

# Viaje de nave espacial en un espacio con asteroides

## usando RRT

### Proyecto robótica

Jesús Bertani y Josué Pérez  
Lic. en Computación Matemática

Universidad de Guanajuato  
5 de diciembre de 2023

---

#### Resumen.

El proyecto consiste en la creación de un espacio tridimensional que incluye asteroides, una nave y un campo de fuerza. El objetivo principal es probar un algoritmo de planificación de movimiento RRT. La implementación del algoritmo se llevó a cabo siguiendo las enseñanzas de clase, y se utilizó el detector de colisiones CGAL para mejorar la precisión. Este enfoque proporciona una plataforma para evaluar la eficacia del algoritmo RRT en entornos complejos.

## Desarrollo teórico

Tomando como base el ejemplo mostrado en el artículo Randomized Kinodynamic Planning visto en el curso, consideramos un modelo con 12 grados de libertad en un entorno sin gravedad. Cada estado consiste de los siguientes parámetros:

$$\begin{aligned}\mathbf{p} &= [p_x, p_y, p_z]^T && \text{Posición global del centro de masa,} \\ \mathbf{q} &= [q_\theta, q_x, q_y, q_z]^T && \text{Cuaternión unitario representando su rotación en } SO(3), \\ \mathbf{v} &= [v_x, v_y, v_z]^T && \text{Velocidad lineal,} \\ \mathbf{w} &= [w_x, w_y, w_z]^T && \text{Velocidad angular.}\end{aligned}$$

Con lo que construimos un vector de estados

$$x(t) = \begin{pmatrix} p(t) \\ q(t) \\ v(t) \\ w(t) \end{pmatrix}. \quad (1)$$

El vector de estados consiste de 13 números reales pero el espacio es de dimensión 12 dado que el cuaternión que representa la rotación debe ser unitario. Además cada control  $u \in \mathcal{U}$  define una pareja de fuerzas  $(F, \tau)$  que actúa en el centro de masa del cuerpo. La ecuación del sistema de movimiento fue definida como

$$\dot{x}(t) = f(x(t), u(t)) = \begin{pmatrix} \dot{p}(t) \\ \dot{q}(t) \\ \dot{v}(t) \\ \dot{w}(t) \end{pmatrix} = \begin{pmatrix} v(t) \\ \frac{1}{2}\hat{\omega}(t) \cdot q(t) \\ F/M \\ R(t)I^{-1}R(t)^T\tau \end{pmatrix}, \quad (2)$$

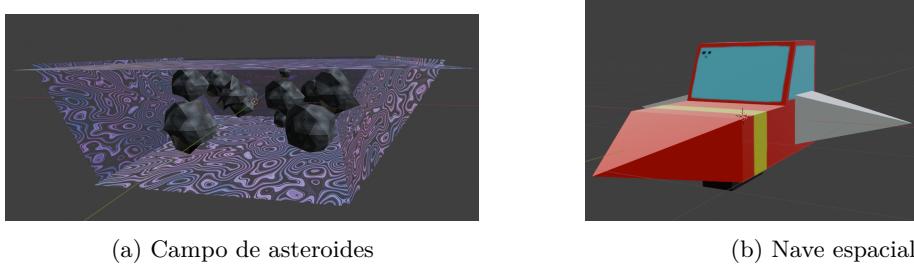


Figura 0.1: Disenños creados

donde  $R(t)$  es la rotación que resulta de convertir el cuaternion  $q(t)$  a una matriz de rotación,  $M$  es la masa del objeto e  $I$  el tensor de inercia.

Las parejas de controles mencionadas se obtienen de los siguientes espacios:

$$\begin{aligned} F \in U_F &:= \{(0, 0, -0.5), (0, 0, 0.5), (0, 0, 0)\}, \\ \tau \in U_\tau &:= \{(0.01, 0, 0), (-0.01, 0, 0), \\ &\quad (0, 0.01, 0), (0, -0.01, 0), \\ &\quad (0, 0, 0.01), (0, 0, -0.01), (0, 0, 0)\}, \end{aligned}$$

Donde  $U_F$  contiene los vectores que representan un movimiento lineal hacia delante y atrás o no avanzar en esta dirección. Mientras que el conjunto  $U_\tau$  guarda las posibilidades para rotar en cada uno de los ejes en sentido horario y anti-horario así como mantener la misma dirección.

