Universidad Zaragoza

Trabajo Fin de Grado

Diseño de una célula de reparación automática para la detección de defectos de pintura en carrocerías de automóviles.

Technical design of a self-repairing cell for the detection of paint defects on car bodies.

Autor/es

Marta Ramos Herrera

Director/es

Jorge Santolaria Mazo

Grado en Ingeniería de Tecnologías Industriales

CURSO 2024/25

ESCUELA DE INGENIERÍA Y ARQUITECTURA

A mi hermana Cristina por siempre confiar en mí, a mis padres y a mis abuelos por brindarme una grandísima educación y a todos mis amigos que esta titulación me ha permitido conocer.

Resumen ejecutivo : Diseño de una célula de reparación automática para la detección de defectos de pintura en la carrocería de automóviles.

El presente Trabajo de Fin de Grado aborda el diseño y simulación de una célula robotizada para la inspección y reparación automática de defectos de pintura en carrocerías de automóviles. Ante la creciente demanda de acabados de alta calidad y la necesidad de optimizar los procesos en la industria automotriz, este proyecto propone una solución para automatizar una tarea tradicionalmente manual, sujeta a errores y limitaciones humanas.

El objetivo principal ha sido desarrollar un entorno virtual que replique fielmente una célula de trabajo real, garantizando un funcionamiento seguro y eficiente. Para ello, se ha utilizado el software de simulación RoboDK, una herramienta que permite el modelado, la programación y la validación de sistemas robotizados.

El núcleo del proyecto consiste en la creación de un modelo 3D completo de la célula, incluyendo el brazo robótico, sus herramientas, la carrocería y los sistemas auxiliares. Sobre este modelo virtual se ha programado la trayectoria del robot, para asegurar una cobertura completa de la superficie de la carrocería, prestando especial atención a la detección y prevención de colisiones entre el robot y los elementos del entorno.

Como resultado principal, se ha obtenido una simulación funcional y verificada en la herramienta de simulación, que demuestra la viabilidad del concepto. El sistema es capaz de ejecutar un ciclo de trabajo completo de manera autónoma y sin colisiones, sentando las bases para la futura implementación física de la célula.

Este proyecto no solo valida una solución técnica a un desafío industrial relevante, sino que también resalta el potencial de la simulación para diseñar y perfeccionar procesos de automatización complejos, reduciendo eficazmente los costes y errores en el desarrollo.

Executive Summary: Thecnical desing of a self-reparing cell for the detection of Paint defects on car bodies.

This Final Degree Project addresses the design and simulation of a robotic cell for the automated inspection and repair of paint defects on car bodies. In response to the growing demand for high-quality finishes and the need to optimize processes in the automotive industry, this project proposes a solution to automate a task that is traditionally manual and subject to human error and limitations.

The main objective has been to develop a virtual environment that faithfully replicates a real work cell, ensuring safe and efficient operation. To achieve this, RoboDK simulation software was used, a tool that allows for the modeling, programming, and validation of robotic systems.

The core of the project consists of creating a complete 3D model of the cell, including the robotic arm, its tools, the car body, and auxiliary systems. The robot's trajectory was programmed on this virtual model to ensure complete coverage of the car body's surface, with special attention paid to the detection and prevention of collisions between the robot and the surrounding elements.

As the primary result, a functional and verified simulation has been achieved in RoboDK that demonstrates the feasibility of the concept. The system is capable of executing a complete work cycle autonomously and without collisions, laying the groundwork for the future physical implementation of the cell. This project not only validates a technical solution to a relevant industrial challenge but also highlights the potential of simulation to design and refine complex automation processes, effectively reducing development cost and errors.

Contenido

Resumen ejecutivo : Diseño de una célula de reparación automática para la detección de defectos de pintura en la carrocería de automóviles. 4

Executive Summary: Thecnical desing of a self-reparing cell for the detection of Paint defects on car bodies. 5

1. Introducción.. 7

1.1. Motivación. 7

1.2. Objetivos.. 7

2. Contexto Tecnológico y Antecedentes: Proceso de inspección de defectos superficiales. 10

2.1. Componentes de la célula de detección de defectos. 10

3. Introducción.. 13

4. Descripción de la célula. 13

4.1. Unidad de posicionamiento y sujección del chasis. 14

4.2. Sistema robótico de reparación. 15

5. Montaje de la célula. 20

5.1. Sistemas de referencia 20

5.2. Posicionamiento de los sólidos 21

6.Desarrollo de la simulación. 23

6.1. Planteamiento del problema y tratamiento de los datos de entrada. 23

6.2. Metodología de Generación de Trayectorias y Validación por Simulación....29

7. Estudio económico. 37

7.1. Definición del escenario de estudio. 37

7.2. Análisis económico de la situación actual (Proceso Manual). 38

7.3. Análisis económico de la propuesta (Célula robotizada)........................ 39

7.4. Evaluación de Rentabilidad e indicadores financieros. 41

7.5. Conclusión del Estudio económico. 44

8. Trabajo futuro. 45

8.1. Futuras líneas de investigación y desarrollo. 45

9. Conclusiones. 47

10.Bibliografía. 48

Capítulo 1

1. Introducción.

1.1. Motivación.

La industria automotriz se encuentra en una evolución constante, impulsada por una competencia global que exige los más altos estándares de calidad y eficiencia.

Dentro de este contexto, el acabado superficial de la carrocería no es un mero detalle técnico, sino uno de los principales atributos de calidad percibidos por el cliente. Un acabado de pintura impecable es sinónimo de un producto premium, y cualquier defecto, por mínimo que sea, puede impactar negativamente en la valoración del vehículo y en la imagen de la marca.

Actualmente, la detección y reparación de defectos en la pintura persiste como una labor predominantemente manual en numerosas plantas de producción, lo que conlleva una inevitable variabilidad en el resultado. No obstante, el paradigma de la Industria 4.0 y la automatización inteligente se plantea como la solución integral a esta problemática.

La integración de sistemas robotizados de precisión permite trascender las limitaciones del trabajo manual, garantizando no solo una calidad homogénea y repetible sino también, la optimización de los tiempos de ciclo y una reducción significativa de los costes operativos.

En este marco tecnológico se fundamenta la motivación principal de este Trabajo de Fin de Grado, ya que este proyecto se concibe como el paso estratégico previo a la automatización; abordando el diseño, la programación y la viabilidad de la célula robotizada, a través de la creación de un gemelo digital propuesto.

1.2. Objetivos.

El objetivo principal de este proyecto es el diseño y simulación de una célula de trabajo robotizada en RoboDK, capaz de generar y ejecutar trayectorias precisas y cinemáticamente válidas, para la reparación de defectos de pintura sobre la carrocería de un automóvil, a partir de un conjunto de coordenadas de entrada proporcionadas por un sistema de detección previo.

Para la consecución de este propósito principal, se establecen los siguientes objetivos específicos, que desglosan los hitos técnicos necesarios para la validación del proyecto:

I. **Modelar el entorno de trabajo virtual:** Diseñar y construir una réplica digital 3D de la célula de reparación en el software RoboDK. Esto implica integrar y posicionar correctamente todos los componentes; el brazo robótico, la herramienta de reparación, el modelo de la carrocería y los sistemas auxiliares necesarios.

II. **Implementar el sistema de coordenadas de defectos:** Desarrollar el método para importar e interpretar el listado de coordenadas de los defectos. Este sistema deberá ser capaz de traducir cada coordenada recibida en un punto de destino (o *target*) alcanzable por el robot dentro del espacio de trabajo.

III. **Programar la trayectoria de reparación:** Crear el algoritmo de movimiento del robot que, para cada defecto detectado, genere una trayectoria óptima y segura. Esta trayectoria deberá incluir las fases de aproximación al punto, la ejecución de la tarea de reparación específica y la retirada a una posición segura.

IV. **Validar y verificar la simulación:** Ejecutar simulaciones completas del proceso para verificar que el robot es capaz de alcanzar todos los puntos de defecto, sin colisiones. Se deberá garantizar la integridad de la célula, asegurando que no exista ningún tipo de interferencia entre el robot, sus herramientas y la carrocería durante todo el ciclo de trabajo.

V. **Validación económica del proyecto:** Desarrollar un estudio económico comparativo que valide la solución robotizada propuesta. Se analizará el impacto económico de la automatización frente al proceso manual, calculando los plazos de recuperación de la inversión y el ahorro operativo proyectado.

Este hito cierra el proyecto complementando la validación técnica y certificando que el sistema diseñado es una solución rentable y competitiva para el sector industrial.

Una vez establecidos los objetivos principales de este proyecto, se muestra a continuación la visión general del mismo mediante el siguiente diagrama. Este ilustra el flujo de trabajo completo que se abordará, su finalidad es servir como guía visual para esclarecer los conceptos clave, los hitos principales y el alcance total del trabajo que se detalla en los siguientes apartados.

