertices between which the edge lies ($Origin(\vec{e})$ and $Origin(Twin(\vec{e}))$), we can determine the shape of any edge in constant time.

The whole sweep line algorithm is now just an extension of the one for point sites, with more cases to be distinguished and handled. However, the algorithm still has only $O(n)$ events, and each can be handled in $O(\log n)$ time.

Theorem 7.11 *The Voronoi diagram of a set of n disjoint line segment sites can be computed in $O(n \log n)$ time using $O(n)$ storage.*

One of the applications of the Voronoi diagram for line segments is in *motion planning* (covered more extensively in Chapter 13). Assume that a set of obstacles is given, consisting of n line segments in total, and that we have a robot $\mathcal{R}$. We assume that the robot can move freely in all directions, and is approximated well by an enclosing disc D. Suppose that we wish to find a collision-free motion from one location of the robot to another, or to decide that none exists.

One motion-planning technique is called *retraction*. The idea of retraction is that the arcs of the Voronoi diagram define the middle between the line segments, and therefore define a path with the most clearance. So a path over the arcs of the Voronoi diagram is the best option for a collision-free path. Figure 7.6 shows a set of line segments inside a rectangle, together with a Voronoi diagram of the line segments and the sides of the rectangle.

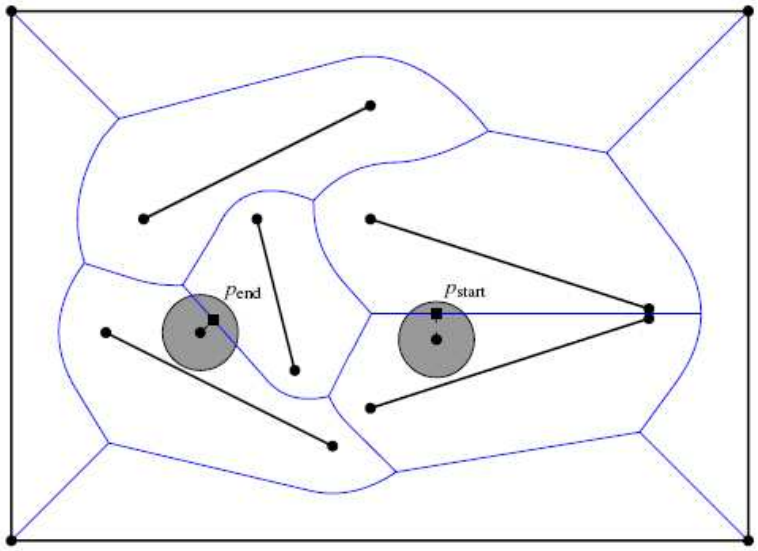


Figure 7.6
Voronoi diagram of line segments, and start and end positions of a disc

We can determine a collision-free path between two disc positions amidst a set of line segments with the following algorithm.

Algorithm RETRACTION($S, q_{\text{start}}, q_{\text{end}}, r$)
Input. A set $S := \{s_1, \ldots, s_n\}$ of disjoint line segments in the plane, and two discs D_{start} and D_{end} centered at q_{start} and q_{end} with radius r. The two disc positions do not intersect any line segment of S.
Output. A path that connects q_{start} to q_{end} such that no disc of radius r with its center on the path intersects any line segment of S. If no such path exists, this is reported.

1. Compute the Voronoi diagram $\text{Vor}(S)$ of S inside a sufficiently large bounding box.
2. Locate the cells of $\text{Vor}(P)$ that contain q_{start} and q_{end}.
3. Determine the point p_{start} on $\text{Vor}(S)$ by moving q_{start} away from the nearest line segment in S. Similarly, determine the point p_{end} on $\text{Vor}(S)$ by moving q_{end} away from the nearest line segment in S. Add p_{start} and p_{end} as vertices to $\text{Vor}(S)$, splitting the arcs on which they lie into two.
4. Let $\mathcal{G}$ be the graph corresponding to the vertices and edges of the Voronoi diagram. Remove all edges from $\mathcal{G}$ for which the smallest distance to the nearest sites is smaller than or equal to r.
5. Determine with depth-first search whether a path exists from p_{start} to p_{end} in $\mathcal{G}$. If so, report the line segment from q_{start} to p_{start}, the path in $\mathcal{G}$ from p_{start} to p_{end}, and the line segment from p_{end} to q_{end} as the path. Otherwise, report that no path exists.