Utilizamos la función de distancia

$$\rho(x_1, x_2) = 0.05\|p_1 - p_2\|^2 + 5(1 - |q_1 \cdot q_2|)^2 + 0.3\|v_1 - v_2\|^2 + 0.5\|w_1 - w_2\|^2,$$

donde los coeficientes constantes de cada uno de los términos se definieron de esa manera pues de forma empírica dieron un buen resultado.

Para la aplicación de los controles definimos un valor para  $\Delta t$  y usamos el método de paso fijo de Euler para integración numérica con el cuál cambiamos de un estado al siguiente.

## Desarrollo práctico

Comenzamos con la creación del espacio para darle realismo a la simulación. Utilizando Blender se creó un campo de fuerza, asteroides y una nave que trataría de escapar de este lugar esquivando los obstáculos. Para la creación de estos objetos partimos de cuerpos 3D básicos que se combinaron y deformaron para después agregar texturas, terminando con archivos *.obj* que fueron los usados para la simulación. Estos fueron agregados en la carpeta que contiene el proyecto. En la Figura 0.1 se muestran los modelos.

Una vez que se tuvieron los modelos se implementó el algoritmo *RRT* basado en el pseudocódigo visto en clase 0.2. La implementación se realizó en C++ y se crearon distintas

estructuras correspondientes a un estado del robot, controles y los nodos que definen la forma del RRT. Cada uno con los métodos necesarios para el funcionamiento del algoritmo.

Comenzamos con la estructura **State** que cuenta con las funciones:

- *State(p, q, v, w)*, inicializa un estado donde se le indican los valores  $p, q, v, w$  de la ecuación (1)
- *State()*, constructor donde los valores de  $p, q, v$  y  $w$  se generan de manera aleatoria, donde los rangos están ajustados a las condiciones del problema.
- *distance(s)*, calcula la distancia entre el estado actual y el pasado como parámetro. Utiliza la métrica mostrada en la primera sección.
- *nearest\_node(root)*, encuentra cuál es el nodo del *RRT* con raíz *root* que tiene el estado más cercano al actual.
- *in\_collision()*, determina si un estado se encuentra en colisión o no con los obstáculos. En esta función entraremos más a detalle después, cuando se hable del detector de colisiones utilizado para el proyecto.
- *finished()*, indicadora si el estado actual ya llegó a la región fijada.

Por otra parte, la estructura **Controls** tiene las siguientes funciones:

- *Controls()*, que crea una instancia de la estructura con todos los controles incializados en cero.
- *Controls(F, τ)*, inicializa los controles de acuerdo a los vectores  $F \in U_F$  y  $\tau \in U_\tau$ .
- *to\_state(prev\_status)*, recibe como parámetro el estado actual del robot y con base en los controles que se tienen determina cual sería la posición del robot en el siguiente instante de tiempo. Siguiendo la ecuación (2) y el método de paso fijo de Euler.

Finalmente, la estructura **Node** que representa los nodos del árbol y cada uno cuenta con un estado (State), el padre de dicho nodo en el árbol (Node) y el conjunto de controles con el cual se llegó de este padre al estado actual (Controls). La estructura cuenta con los miembros

- *Node(state)*, inicializa un nodo solo con el estado. Esta inicialización es útil para la raíz que no tiene un parent o controles con la cual se llegó a ella.
- *Node(state, controls, parent)*, crea un nodo con las tres características mencionadas en la descripción de la estructura.
- *create\_next\_node(random\_state)*, partiendo del estado actual se generan controles aleatorios, los cuales generan estados y dentro de estos estados agregamos al arbol el que sea más cercano al estado random pasado como parámetro.
- *finished()*, determina si el estado del nodo actual llegó a la meta.