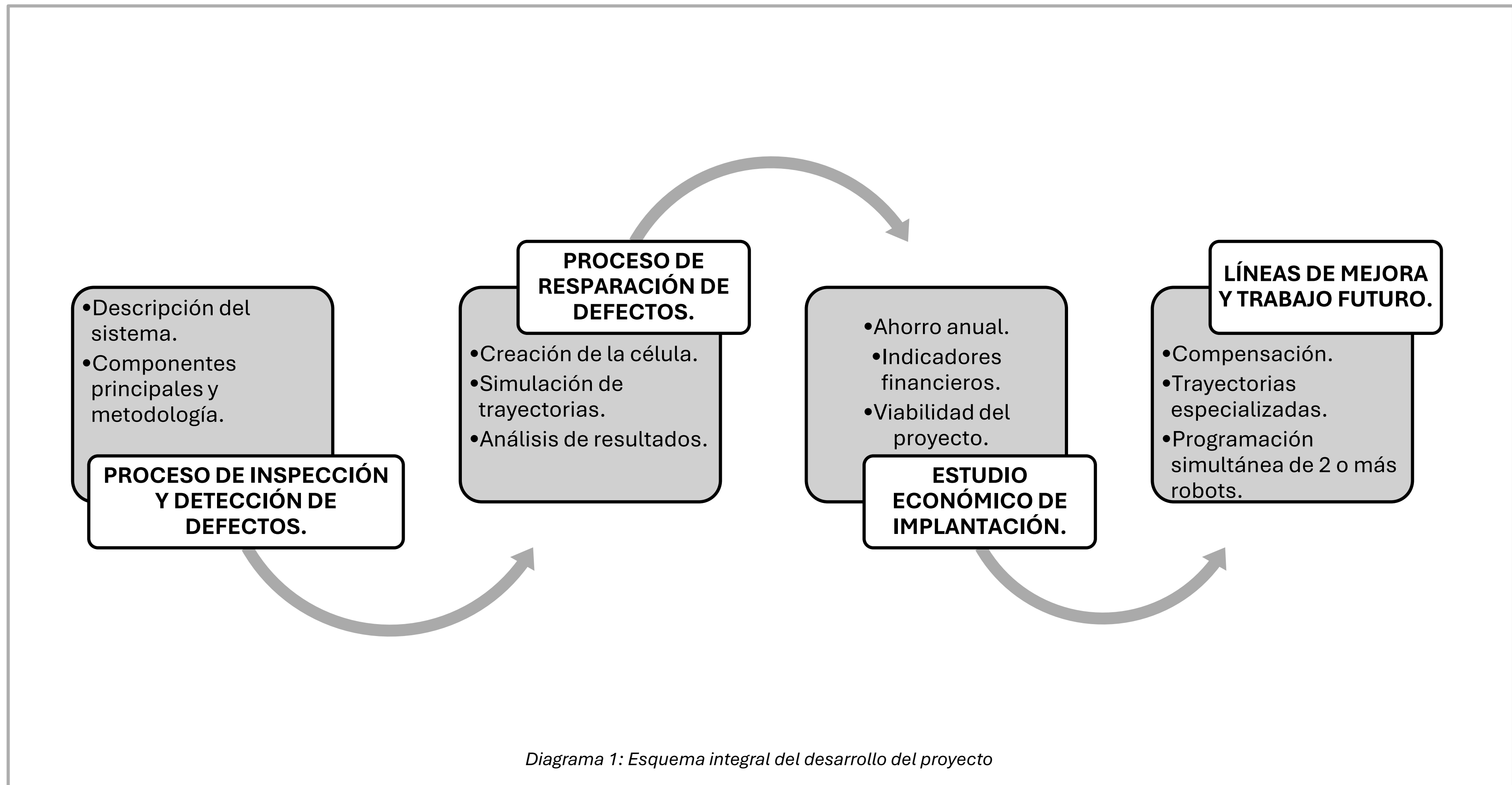


Diagrama 1: Esquema integral del desarrollo del proyecto

Capítulo 2

2. Contexto Tecnológico y Antecedentes: Proceso de inspección de defectos superficiales.

Antes de abordar la metodología de reparación, es necesario establecer el punto de partida, es decir; *¿cómo se identifican los defectos en el chasis de un automóvil?*

En la industria automotriz moderna, este proceso ha evolucionado hacia sistemas automatizados de alta precisión, lo que plantea la cuestión; *¿qué tecnologías se emplean actualmente en una línea de producción para esta tarea?*

Para responder a esta pregunta y contextualizar adecuadamente este trabajo, se han analizado las tecnologías de inspección automatizada, ejemplificadas en el proyecto "Eagle Eye" de J3D Vision, que ha permitido definir el estado del arte en este campo. Por ello, en el presente apartado se describirán tanto sus componentes principales como el proceso operativo que se lleva a cabo en dicha célula de inspección. El objetivo primordial de esta es garantizar una detección objetiva, repetible y de alta velocidad de los defectos presentes en la carrocería del vehículo, constituyendo así un paso esencial para la eficacia de la etapa de reparación que le sucede.

2.1. Componentes de la célula de detección de defectos.

El propósito fundamental de esta célula reside en la inspección integral de la superficie del vehículo a la velocidad de la línea de producción.

Con el objeto de proporcionar el contexto técnico necesario para el desarrollo de este trabajo, a continuación, se detallará la arquitectura del sistema y se esquematizará su principio de funcionamiento, estableciendo así las bases operativas de la detección de defectos.

I. Estructura principal y sistema de visión.
   El componente principal es un túnel o estructura que rodea al vehículo. Esta estructura no contiene partes móviles y su función es albergar los sistemas de

captura de datos y permitir que el vehículo sea analizado en continuo movimiento, sin detener la línea de producción.

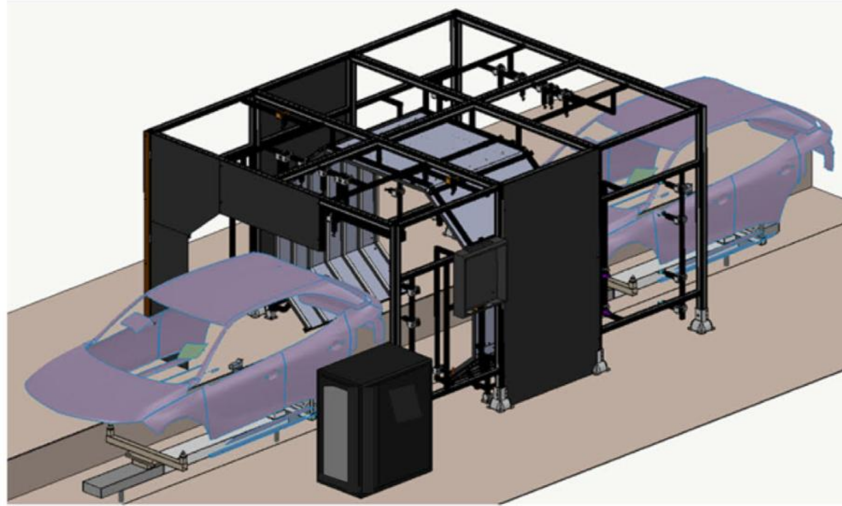


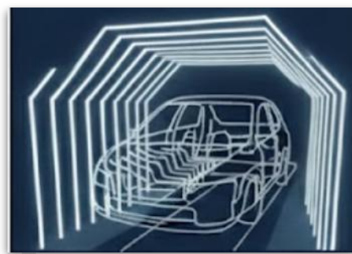
Ilustración 1:Célula de inspección de defectos "Eagle Eye" de J3D Vision.

Dentro de esta estructura se alojan los dos sistemas clave para la detección: el sistema de cámaras y el sistema de iluminación. El primero consiste en un conjunto de cámaras estáticas (cuyo número puede variar de 24 y 36 según la célula), posicionadas estratégicamente para obtener una cobertura visual completa de la geometría del vehículo.

El segundo es un sistema de iluminación basado en la deflectometría, que proyecta un patrón de luz estructurada sobre la superficie. Dado que la carrocería actúa como un espejo, las cámaras no buscan el defecto en sí, sino el reflejo del patrón. Una superficie perfecta devuelve un reflejo predecible, pero cualquier imperfección, altera la curvatura local y distorsiona el patrón reflejado, que detecta el software localizando así la imperfección.

II. <u>Proceso de inspección en movimiento.</u>

El esquema adjunto ilustra la inspección que precede a la reparación, la cual se divide en las siguientes cuatro fases:

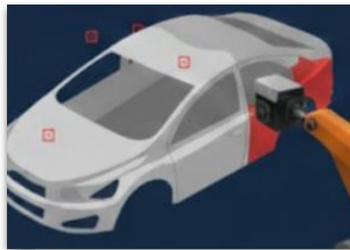


1. INGRESO AL TÚNEL

- La carrocería se inspecciona en movimiento, sin necesidad de detenerse.
- Adapración a la velocidad de la línea (150 vehículos/hora)

3. PROCESAMIENTO Y DETECCIÓN

- El software procesa las imagenes en tiempo real.
- Detecta y clasifica los defectos.