The line segment connecting q_{start} to p_{start} cannot give a collision, because the disc only moves further away from the nearest obstacle. Similarly, the line segment between p_{end} and q_{end} is collision-free. For any two discs centered on the Voronoi diagram, a collision-free path between them exists on the Voronoi diagram if and only if such a path exists at all. Hence, for a disc-shaped robot, a path is found if one exists.

Theorem 7.12 *Given n disjoint line segment obstacles and a disc-shaped robot, the existence of a collision-free path between two positions of the robot can be determined in $O(n \log n)$ time using $O(n)$ storage.*

7.4 Farthest-Point Voronoi Diagrams

We now continue with a different application where Voronoi diagrams are needed. When objects are manufactured, slight deviations in the shapes of the objects will occur. When the objects need to be perfectly round, the manufactured objects are tested for their roundness. This is done by *coordinate measurement machines*, which sample points on the surface of the object. Assume that we have constructed a disc, and wish to determine its roundness. The machine gives us a set P of points in the plane that lie nearly on a circle. The *roundness* of a set of points is defined as the width of the smallest-width annulus that contains the points. An annulus is the region between two concentric circles, and its width is the difference between the radii of those circles.

The smallest-width annulus must of course have some points of the set P on its bounding circles. Let us call the outer circle C_{outer} and the inner circle C_{inner}.

Clearly, there must be at least one point on C_{outer}, otherwise we can reduce the size of C_{outer}, and at least one point on C_{inner}, otherwise we can increase the size of C_{inner}. But one point on each bounding circle cannot give us a smallest-width annulus yet. There appear to be three different cases, each with a total of four points on the two circles (Figure 7.7):

- C_{outer} contains at least three points of P, and C_{inner} contains at least one point of P.
- C_{outer} contains at least one point of P, and C_{inner} contains at least three points of P.
- C_{outer} and C_{inner} both contain two points of P.

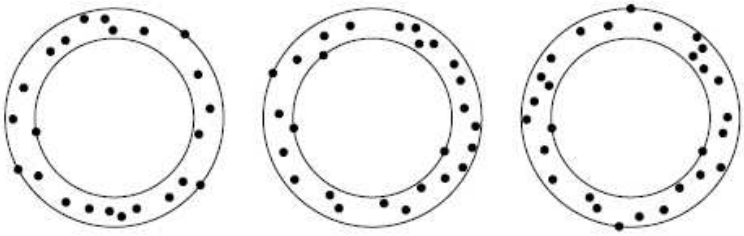
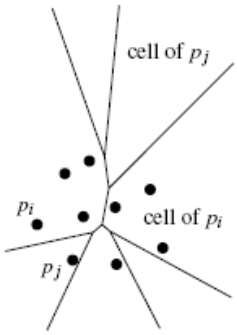


Figure 7.7
Three cases of the smallest-width annulus

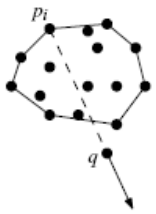
If C_{inner} or C_{outer} contains fewer points than listed in any of these cases, then we can always find an annulus with a smaller width. The problem of finding the smallest-width annulus enclosing a given point set looks similar to the problem of finding the smallest disc enclosing a point set, studied in Section 4.7. The technique we used for the smallest-disc problem, however, does not work for the smallest-width annulus: the property that an added point that does not lie in the optimal annulus so far must always lie on the boundary of the new optimal annulus does not hold.



Finding the smallest-width annulus is equivalent to finding its center point. Once the center point—let's call it q—is fixed, the annulus is determined by the points of P that are closest to and farthest from q. If we have the Voronoi diagram of P, then the closest point is the one in whose cell q lies. It turns out that a similar structure exists for the farthest point, namely the *farthest-point Voronoi diagram*. This divides the plane into cells in which the same point of P is the farthest point. The farthest-point Voronoi cell of a point p_i is the intersection of $n-1$ half-planes, just as for a standard Voronoi cell, but we take the "other sides" of the bisectors, the sides where p_i is farther away. Hence, all cells of the farthest-point Voronoi diagram are convex. Not every point of P has a cell in the farthest-point Voronoi diagram: the intersections of the half-planes can be empty. It is not hard to see that for any point in the plane, its farthest point in the set P must be a point that lies on the convex hull of P. Therefore, a point that lies inside the convex hull cannot have a cell in the farthest-point Voronoi diagram.