<b>Algorithm 3:</b> RRT
<pre> 1 <math>V \leftarrow \{x_{\text{init}}\}; E \leftarrow \emptyset;</math> 2 <b>for</b> <math>i = 1, \dots, n</math> <b>do</b> 3   <math>x_{\text{rand}} \leftarrow \text{SampleFree};</math> 4   <math>x_{\text{nearest}} \leftarrow \text{Nearest}(G = (V, E), x_{\text{rand}});</math> 5   <math>x_{\text{new}} \leftarrow \text{Steer}(x_{\text{nearest}}, x_{\text{rand}});</math> 6   <b>if</b> <math>\text{ObstacleFree}(x_{\text{nearest}}, x_{\text{new}})</math> <b>then</b> 7     <math>V \leftarrow V \cup \{x_{\text{new}}\}; E \leftarrow E \cup \{(x_{\text{nearest}}, x_{\text{new}})\};</math> 8 <b>return</b> <math>G = (V, E);</math></pre>

Figura 0.2: Pseudocódigo RRT

Estas funciones se encuentran en el archivo *rtt.hpp*, incluido en la carpeta, donde se pueden ver más detalles del código comentados en el mismo. En este mismo archivo es donde se codificó toda la parte teórica mencionada en la sección anterior.

Finalmente, para el cálculo del *RRT* solo hacía falta ver cuando la nave esta en colisión. Para ello se utilizó una adaptación del detector de colisiones de *CGAL* utilizando las librerías *Eigen* y *GLM* de C++. La idea para detectar las colisiones es crear la nave representada por medio de un árbol AABB y se utiliza otro árbol para crear la escena que contiene los obstáculos. Estos dos árboles son los que se utilizan para detectar si los objetos están en colisión. La implementación también puede encontrarse en al carpeta entregada en el archivo *collisions.hpp*.

Con el algoritmo *RRT* y el detector de colisiones se tiene todo lo necesario para realizar las simulaciones. Estas fueron mostradas utilizando OpenGL.

## Experimento

Partiendo del estado inicial

$$x_0 = \begin{pmatrix} (0, 0, 19) \\ (1, 0, 0, 0) \\ (0, 0, 0) \\ (0, 0, 0) \end{pmatrix},$$

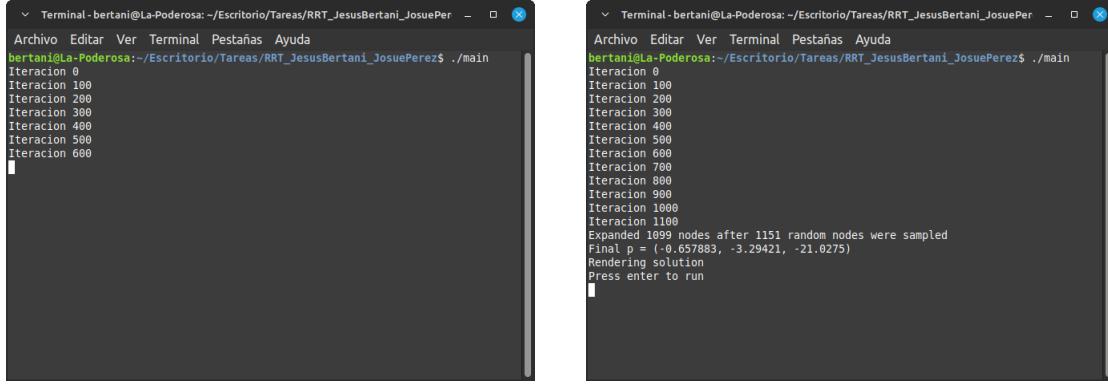
tratamos de salir del campo de fuerza modelado. El campo de fuerza es un conjunto de planos que para usos prácticos podemos decir que abarcan el producto cartesiano de  $[-16, 16] \times [-8, 8] \times [-20, 20]$  de donde obtenemos que matemáticamente escapar de él significa que la coordenada  $z$  de la posición de la nave es menor a  $-21$ . Sin embargo, la región meta la definimos como  $\{(x, y, z) | |x| < 5, |y| < 5, z < -21\}$  para obtener un escape cercano al centro del escenario. Notemos que no decimos nada sobre su orientación por lo que el algoritmo puede encontrar una solución donde sale "de reversa." posiciones extrañas.