2. CAPTURA Y COMPENSACIÓN

-El sistema de cámaras captura las imagenes mientras el vehículo avanza.

4. GENERACIÓN DE RESULTADOS

- Crea un mapa de defectos para cada vehículo.

-Envia los resultados a la la célula de reparación de defectos posterior.

Ilustración 2: Descripción de detección de defectos mediante la célula de inspección.

La importancia de contextualizar esta etapa dentro del presente trabajo radica en el objetivo final de la célula de inspección: la obtención de un listado de targets exhaustivo y parametrizado. Este sistema procesa los datos capturados para entregar no solo la ubicación espacial de cada defecto, sino también información crítica sobre su naturaleza y tamaño (tipificando el fallo como cráter, grano, suciedad, etc.).

Esta caracterización detallada es el pilar sobre el que se sustenta la metodología de este proyecto, ya que es precisamente la naturaleza del defecto la que fundamenta y condiciona las trayectorias de reparación posteriores. Sin esta clasificación previa, sería inviable definir la estrategia cinemática adecuada para abordar y corregir cada defecto de manera óptima.

Capítulo 3

3. Introducción

En el capítulo anterior se han analizado los estudios previos relativos al proceso de inspección automatizada de defectos superficiales. Este proceso de detección no es un fin en sí mismo, sino el paso previo fundamental que proporciona los datos de entrada (localización, tipo y severidad del defecto) para la siguiente etapa; la reparación automatizada.

El valor fundamental de este proyecto radica en su capacidad para vincular la fase de inspección digital con la de reparación robotizada. La clave de esta transformación es la digitalización del criterio de reparación: en lugar de depender de una evaluación humana, subjetiva y variable, el sistema utiliza los datos métricos de la inspección para comandar la célula robotizada. Esto permite cerrar el bucle "detección-corrección" y elevar un proceso manual, a un flujo de trabajo de máxima eficiencia operativa.

Se consigue así, un proceso optimizado, repetible y trazable que reduce drásticamente los tiempos de ciclo y los costes asociados al retrabajo, asegurando un elevado estándar de calidad.

El presente capítulo aborda, por tanto, la metodología y el caso de aplicación de este trabajo, centrándose en el diseño y simulación de la estación encargada de ejecutar dicha reparación. El primer paso para ello es definir conceptualmente el sistema desarrollado.

4. Descripción de la célula.

La célula de trabajo de este proyecto es un modelo de simulación funcional creado en RoboDK, que emula una estación de reparación automatizada para la industria automotriz. Para llevar a cabo su función, la célula se ha dividido conceptual y físicamente en dos unidades principales: una unidad estacionaria, que se encarga de la sujeción y posicionamiento preciso de la pieza de trabajo, y una unidad robotizada, que constituye el elemento activo del sistema y ejecuta las tareas de proceso, detalladas a continuación.

4.1. Unidad de posicionamiento y sujección del chasis.

Esta unidad conforma la infraestructura sobre la que se presenta la carrocería.Su función es recibir el vehículo, posicionarlo y mantenerlo inmóvil durante el ciclo de trabajo para garantizar la repetibilidad del proceso. Está compuesta por:

I. **Skid**: Es un utillaje de transporte sobre el cual se monta y se fija la carrocería de un vehículo para su desplazamiento a lo largo de las distintas estaciones de la línea de producción. En esta célula, el skid se desplaza sobre mesas de rodillos motorizadas. Su propósito principal es dual: por un lado, actúa como la interfaz diseñada para el sistema de transporte, garantizando un guiado correcto dentro de la estación; y por otro, proporciona una base de montaje estable y referenciada para la carrocería. Para asegurar esta referenciación precisa, el skid incorpora cuatro bulones de posicionamiento:
   - Dos bulones delanteros: Su función es registrar y definir la posición exacta del coche.
   - Dos bulones traseros de línea: Sirven como apoyo principal de la carrocería sobre la estructura del skid.

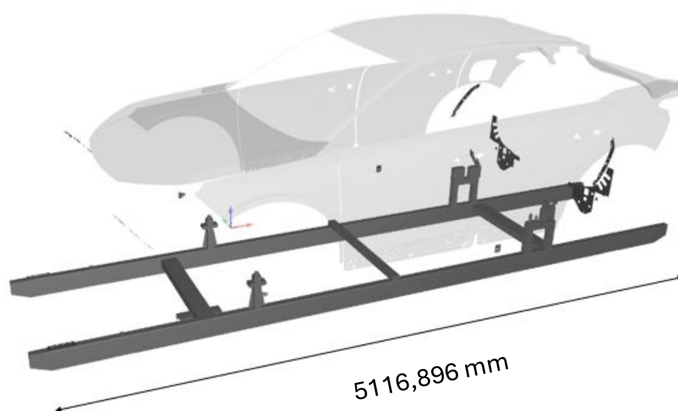


Ilustración 3: Posicionamiento y cota del skid en la herramienta de simulación.

II. **Chasis del vehículo (Pieza de Trabajo):** Es el componente central y objeto del proceso de reparación. Para este proyecto, el modelo de carrocería utilizado corresponde al de un "*Peugeot 3008"*. Se trata de una carrocería con una estructura unificada, que integra el chasis y todos los paneles exteriores en un único conjunto.

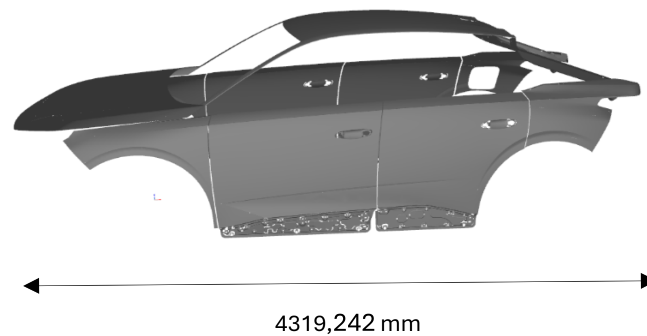


Ilustración 4: Cota horizontal del modelo de estudio "Peugeot 3008".

En el contexto de este proyecto, la carrocería cumple una doble función: por un lado, es la pieza de trabajo que contiene los defectos a reparar y por otro, actúa como la referencia geométrica. Sus superficies definen el espacio en el que el robot debe planificar y ejecutar trayectorias precisas, convirtiendo su correcta digitalización y posicionamiento en un factor crítico de la operación.

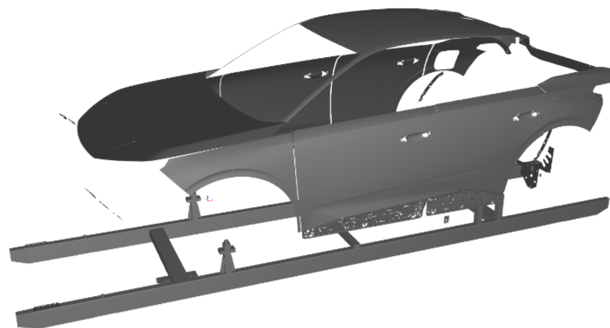


Ilustración 5: Posicionamiento del skid y el chasis en la herramienta de simulación.

4.2. Sistema robótico de reparación.

Esta unidad constituye el conjunto activo y cinemático de la célula, responsable de ejecutar físicamente las tareas de reparación sobre la carrocería. Se compone de una jerarquía de elementos montados entre sí para dotar al sistema de la versatilidad y el alcance necesarios, el sistema se compone de los siguientes componentes:

I. **Eje Lineal (Séptimo Eje):** Es el carril sobre el que se desplaza todo el conjunto del robot. Este componente actúa como un séptimo grado de libertad, extendiendo el espacio de trabajo del robot a lo largo del eje longitudinal del vehículo. Su uso es fundamental para poder acceder a la totalidad de la carrocería con un único robot, optimizando la configuración de la célula. La configuración de este carril es única para cada célula de trabajo, ya que depende de factores como el número de robots que operen en ella y de las condiciones específicas de la instalación.

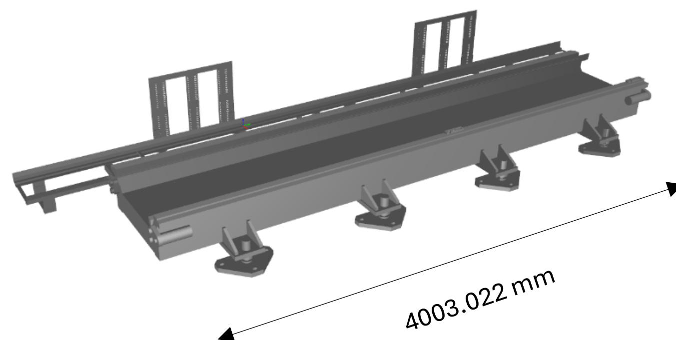


Ilustración 6: Cota longitudinal del séptimo eje y visualización en la herramienta de simulación.