Observation 7.13 *Given a set P of points in the plane, a point of P has a cell in the farthest-point Voronoi diagram if and only if it is a vertex of the convex hull of P.*

We can prove more properties of the farthest-point Voronoi diagram. Suppose that a point $p_i \in P$ lies on the convex hull, and let q be some point in the plane for which p_i is the farthest point. Let $\ell(p_i, q)$ be the line through p_i and q. Then all points on the half-line starting at q, contained in $\ell(p_i, q)$, and not containing p_i, must also be in the farthest-point Voronoi cell of p_i. This implies that all cells are unbounded. The vertices and edges of the farthest-point Voronoi diagram form a tree-like structure (in the graph sense), because the diagram is connected and does not have cycles. A cycle would imply a bounded cell.



We can show that the farthest-point Voronoi diagram of n points has $O(n)$ vertices, edges, and cells (see also Exercise 7.14). There is another interesting property: the center of the smallest enclosing disc (see Section 4.7) is either a vertex of the farthest-point Voronoi diagram or the midpoint of two sites defining an edge of the farthest-point Voronoi diagram. In the former case, there are three equidistant farthest points, and in the latter case, two. Clearly, the center of the smallest enclosing disc cannot have just one point that is farthest from it.

Since the farthest-point Voronoi diagram has half-infinite edges, we cannot store it in a doubly-connected edge list, but we can adapt the structure slightly to deal with such subdivisions. We use a special vertex-like record as the origin of each half-edge that has no real vertex as its origin. These new records store the direction of the half-infinite edge instead of coordinates. Furthermore, half-edge records corresonding to half-infinite edges have either $Next(\vec{e})$ or $Prev(\vec{e})$ undefined. We shall still use the term "doubly-connected edge list" for this adapted version.

We now present an algorithm to compute the farthest-point Voronoi diagram of a set P of n points in the plane. First, we compute the convex hull of P, take its vertices, and put them in random order. Let this random order be $p_1, \ldots, p_h$. We remove the points $p_h, \ldots, p_4$ one by one from the cyclic order, and when removing p_i, store its clockwise neighbor $cw(p_i)$ and counterclockwise neighbor $ccw(p_i)$ at the time of removal. After a point has been removed, it cannot be the clockwise or counterclockwise neighbor anymore of points removed later.

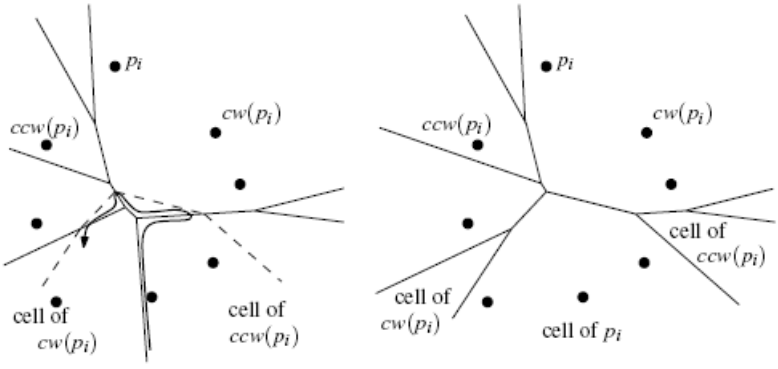
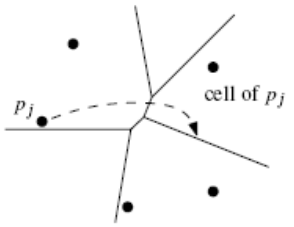


Figure 7.8
Addition of a point p_i to the farthest-point Voronoi diagram of $p_1, \ldots, p_{i-1}$

We compute the farthest-point Voronoi diagram of p_1, p_2, p_3 to initialize



the incremental construction. Then we insert the remaining points $p_4, \ldots, p_h$ while constructing the farthest-point Voronoi diagram. To be able to add the farthest-point Voronoi cell of p_i efficiently, given the farthest-point Voronoi diagram of $\{p_1, \ldots, p_{i-1}\}$, we maintain a pointer for each point p_j, $1 \leqslant j < i$, to the half-infinite half-edge of the doubly-connected edge list that is most counterclockwise in a traversal of the boundary of the farthest-point Voronoi cell of p_j.