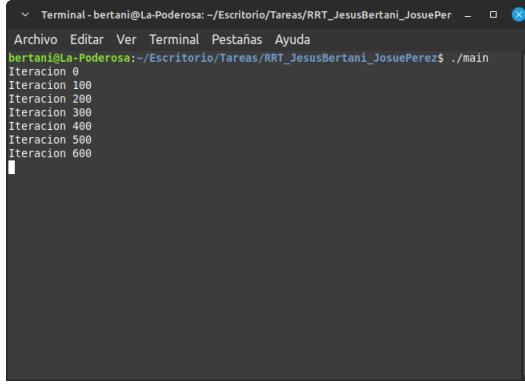
Ahora, el muestreo de estados se hace de la siguiente manera:

- La posición se muestrea uniformemente en  $[-16, 16] \times [-8, 8] \times [-25, 25]$ . Esto para asegurarnos que se muestreen estados más allá de la coordenada  $z = -21$ .
- La orientación siempre es el cuaternio identidad pues queremos que gire cuando sea estrictamente necesario,

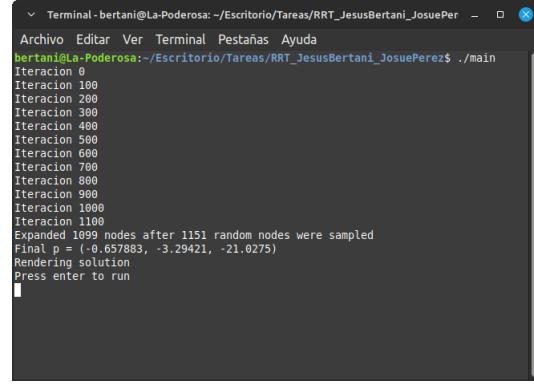
- La velocidad espacial se muestrea uniformemente como un vector con entradas entre -5 y 5,
- La velocidad angular se muestrea uniformemente también como un vector pero con coordenadas entre -2 y 2.

Dadas estas consideraciones, usando  $\Delta t = 0.3$  el algoritmo logró encontrar una trayectoria tras expandir 1099 nodos de los 1151 totales que intentó conectar. Para este caso, la nave logró llegar al punto (-0.657883, -3.29421, -21.0275) que cumple con las condiciones mencionadas. El resultado de esta trayectoria se puede observar en el video adjunto. A continuación algunas capturas del recorrido y el programa corriendo.