II. **Carro de Traslación:** El componente que se mueve a lo largo del eje lineal. Sobre este carro se monta un pedestal o elevador, cuya función es posicionar la base del robot a la altura de trabajo óptima con respecto a la carrocería, garantizando que pueda operar en una configuración cinemática favorable.

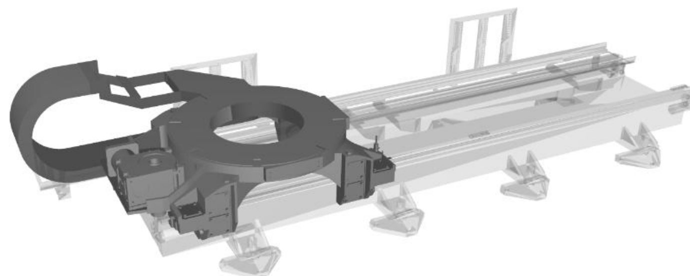


Ilustración 7: Posicionamiento y anclaje del carro de traslación al 7 º eje.

III. **Brazo Robótico (Fanuc R-2000iC/165F):** El componente cinemático principal de la unidad es un robot industrial de seis grados de libertad que actúa como el manipulador principal. Es el elemento encargado de ejecutar

todos los movimientos y materializar físicamente las trayectorias programadas para la reparación de los defectos dentro de la célula.

Ilustración 8: Robot Fanuc R-2000iC/165F, representación en el entorno de simulación frente a la unidad industrial real.

IV. **Herramientas acopladas:** A la brida del brazo robótico se le acoplan diferentes herramientas (*tools*) para ejecutar el proceso de reparación, por lo que la secuencia de operación está directamente ligada a las capacidades de estas. En este caso, se utiliza la herramienta proporcionada por "*FerRobotics"*, tal como muestra la imagen adjunta y el gemelo digital que encontraremos en la célula correspondiente. Esta herramienta está compuesta por dos útiles principales: la lijadora y la herramienta de limpieza, las cuales describiremos en detalle.

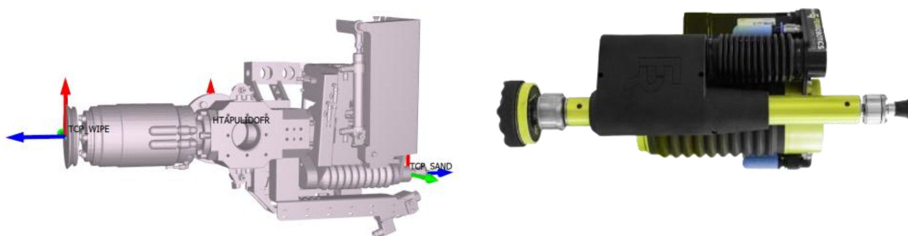


Ilustración 9: Herramienta "AAK 505/605" y su homólogo digital en la célula diseñada en RoboDK.

- **Lijadora:** Esta herramienta se utiliza para tratar los defectos más significativos que presentan un relieve físico. Su función es realizar un lijado de precisión sobre la imperfección para nivelar la superficie y prepararla para el posterior aporte de pintura.

- **Pulidora:** Esta herramienta consiste en un efector diseñado para sujetar un paño o aplicador especial, y su función es realizar la limpieza y preparación de la superficie, eliminando polvo, grasa u otros contaminantes para asegurar que las siguientes operaciones se realicen sobre un área limpia.

Una vez descrito el conjunto final, se presenta la secuencia completa de reparación de defectos diseñada para la célula. Para el presente caso de estudio, el alcance se ha focalizado en las operaciones iniciales del proceso, las cuales aparecen destacadas en azul oscuro en el diagrama de flujo operativo mostrado a continuación.

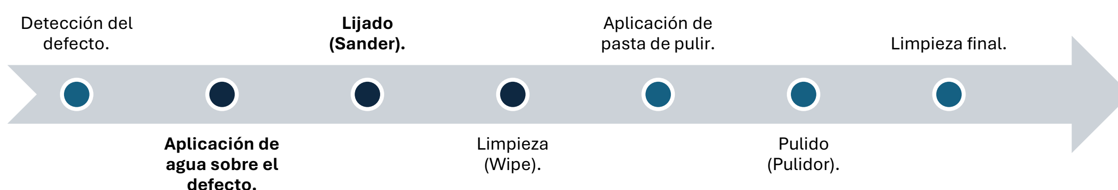


Diagrama 2: Desglose de las fases de la trayectoria de reparación.

Cabe destacar que el proceso de reparación integral conlleva múltiples fases, nuestra célula está dimensionada inicialmente para asumir las operaciones de lijado y limpieza, identificadas en azul oscuro en el esquema anterior. Sin embargo, es importante matizar el alcance de la programación: aunque el hardware propuesto es capaz de realizar ambas tareas, las trayectorias y la lógica de control desarrolladas en este proyecto están optimizadas específicamente para la primera operación, **la aplicación de agua sobre el defecto y el lijado**. Por tanto, la simulación presentada validará la capacidad del robot para ejecutar esta tarea principal, dejando la limpieza y fases posteriores (como el pulido, que requeriría un segundo robot) fuera del alcance de esta simulación específica.

Ilustración 10: Esquema del proceso y acotación de las fases automatizadas en este estudio.

Es importante destacar que las herramientas empleadas se integran en el modelo de simulación a través de un **Punto Central de la Herramienta (TCP)** específico. Este Tool Center Point define la posición funcional exacta que ocupa la lija y la pulidora permitiendo al robot , mostradas en la "*Ilustración 9: Herramienta "AAK 505/605" y su homólogo digital en la célula diseñada en RoboDK*. La definición precisa de estos TCP, es fundamental para garantizar la alineación precisa de la herramienta con el defecto a reparar.

Capítulo 4

5. Montaje de la célula.

5.1. Sistemas de referencia

Como primer paso en el desarrollo de la célula digital, se procede al montaje estructural de los elementos principales que la componen. Para ello, se han definido dos sistemas de coordenadas globales denominados "***Frame P64*" y "*Linear_Axis_Base*"**, que actúa como referencias principales para la ubicación espacial de todos los sólidos implicados en el proceso.

I. Sistema de referencia *"Frame P64".*

El sistema "*Frame P64"* se ha definido como referencia principal de la célula, dado que coincide con el sistema de referencia intrínseco del vehículo. De este modo, todas las operaciones robóticas quedan referenciadas espacialmente respecto a la posición del chasis, estableciendo el vínculo entre la célula previa de inspección y reparación.

De la misma manera, esta decisión responde a un criterio funcional debido a que, en entorno real de trabajo, uno de los parámetros más importantes es la forma en que la carrocería accede a la célula, ya que en los distintos modelos se presentan variaciones en su posición y orientación inicial.

Por este motivo, la elección de esta referencia como base de posicionamiento permite absorber dichas variaciones sin comprometer la lógica de trabajo del robot. Al establecer este sistema como origen geométrico, se garantiza una referenciación coherente entre el vehículo y el entorno de trabajo, lo que facilita la programación de trayectorias, la alineación de herramientas y la adaptación a múltiples configuraciones de carrocería dentro de la misma célula.

II. Sistema de referencia Linear Axis Base:

Para caracterizar la zona robotizada de la célula, se ha definido un segundo sistema de referencia denominado ***"Linear_Axis_Base",*** el cual depende jerárquicamente de "Frame P64" y ha sido alineado con él en los ejes principales, de modo que ambos comparten la misma altura y orientación espacial, esta referencia ha sido desplazada **2105,523 mm** en dirección transversal respecto a "Frame P64",

siguiendo criterios de accesibilidad, seguridad y funcionalidad, conforme a los principios establecidos en las normas ISO 10218-2 e ISO 12100, y considerando el alcance operativo del robot *“Fanuc R-2000iC/165”.* Esta distancia permite que el robot opere con seguridad y precisión sobre toda la superficie del chasis sin interferencias ni limitaciones cinemáticas.

Adicionalmente, se ha definido una **traslación longitudinal de -882,621 mm en el eje X** para el sistema base del eje lineal.

Esta corrección geométrica tiene como objetivo alinear el origen cinemático del séptimo eje con la referencia del chasis, optimizando así el espacio de trabajo útil . Al referenciar y desplazar la base del eje lineal respecto al sistema global (Frame_P64), tal y como se detalla en la imagen adjunta, se garantiza que el recorrido del robot esté perfectamente sincronizado longitudinalmente con la carrocería, asegurando la accesibilidad a todas las zonas de intervención.