We now look at the addition of the cell of p_i in more detail, see Figure 7.8. The cell will come "in between" the cells of $cw(p_i)$ and $ccw(p_i)$. Just before p_i is added, $cw(p_i)$ and $ccw(p_i)$ are each other's neighbors on the convex hull of $\{p_1, \ldots, p_{i-1}\}$, so their cells are separated by a half-infinite edge that is part of their bisector. The point $ccw(p_i)$ has a pointer to this edge. The bisector of p_i and $ccw(p_i)$ will give a new half-infinite edge that lies in the farthest-point Voronoi cell of $ccw(p_i)$, and is part of the boundary of the farthest-point Voronoi cell of p_i. We traverse the cell of $ccw(p_i)$ in the clockwise direction to see which edge the bisector intersects. On the other side of this edge is the farthest-point Voronoi cell of another point p_j from $\{p_1, \ldots, p_{i-1}\}$, and the bisector of p_j and p_i will also give an edge of the farthest-point Voronoi cell of p_i. We again traverse the cell of p_j in the clockwise direction to determine where the other insertion of the cell boundary and the bisector is located. By tracing cell boundaries in clockwise order, we trace the farthest-point Voronoi cell in counterclockwise order. The last bisector that we will find is with $cw(p_i)$, and it will give a new half-infinite edge in the farthest-point Voronoi diagram. All new edges found are added to the doubly-connected edge list representation, after which all edges that lie inside the farthest-point Voronoi cell of p_i are removed. They are no longer valid edges of the farthest-point Voronoi diagram of $\{p_1, \ldots, p_i\}$.

In short, the insertion of the next farthest-point Voronoi cell is done by tracing the new cell with the help of the existing diagram, adding the new edges, and removing the edges that have become obsolete.

Theorem 7.14 *Given a set of n points in the plane, its farthest-point Voronoi diagram can be computed in $O(n \log n)$ expected time using $O(n)$ storage.*

Proof. It takes $O(n \log n)$ time to compute the h points on the convex hull in counterclockwise order. The farthest-point Voronoi diagram actually takes only $O(h)$ expected time to construct after we have the points on the convex hull in sorted order. To see this, we apply backwards analysis. We consider the situation after the insertion of the cell of p_i. We observe that if the cell of p_i has k edges on its boundary, then the traversal performed to trace this cell visited k cells in the farthest-point Voronoi diagram of $\{p_1, \ldots, p_{i-1}\}$, and visited at most $4k - 6$ boundary edges of these cells in total.

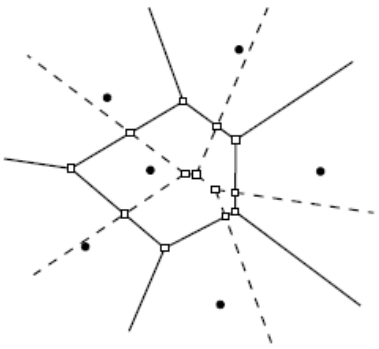
The farthest-point Voronoi diagram of $\{p_1, \ldots, p_i\}$ has at most $2i - 3$ edges (see Exercise 7.14), each used by two cells. Since every point of $\{p_1, \ldots, p_i\}$ has the same probability of having been the last one added, the expected size of the cell of p_i is less than four. Hence, the expected time needed for each insertion is $O(1)$, and the algorithm runs in $O(h)$ expected time. □

Now we return to the problem of computing the smallest-width annulus. Suppose that the smallest-width annulus is such that C_{inner} contains at least three points of P. Then its center is a vertex of the normal Voronoi diagram of P. Similarly, if the smallest-width annulus is such that C_{outer} contains at least three points of P, its center is a vertex of the farthest-point Voronoi diagram of P. Finally, if the smallest-width annulus is such that C_{inner} and C_{outer} both contain two points of P, then its center must lie on an edge of the Voronoi diagram and on an edge of the farthest-point Voronoi diagram simultaneously. This means that we can obtain a reasonably small set of points that must contain the center of a smallest-width annulus.

To do this, we generate the vertices of the *overlay* of the Voronoi diagram and the farthest-point Voronoi diagram. The vertices of the overlay are exactly the candidate centers of the smallest-width annulus, covering all three cases. We don't really need to compute the overlay itself. Once we know a vertex and the four points that determine C_{inner} and C_{outer}, we can compute the smallest-width annulus of those four points directly in $O(1)$ time. This is a candidate for the smallest-width annulus.