(a) Primeras iteraciones



(b) Resultado en consola

Figura 0.3: Retroalimentación del algoritmo en consola

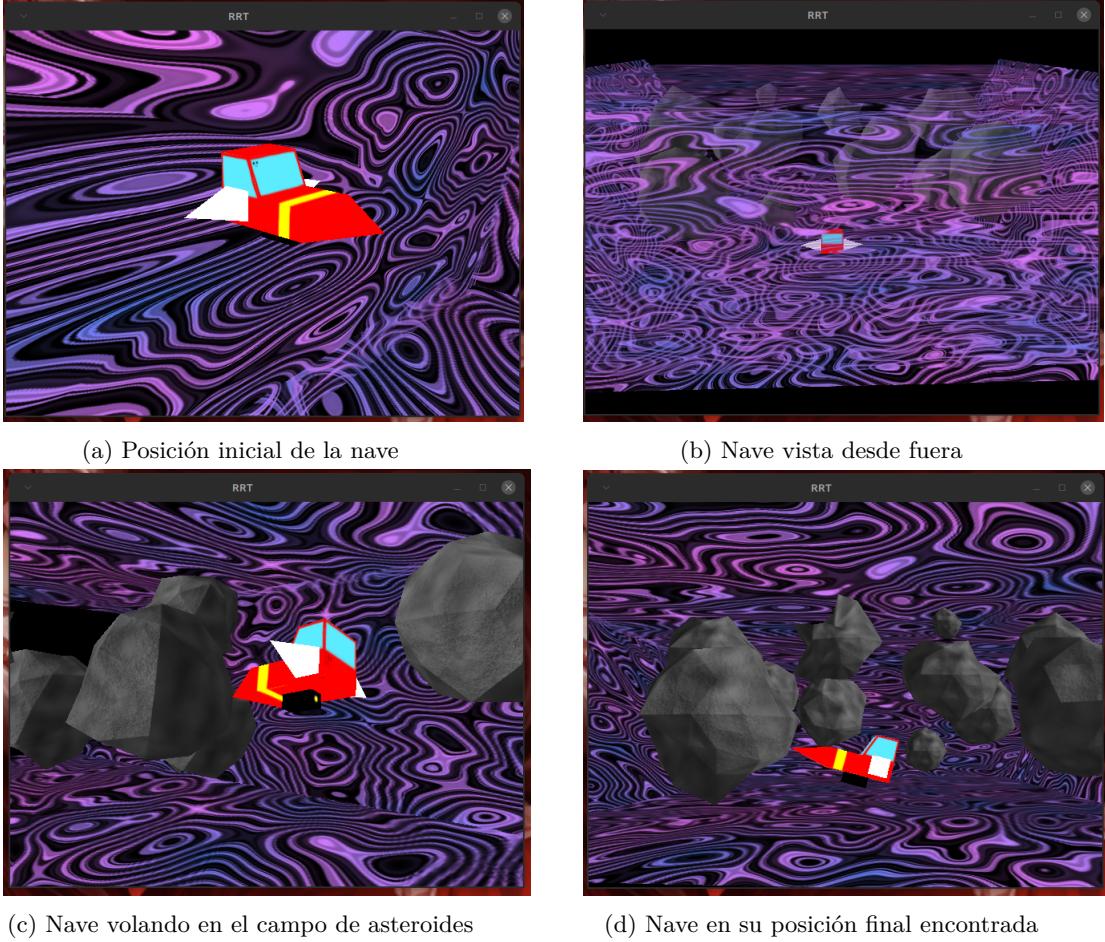


Figura 0.4: Capturas de la trayectoria encontrada

Cabe resaltar que todos estos parámetros pueden ser cambiados dentro del código para hacer nuevos experimentos, cuidando que la posición inicial no esté en colisión para que el problema tenga sentido.

## Conclusiones y aprendizajes

Siguiendo las pautas aprendidas en clase, implementamos el algoritmo RRT como método de planificación de movimiento. Esta implementación se adapta específicamente a entornos complejos, lo que facilita la evaluación de su rendimiento en situaciones más realistas.

Hemos creado un entorno tridimensional que simula un escenario con asteroides y un campo de fuerza, proporcionando un contexto desafiante para evaluar la capacidad del algoritmo RRT. La creación del entorno ha destacado la importancia de considerar los pesos para la

función distancia, el tamaño del paso del tiempo y el muestreo de estados pues se llevaron a cabo muchas pruebas antes de encontrar una combinación de estos parámetros que regresaran una solución satisfactoria.

La integración del detector de colisiones CGAL ha mejorado significativamente la precisión y rapidez en la identificación de colisiones entre la nave y los asteroides, proporcionando un marco más confiable para evaluar la efectividad del algoritmo RRT.

## Bibliografía

- LaValle, S.M., Kuffner J.J. (1999) Randomized kinodynamic planning. *Proceedings 1999 IEEE International Conference on Robotics and Automation (Cat. No.99CH36288C)*, 1 (1), 473-479. <https://doi.org/10.1109/ROBOT.1999.770022>