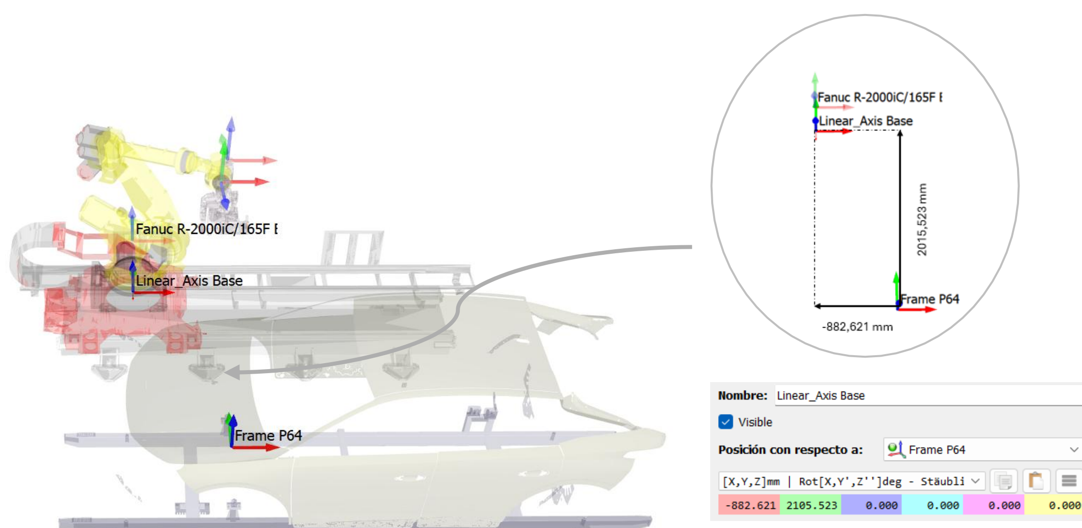


Ilustración 11: Orientación del sistema de referencia del robot (Linear_Axis_Base), respecto al sistema de referencia global de la célula (Frame P64).

5.2. Posicionamiento de los sólidos

Una vez definidos los sistemas de referencia, la colocación de los sólidos en la célula digital se realiza de forma inmediata.

En primer lugar, se procede al posicionamiento del chasis. Inicialmente, la carrocería estaba modelada como cinco sólidos independientes, correspondientes a distintas zonas del vehículo.

Para facilitar su tratamiento geométrico y optimizar la programación robótica, dichos elementos han sido unificados en un único sólido común, quedando el chasis alineado con el sistema de referencia global. En segundo lugar, se posiciona el carril lineal sobre el sistema *Linear_Axis_Base*, este carril como ya sabemos, servirá como guía para el desplazamiento longitudinal del robot, permitiendo ampliar su volumen de trabajo efectivo.

A continuación, se añade el carro de traslación, el cual se monta sobre el carril previamente posicionado. Este carro será el soporte móvil sobre el que se instalará el robot, y su alineación con el sistema de referencia garantiza que el desplazamiento se mantenga paralelo al eje longitudinal de la célula. Una vez montado el carro, se procede a posicionar el robot *"Fanuc R-2000iC/165"* sobre dicho soporte. El robot se alinea utilizando su propio sistema de referencia denominado con el nombre del modelo, asegurando que su base quede correctamente orientada respecto al carro y al carril de traslación.

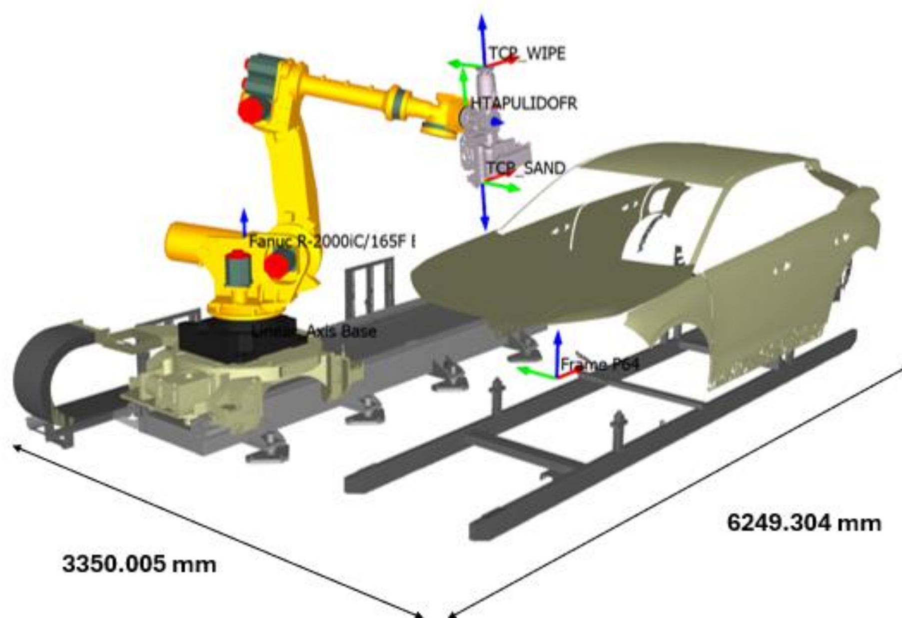


Ilustración 12: Disposición y configuración de la célula de reparación para el tratamiento de la carrocería, emulada en la herramienta de simulación.

Una vez definida la geometría, se ha llevado a cabo la caracterización funcional mediante la sincronización manual de ejes externos; esta operación integra el movimiento del carro dentro de la cadena cinemática del robot, constituyendo un séptimo eje que habilita la ejecución de trayectorias extendidas y continuas a lo largo de toda la longitud del chasis.

Es importante destacar que la configuración optada en este trabajo; un único robot sobre un eje lineal no es la única solución posible. La arquitectura de una célula de reparación depende en gran medida de las condiciones específicas de la línea, como el número de operaciones a realizar, el tiempo de ciclo requerido, o si varios robots deben trabajar en paralelo o en serie. En este proyecto se ha optado por la configuración de un solo robot con el objetivo de simplificar el modelo y facilitar el desarrollo y la validación de las ejecuciones posteriores.

6. Desarrollo de la simulación.

6.1. Planteamiento del problema y tratamiento de los datos de entrada.

I. Targets y parámetros de entrada:

Partiendo de la definición de la célula de trabajo presentada anteriormente, este apartado aborda la resolución del problema central: la generación de la **trayectoria de reparación**. La metodología se fundamenta en el procesamiento de un conjunto de datos de entrada provenientes de la célula de inspección previa (*detallada en el Capítulo 2, donde se encuentra el Contexto Tecnológico y Antecedentes: Proceso de inspección de defectos superficiales.*)

Dicha información se estructura como un listado de defectos, donde cada uno se define espacialmente mediante una serie de puntos sobre la carrocería del vehículo, denominados **targets** en el contexto de este trabajo. Es fundamental destacar que este listado incluye información detallada sobre la naturaleza del fallo, especificando el tipo de defecto (grano, suciedad, cráter, burbuja, etc.) y su magnitud o tamaño.

Esta caracterización tipológica es crítica, ya que vincula directamente el defecto con la estrategia de reparación a seguir; es decir, determina cómo debe moverse la herramienta una vez entra en contacto con la superficie (por ejemplo, mediante una trayectoria en espiral para defectos extensos o una acción localizada).

No obstante, para el alcance específico de este proyecto, se asume que todos los defectos detectados son de carácter leve. Bajo esta premisa, la estrategia de reparación se simplifica a una trayectoria puntual, el sistema se aproxima al *target* y ejecuta la reparación en una posición estática (como si se tratase de un punto único) y procede al retroceso, sin necesidad de realizar recorridos complejos sobre la superficie del defecto.

De manera general, un target está definido por un conjunto de seis valores: tres **coordenadas cartesianas (X, Y, Z)**, que determinan su posición exacta sobre la

carrocería, y tres **coordenadas de rotación ($\theta_x, \theta_y, \theta_z$)**, que especifican la orientación que debe adoptar la herramienta. El objetivo es procesar esta lista de targets para posteriormente, generar las trayectorias de reparación correspondientes.

$$Target_n = [x, y, z, \theta_x, \theta_y, \theta_z]$$

Expresión 1: Definición de un target con dimensión [1x6].

Para llevar a cabo el procesamiento de esta lista de targets, detallada íntegramente en el "*ANEXO 1- PARÁMETROS DE ENTRADA Y CÓDIGOS PARA MODELADO Y SIMULACIÓN DE LA CÉLULA.*", se ha desarrollado una solución algorítmica en Python. Es importante precisar que el sistema de inspección genera originalmente la ubicación de los defectos respecto al sistema de referencia del propio vehículo, por lo que para nosotros ese sistema de coordenadas es el sistema global de la célula , denominado *Frame_P64*.

Esta distinción es fundamental, dado que el *Frame_P64* constituye la única referencia geométrica común entre la célula de inspección y la de reparación. Al estandarizar los datos en este sistema global desde la entrada, se garantiza la coherencia espacial entre ambas etapas del proceso, eliminando la necesidad de realizar transformaciones de coordenadas adicionales en la fase de recepción de datos.