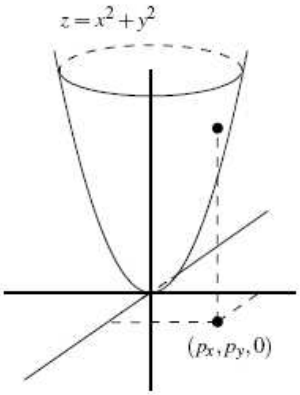
The whole algorithm to compute the smallest-width annulus of a set P of n points in the plane is as follows. Compute the Voronoi diagram and the farthest-point Voronoi diagram of P. For each vertex of the farthest-point Voronoi diagram, determine the point of P that is closest. For each vertex of the normal Voronoi diagram, determine the point of P that is farthest. This gives us $O(n)$ sets of four points that define the candidate annuli in the first and second cases. Next, for every pair of edges, one from each of the diagrams, test if they intersect. If so, we have another set of four points that forms a candidate annulus. For all candidates of all three types, choose the one that gives the smallest-width annulus as the solution.

Theorem 7.15 *Given a set P of n points in the plane, the smallest-width annulus (and the roundness) can be determined in $O(n^2)$ time using $O(n)$ storage.*



7.5 Notes and Comments

Although it is beyond the scope of this book to give an extensive survey of the history of Voronoi diagrams it is appropriate to make a few historical remarks. Voronoi diagrams are often attributed to Dirichlet [148]—hence the name *Dirichlet tessellations* that is sometimes used—and Voronoi [379, 380]. They can be found in Descartes's treatment of cosmic fragmentation in Part III of his *Principia Philosophiae*, published in 1644. In the twentieth century, the Voronoi diagram was rediscovered several times. In biology this even happened twice in a very short period. In 1965 Brown [75] studied the intensity of trees in a forest. He defined the *area potentially available* to a tree, which was in fact the Voronoi cell of that tree. One year later Mead [272] used the same concept for plants, calling the Voronoi cells *plant polygons*. Now, there is an impressive amount of literature concerning Voronoi diagrams and their applications in all kinds of research areas. The book by Okabe et al. [297]



contains an ample treatment of Voronoi diagrams and their applications. We confine ourselves in this section to a discussion of the various aspects of Voronoi diagrams encountered in the computational geometry literature.

In this chapter we have proved some properties of the Voronoi diagram, but it has many more. For example, if one connects all the pairs of sites whose Voronoi cells are adjacent then the resulting set of segments forms a triangulation of the point set, called the Delaunay triangulation. This triangulation, which has some very nice properties, is the topic of Chapter 9.

There is a beautiful connection between Voronoi diagrams and convex polyhedra. Consider the transformation that maps a point $p = (p_x, p_y)$ in $\mathbb{E}^2$ to the non-vertical plane $h(p) : z = 2p_x x + 2p_y y - (p_x^2 + p_y^2)$ in $\mathbb{E}^3$. Geometrically, $h(p)$ is the plane that is tangent to the unit paraboloid $\mathcal{U} : z = x^2 + y^2$ at the point vertically above $(p_x, p_y, 0)$. For a set P of point sites in the plane, let $H(P)$ be the set of planes that are the images of the sites in P. Now consider the convex polyhedron $\mathcal{P}$ that is the intersection of all positive half-spaces defined by the planes in $H(P)$, that is, $\mathcal{P} := \bigcap_{h \in H(P)} h^+$, where h^+ denotes the half-space above h. Surprisingly, if we project the edges and vertices of the polyhedron vertically downwards onto the xy-plane, we get the Voronoi diagram of P [167]. See Chapter 11 for a more extensive description of this transformation. A similar transformation exists for the farthest-point Voronoi diagram.

We have studied Voronoi diagrams in their most basic setting, namely for a set of point sites in the Euclidean plane. The first optimal $O(n \log n)$ time algorithm for this case was a divide-and-conquer algorithm presented by Shamos and Hoey [350]; since then, many other optimal algorithms have been developed. The plane sweep algorithm that we described is due to Fortune [183]. Fortune's original description of the algorithm is a little different from ours, which follows the interpretation of the algorithm given by Guibas and Stolfi [203].