Posteriormente, se realiza un prefiltrado de los puntos para depurar la lista y optimizar el proceso. Para ello, se ha construido una caja delimitadora (o "*bounding box"*) de forma virtual, tomando como base las medidas reales del chasis para definir el volumen de trabajo útil. Esta caja se ha implementado en el script de Python para descartar automáticamente cualquier target que se sitúe fuera de ella.

Una vez filtrada la lista de defectos, se crean ***"targets de aproximación"*** asociados a cada defecto. La función de estos puntos es evitar que el robot se desplace directamente de un defecto a otro; en su lugar, el robot se aproxima primero a este punto de aproximación y desde allí, se dirige al defecto final, simulando un comportamiento más controlado.

De la misma manera, se han definido ***"puntos pivote"***. La función de estos puntos es doble: por un lado, actúan como un punto seguro al que el robot puede acudir en caso de detectarse una posible colisión, permitiéndole recalcular una trayectoria segura para ir de un target a otro y, por otro lado, se emplean como puntos de transición estratégicos entre diferentes superficies, asegurando que el robot no colisione con el chasis en ningún momento durante estos movimientos.

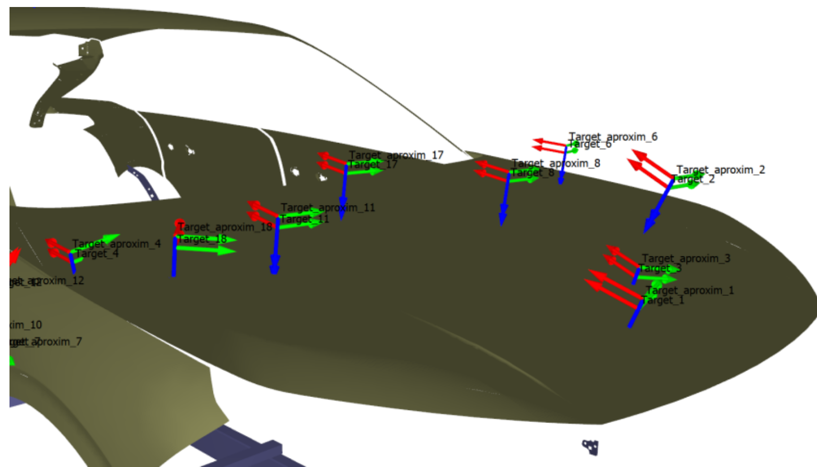


Ilustración 13: Creación de los targets (Target_n) y los targets de aproximación (Target_aproxim_n) para el listado de defectos otorgados.

II. <u>Desafío en la generación de los targets y alcanzabilidad cinemática:</u>

Una de las problemáticas identificadas durante la implementación del programa reside en la generación de los targets en el entorno de simulación. Se ha constatado que la creación de un target está intrínsecamente ligada a su **alcanzabilidad cinemática** desde la configuración articular actual del robot en el instante de su generación.

Si un target se define en una coordenada que el robot no puede alcanzar desde su estado presente, la interfaz de programación o el software de simulación no registra dicho target. En consecuencia, este punto no existe en el programa y no puede ser direccionado posteriormente en la trayectoria.

Para abordar esta dependencia, el procedimiento inicial intentaba generar la totalidad de los targets (capó, laterales y techo) partiendo de la posición de reposo *"Home"* del robot, donde todos sus ángulos articulares son cero.

Mediante la validación en simulación, se verificó que, desde esta configuración inicial, el espacio de trabajo del robot es insuficiente para alcanzar los puntos designados en el techo del chasis.

Es por este motivo que en la Ilustración 13, no se observan targets definidos en dicha superficie, ya que esta representa el estado del sistema previo al desplazamiento del séptimo eje.

Para revolver esta limitación el proceso se ha reestructurado de la siguiente manera:

1. El robot, partiendo de su posición *"Home",* genera y ejecuta las trayectorias correspondientes al capó y las zonas laterales, ya que estas son alcanzables desde su posición inicial.
2. Una vez completadas dichas inspecciones, el programa comanda un movimiento específico del séptimo eje ($\theta_7 = 1.700{,}00\ mm$) a una posición determinada, reubicando la base del robot.
3. Desde esta nueva configuración, que sitúa al robot en un espacio de trabajo óptimo para la parte superior, se procede a la creación *a posteriori* de los targets del techo.

Al ser generados desde esta nueva posición, los targets del techo resultan cinemáticamente alcanzables, permitiendo su correcta creación en el programa y su posterior inclusión en la secuencia de inspección.

III. Matrices de transformación homogénea.

Otro tema fundamental en el tratamiento de datos es la representación de los targets "pose", se suelen describir como un vector de tipo *[1x6]* como hemos visto anteriormente , que contiene la posición (x, y, z) y la orientación ($\theta_x, \theta_y, \theta_z$) de dicho target respecto a un sistema de referencia global.

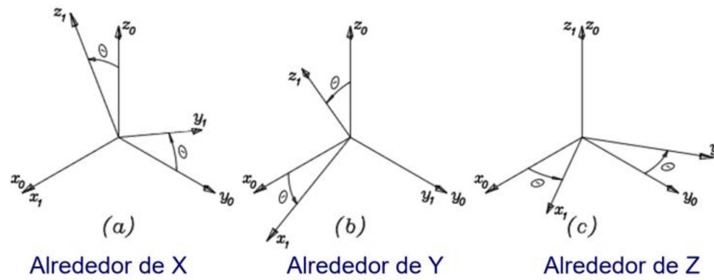
Aunque la representación de una pose como un vector *[1x6]*, es intuitiva a primera vista, presenta inconvenientes fundamentales en robótica. El principal problema reside en los ángulos de Euler, ya que las rotaciones no son conmutativas; es decir, el orden de los giros altera drásticamente el resultado final. Por este motivo, no podemos "encadenar" movimientos secuenciales mediante una simple operación de suma, ya que este cálculo no tendría un significado físico correcto. Si disponemos de un target y queremos cambiarlo de sistema de referencia no podremos ya que no podremos hacer inversas de matrices *[1x6].*

$$Pose_n = [T_{11}, T_{12}, T_{13}, \theta_x, \theta_y, \theta_z]$$

Expresión 2: Definición de Target.

Por estas razones, acudimos a las **matrices de transformación homogénea**, definida de la siguiente manera:

1. Para calcular la matriz de rotación homogénea, se obtiene primeramente cada matriz de giro definidas de la siguiente manera:



$$R_{x,\theta} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & cos(\theta_x) & -sen(\theta_x) \\ 0 & sen(\theta_x) & cos(\theta_x) \end{pmatrix}; \quad R_{y,\theta} = \begin{pmatrix} cos(\theta_y) & 0 & sen(\theta_y) \\ 0 & 1 & 0 \\ -sen(\theta_y) & 0 & cos(\theta_y) \end{pmatrix};$$

$$R_{z,\theta} = \begin{pmatrix} cos(\theta_z) & -sen(\theta_z) & 0 \\ sen(\theta_z) & cos(\theta_z) & 0 \\ 0 & 0 & 1 \end{pmatrix};$$

Expresión 3: Matrices de rotación para cada eje.

2. Una vez definidas las matrices de rotación se calcula la matriz de rotación total y se acopla a la de traslación con las siguientes expresiones:

Matriz Homogénea de rotación:

$$R = R_{x,\theta}.R_{y,\theta}.R_{z,\theta} = \begin{pmatrix} R_{11} & R_{12} & R_{13} \\ R_{21} & R_{22} & R_{23} \\ R_{31} & R_{32} & R_{33} \end{pmatrix};$$

Matriz Homogénea de traslación:

$$T = \begin{pmatrix} T_{11} \\ T_{12} \\ T_{13} \end{pmatrix};$$

$$Matriz\ Homogénea\ de\ transformación = \begin{pmatrix} R_{11} & R_{12} & R_{13} & T_{11} \\ R_{21} & R_{22} & R_{23} & T_{12} \\ R_{31} & R_{32} & R_{33} & T_{13} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Expresión 4: Matrices de rotación y traslación homogénea y ensamblaje de la matriz global de transformación homogénea.

Es importante señalar que la última fila de la matriz de transformación homogénea, representada se utiliza como el factor de escalado.

Esta representación es la herramienta estándar, ya que soluciona ambos problemas. Primero, permite la concatenación de movimientos mediante la multiplicación de matrices. La operación $Matriz_Tot = Mat_A \ x \ Mat_B$ sí tiene en cuenta el orden (A x B es diferente de B x A) y modela perfectamente la física real y segundo, la matriz homogénea permite cambiar de referencia gracias a la operación inversa.

Para gestionar esta conversión de formatos, la biblioteca de la *"API de RoboDK"* proporciona un módulo específico de utilidades matemáticas. La función concreta que realiza esta operación es "*robomath.xyzrpw_2_pose(vector pose) "*. Esta utilidad recibe como argumento el vector de *[1x6]* y se encarga de computar internamente el bloque de rotación y de ensamblar la matriz de transformación homogénea completa. La implementación de esta función, junto con un ejemplo de verificación de su correcto cálculo, se puede observar a continuación estableciéndose así el primer paso del proyecto para alcanzar el objetivo final del mismo.

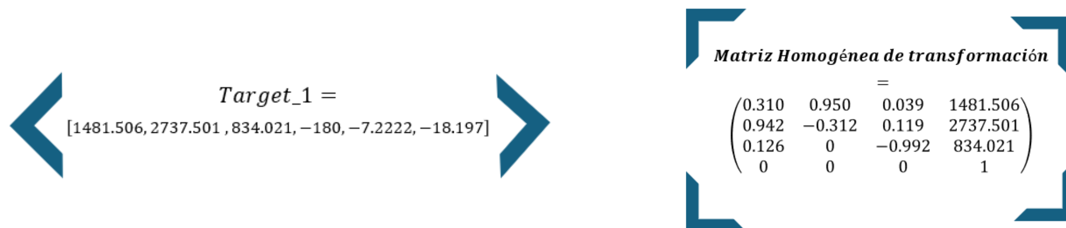


Ilustración 14: Ejemplificación de la función robomath.xyzrpw_2_pose(vector pose), con parámetro de entrada Target_1 y salida la matriz de trasformación homogénea a dicho target.

IV. Cinemática Inversa

Asimismo, es crucial subrayar que para garantizar la funcionalidad operativa de un sistema robótico, no basta con la mera definición espacial de una serie de puntos o trayectorias cartesianas. Resulta fundamental resolver la configuración interna que el manipulador debe adoptar para alcanzar dichas posiciones físicas.

En este contexto, cobra especial relevancia el concepto de **cinemática inversa**, entendido como el proceso matemático que permite determinar los valores de las coordenadas articulares (los ángulos de rotación específicos de cada eje: θ_1, $\theta_2, \theta_{3,}\theta_{n,}$) necesarios para que el efector final (TCP) logre una pose deseada en el espacio.

Aunque en el marco de este proyecto el software de simulación RoboDK asume la carga computacional de resolver estos algoritmos de manera automática, es imperativo destacar la cinemática inversa como un pilar conceptual clave, ya que constituye el motor matemático subyacente que posibilita la simulación y valida la viabilidad física de los movimientos.

6.2. Metodología de Generación de Trayectorias y Validación por Simulación.

Una vez situados los defectos en la carrocería, se procede a la fase de diseño y simulación completa de las trayectorias de inspección. El objetivo fundamental de este apartado es mediante un código desarrollado en Python ,que genera una trayectoria capaz de recorrer todos los defectos detectados en la carrocería, asegurando un movimiento continuo y libre de colisiones.

El código fuente íntegro que ha permitido esta simulación se adjunta en el Anexo 2. No obstante, algunas de las características principales del mismo deben ser expuestas en este capítulo, ya que constituyen la solución directa a ciertos problemas y desafíos que se han encontrado durante la implementación. A su vez, en el desarrollo de este apartado, se concluirá con la presentación de la trayectoria completa y los resultados de la simulación finalizada.

I. Estudio de alcanzabilidad y Detección de Colisiones entre Puntos.

El primer desafío a plantear en este apartado es verificar si la trayectoria entre dos puntos consecutivos es viable. Aunque a priori este reto parece sencillo, la propia implementación del simulador RoboDK introduce una complicación fundamental: El programa de por sí, detecta una colisión en el instante en que el robot entra en contacto físico con el chasis, pero al hacerlo, la célula de simulación queda paralizada y bloqueada. Este comportamiento, como se muestra en la Ilustración 15, impide que el programa continúe ejecutando el resto de los movimientos.

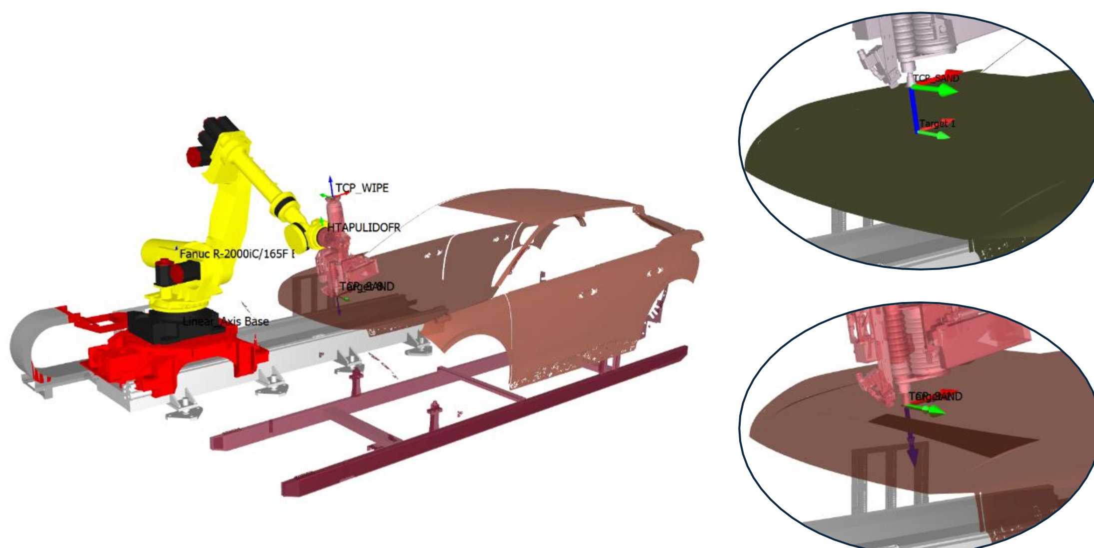


Ilustración 15: Detención completa de la célula al colisionar el Robot con el chasis.

Por lo tanto, el primer problema a resolver es cómo conseguir un método para detectar colisiones sin producir un colapso en la célula. Es decir, se debe encontrar una forma de saber a priori si el movimiento lineal de “*target A*” *un* “*target B*” generará una colisión, antes de comandar al robot que ejecute dicho movimiento.

Para abordar este problema, se recurre a la función ***“CollisionLine”,*** perteneciente a la propia biblioteca de la API de RoboDK. Esta función calcula un segmento de línea recta entre dos puntos (targets) y verifica si dicho segmento atraviesa algún objeto sólido de la célula, en cuyo caso devolvería un estado de colisión, sin implicar el movimiento del robot.

Durante la implementación de esta función se detectó una anomalía: en ocasiones, la comprobación no evaluaba la totalidad del segmento lineal, dejando ciertos tramos de la trayectoria sin verificar y, por ende, invalidando el resultado. Para solventar esta limitación, se optó por refinar el método: la línea principal se descompone programáticamente en una serie de subsegmentos contiguos más pequeños. Posteriormente, se itera sobre cada subsegmento aplicando la comprobación de colisión de forma individual. Si uno solo de esos tramos cruza un sólido, se determina que el movimiento completo no es viable y se determina la colisión.

En la ilustración posterior se puede observar el resultado de esta implementación. Se muestra una transición deliberadamente anómala entre el Target 1 y el Target 2; si bien la posición del Target 2 no sería realista (ya que, en un caso real, sería

descartada por la 'bounding box' de la zona), se ha forzado esta situación para ilustrar el correcto funcionamiento de la función CollisionLine.

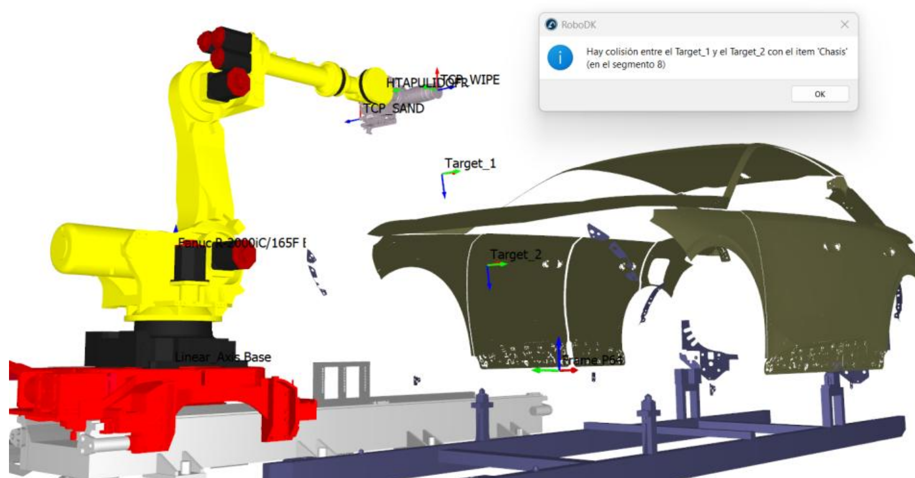


Ilustración 16: Resultado de la implementación programa en Python , que devuelve la existencia de colisón entre dos Targets, con el siguiente mensaje " Hay colision entre el target_n y el target_n+1 en el segmento i.