Voronoi diagrams can be generalized in many ways [28, 297]. One generalization is to point sets in higher-dimensional spaces. In $\mathbb{E}^d$, the maximum combinatorial complexity of the Voronoi diagram of a set of n point sites (the maximum number of vertices, edges, and so on, of the diagram) is $\Theta(n^{\lceil d/2 \rceil})$ [239] and it can be computed in $O(n \log n + n^{\lceil d/2 \rceil})$ optimal time [93, 133, 346]. The fact that the dual of the Voronoi diagram is a triangulation of the set of sites, and the connection between Voronoi diagrams and convex polyhedra as discussed above still hold in higher dimensions.

Another generalization concerns the metric that is used. In the L_1-metric, or Manhattan metric, the distance between two points p and q is defined as

$$\text{dist}_1(p, q) := |p_x - q_x| + |p_y - q_y|,$$

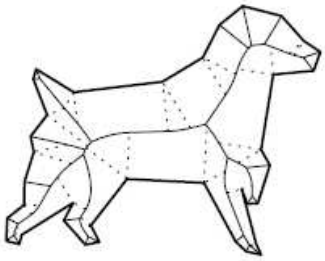
the sum of the absolute differences in the x- and y-coordinates. In a Voronoi diagram in the L_1-metric, all edges are horizontal, vertical, or diagonal (at an angle of 45° to the coordinate axes). In the more general L_p-metric, the distance between two points p and q is defined as

$$\text{dist}_p(p, q) := \sqrt[p]{|p_x - q_x|^p + |p_y - q_y|^p}.$$

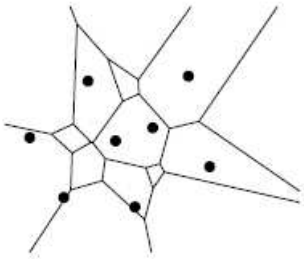
Note that the L_2-metric is simply the Euclidean metric. There are several papers dealing with Voronoi diagrams in these metrics [118, 248, 252]. One can also define a distance function by assigning a weight to each site. Now the distance from a site to a point is the Euclidean distance to the point, plus its additive weight. The resulting diagrams are called weighted Voronoi diagrams [183]. The weight can also be used to define the distance from a site to a point as the Euclidean distance times the weight. Diagrams based on this multiplicatively weighted distance are also called weighted Voronoi diagrams [29]. Power diagrams [25, 26, 27, 30] are another generalization of Voronoi diagrams where a different distance function is used. It is even possible to drop the distance function altogether and define the Voronoi diagram in terms only of bisectors between pairs of sites. Such diagrams are called abstract Voronoi diagrams [240, 241, 242, 274].

Other generalizations concern the shape of the sites. We have seen the Voronoi diagram of a set of disjoint line segments in this chapter. We discussed the application of this diagram to motion planning using the retraction technique; Chapter 13 discusses motion planning in general.

An important special case of the Voronoi diagram of line segments is the Voronoi diagram of the edges of a simple polygon, interior to the polygon itself. Since the edges share endpoints, there can be whole regions inside the polygon where two edges are equally close. This occurs at every reflex vertex of the polygon. The Voronoi diagram is the subdivision of the interior of the polygon into faces where one or two edges are the closest. This Voronoi diagram is also known as the medial axis or skeleton, and it has applications in shape analysis [366, 377]. The medial axis can be computed in time linear in the number of edges of the polygon [123].



Instead of partitioning the space into regions according to the closest sites, one can also partition it according to the k closest sites, for some $1 \leqslant k \leqslant n-1$. The diagrams obtained in this way are called higher-order Voronoi diagrams, and, for given k, the diagram is called the order-k Voronoi diagram [6, 31, 70, 98]. Note that the order-1 Voronoi diagram is nothing more than the standard Voronoi diagram. The order-$(n-1)$ Voronoi diagram is the farthest-point Voronoi diagram, because the Voronoi cell of a point p_i is now the region of points for which p_i is the farthest site. The maximum complexity of the order-k Voronoi diagram of a set of n point sites in the plane is $\Theta(k(n-k))$ [249]. Currently the best known algorithms for computing the order-k Voronoi diagram run in $O(n\log^3 n + nk)$ time [6] and in $O(n\log n + nk2^{c\log^* k})$ time [326], where c is a constant.



The farthest-point Voronoi diagram takes $O(n\log n)$ time to compute, but if the points are in convex position and are given in the order along the convex hull, then there exists a simple $O(n)$ expected-time algorithm [116], given in this chapter, and also an $O(n)$ time deterministic algorithm [11]. Testing the roundness of an object or set of points is a problem that arises in metrology, the science of measurement. Several definitions of roundness exist, the one used in this chapter being the most widely accepted one. A quadratic-time

algorithm for the roundness problem was given by Ebarra et al. [155]. A complex, subquadratic-time algorithm was suggested by Agarwal and Sharir [9]. In special cases that correspond to point sets that may occur in practice, linear-time or near-linear-time algorithms exist [52, 142, 187]. A survey of computational metrology has been given by Yap and Chang [396].

7.6 Exercises

7.1 Prove that for any $n > 3$ there is a set of n point sites in the plane such that one of the cells of $\mathrm{Vor}(P)$ has $n-1$ vertices.

7.2 Show that Theorem 7.3 implies that the average number of vertices of a Voronoi cell is less than six.

7.3 Show that $\Omega(n \log n)$ is a lower bound for computing Voronoi diagrams by reducing the sorting problem to the problem of computing Voronoi diagrams. You can assume that the Voronoi diagram algorithm should be able to compute for every vertex of the Voronoi diagram its incident edges in cyclic order around the vertex.

7.4 Prove that the breakpoints of the beach line, as defined in Section 7.2, trace out the edges of the Voronoi diagram while the sweep line moves from top to bottom.

7.5 Give an example where the parabola defined by some site p_i contributes more than one arc to the beach line. Can you give an example where it contributes a linear number of arcs?

7.6 Give an example of six sites such that the plane sweep algorithm encounters the six site events before any of the circle events. The sites should lie in general position: no three sites on a line and no four sites on a circle.

7.7 Do the breakpoints of the beach line always move downwards when the sweep line moves downwards? Prove this or give a counterexample.

7.8 Write a procedure to compute a big enough bounding box from the incomplete doubly-connected edge list and the tree $\mathcal{T}$ after the sweep is completed. The box should contain all sites and all Voronoi vertices.

7.9 Write a procedure to add all cell records and the corresponding pointers to the incomplete doubly-connected edge list after the bounding box has been added. That is, fill in the details of line 8 of Algorithm VORONOIDIAGRAM.

7.10 Let P be a set of n points in the plane. Give an $O(n \log n)$ time algorithm to find two points in P that are closest together. Show that your algorithm is correct.

Section 7.6
EXERCISES

7.11 Let P be a set of n points in the plane. Give an $O(n \log n)$ time algorithm to find for each point p in P another point in P that is closest to it.

7.12 Let the Voronoi diagram of a point set P be stored in a doubly-connected edge list inside a bounding box. Give an algorithm to compute all points of P that lie on the boundary of the convex hull of P in time linear in the output size. Assume that your algorithm receives as its input a pointer to the record of some half-edge whose origin lies on the bounding box.

7.13 For each of the ten breakpoints shown in Figure 7.5, determine which of the five types it corresponds to.

7.14 Show that the farthest point Voronoi diagram on n points in the plane has at most $2n - 3$ (bounded or unbounded) edges. Also give an exact bound on the maximum number of vertices in the farthest point Voronoi diagram.

7.15 Show that the smallest width annulus cannot be constructed with randomized incremental construction. To this end, show that a point p_i can be added to a set P_{i-1} that does not lie in the minimum width annulus, but does not lie on the boundary of the smallest width annulus of $P_i := P_{i-1} \cap \{p_i\}$.

7.16 Show that for some set P of n points, there can be $\Omega(n^2)$ intersections between the edges of the Voronoi diagram and the farthest site Voronoi diagram.

7.17 Show that if there are only $O(n)$ intersections between the edges of the Voronoi diagram and the farthest site Voronoi diagram, then the smallest width annulus can be computed in $O(n \log n)$ expected time.

7.18* In the Voronoi assignment model the goods or services that the consumers want to acquire have the same market price at every site. Suppose this is not the case, and that the price of the good at site p_i is w_i. The trading areas of the sites now correspond to the cells in the weighted Voronoi diagram of the sites (see Section 7.5), where site p_i has an additive weight w_i. Generalize the sweep line algorithm of Section 7.2 to this case.

7.19* Suppose that we are given a subdivision of the plane into n convex regions. We suspect that this subdivision is a Voronoi diagram, but we do not know the sites. Develop an algorithm that finds a set of n point sites whose Voronoi diagram is exactly the given subdivision, if such a set exists.