El análisis, tal y como se esperaba, determina que la transición entre ambos puntos provocará una colisión con el chasis. El método de descomposición de la línea identifica correctamente el impacto, reportando la **colisión en el segmento 5** de la trayectoria.

Del mismo modo, la metodología implementada permite verificar el caso opuesto: la **validación de trayectorias viables**. Como se observa en la Ilustración 17, se analiza la transición entre el Target_1y el Target_2. Tras aplicar la descomposición en subsegmentos y ejecutar la función "*CollisionLine"* en cada uno de ellos, el algoritmo comprueba que ninguno de los tramos interseca con el chasis ni con ningún otro sólido de la célula.

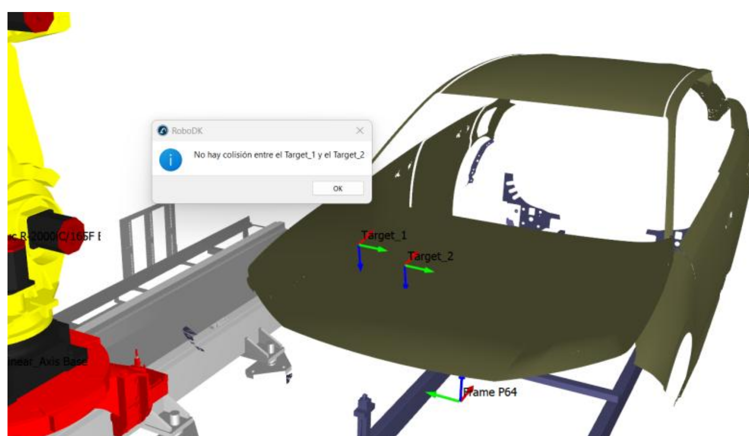


Ilustración 17: Implementación del Programa y Verificación Exitosa de Ausencia de Colisiones entre Targets.

El sistema reporta correctamente el mensaje *"No hay colisión"*, haciendo acopio del buen funcionamiento de la función implementada. Esto demuestra su fiabilidad no solo para detectar colisiones, sino también para confirmar que un movimiento es seguro antes de comandar su ejecución real al robot.

II. Ejecución de las trayectorias y movimientos.

Una vez verificado la trayectoria entre dos puntos con las funciones propuestas anteriormente, en este apartado procederemos a la ejecución de la simulación de la trayectoria completa. Para ello, se aplicará la estrategia de los targets de aproximación definidos anteriormente, con el fin de obtener una simulación más realista del proceso de inspección.

El robot no se moverá directamente de un defecto a otro. En su lugar, para cada punto, primero ejecutará un movimiento al target de aproximación asociado (a una distancia segura). A continuación, realizará un movimiento lineal de acercamiento al target del defecto en cuestión y, finalmente, un movimiento de retroceso al mismo target de aproximación antes de continuar con el siguiente punto.

Asimismo, para asegurar una simulación robusta y estructurada, la trayectoria completa se ha dividido para trabajar por secciones. El orden de ejecución se muestra en el diagrama a continuación:

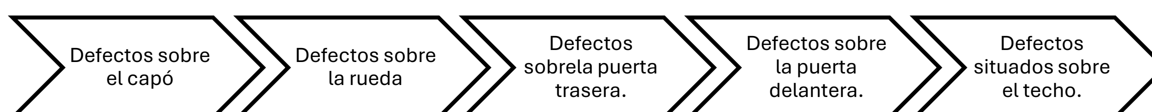


Ilustración 18: Orden de ejecución de las trayectorias por secciones del objeto de estudio.

III. Tratamiento defectos sobre el capó.

Para facilitar la comprensión del proceso de tratamiento de defectos, la operación se ha descompuesto y se muestra por secciones, como se ha mostrado en el diagrama anterior. Este enfoque permite observar con mayor detalle la trayectoria que sigue el robot durante la simulación. Tal y como se puede apreciar en las imágenes, dicha trayectoria se representa visualmente mediante las líneas de color amarillo que conectan los sucesivos puntos de control sobre la superficie de la pieza.

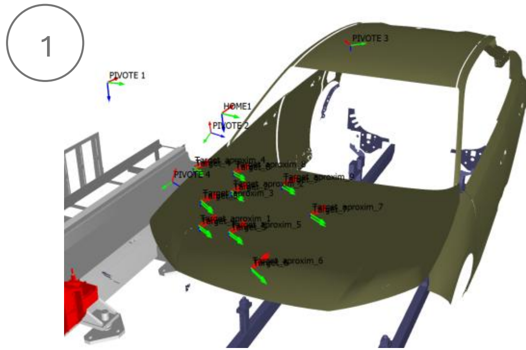


Ilustración 20:Fase inicial creación de los targets sobre el capó.

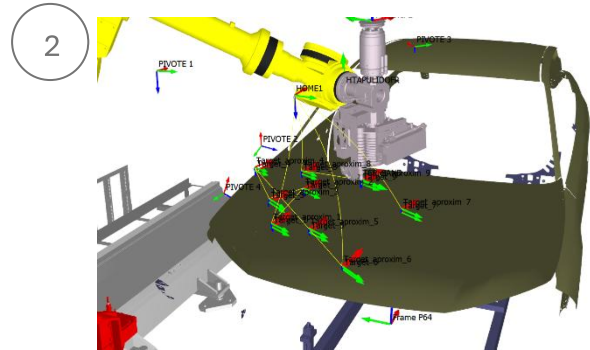


Ilustración 19: Fase de ejecución de las trayectorias.

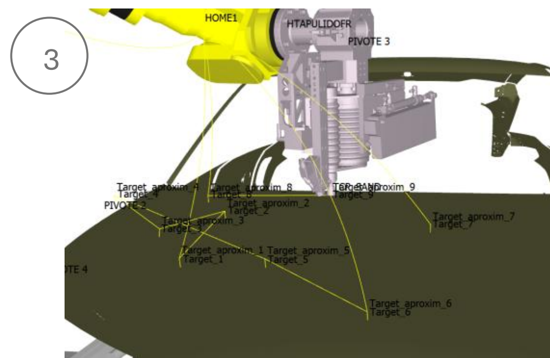
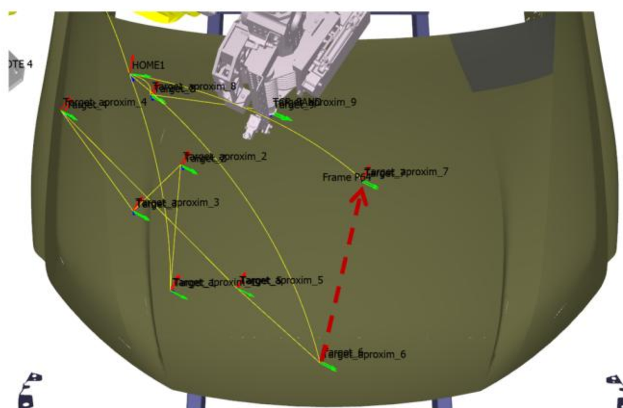


Ilustración 21:Reperación de los defectos sobre el capó completada.

IV. <u>Observaciones:</u>

Esta secuencia de imágenes mostradas a continuación valida la lógica de control programada. Se observa cómo el robot ejecuta la secuencia de trabajo (líneas amarillas) conectando Target_aprox_1 de manera secuencial hasta el Target_aprox_9. El análisis de la simulación confirma que esta ruta es viable, ya que el movimiento entre estos targets se mantiene a una distancia de seguridad de la geometría de la pieza y no genera ninguna interferencia o colisión.



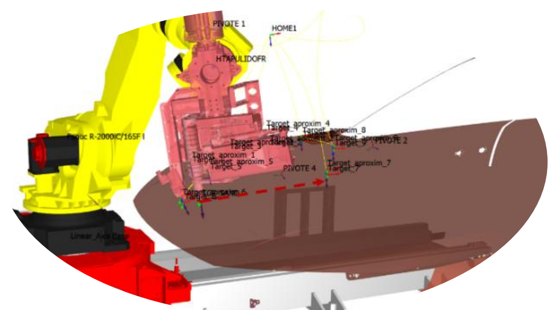


Ilustración 22: Trayectoria libre de colisiones (líneas amarillas) y evasión de las trayectorias que generan colisión (línea discontinua roja).

Por el contrario, el programa detecta que la trayectoria lineal directa desde Target_aprox_6 hasta Target_aprox_7 (línea roja discontinua) es inviable, puesto que generaría una colisión directa del robot contra el chasis.

Para solventar esta situación, se activa la maniobra de evasión programada: el robot realiza una retracción a la posición de seguridad "*HOME1",* un punto intermedio libre de obstáculos. Desde allí, puede recalcular la trayectoria y abordar el Target_aprox_7 sin riesgo. Esta secuencia demuestra la efectividad del algoritmo de prevención de colisiones, cumpliendo así un hito principal del proyecto.

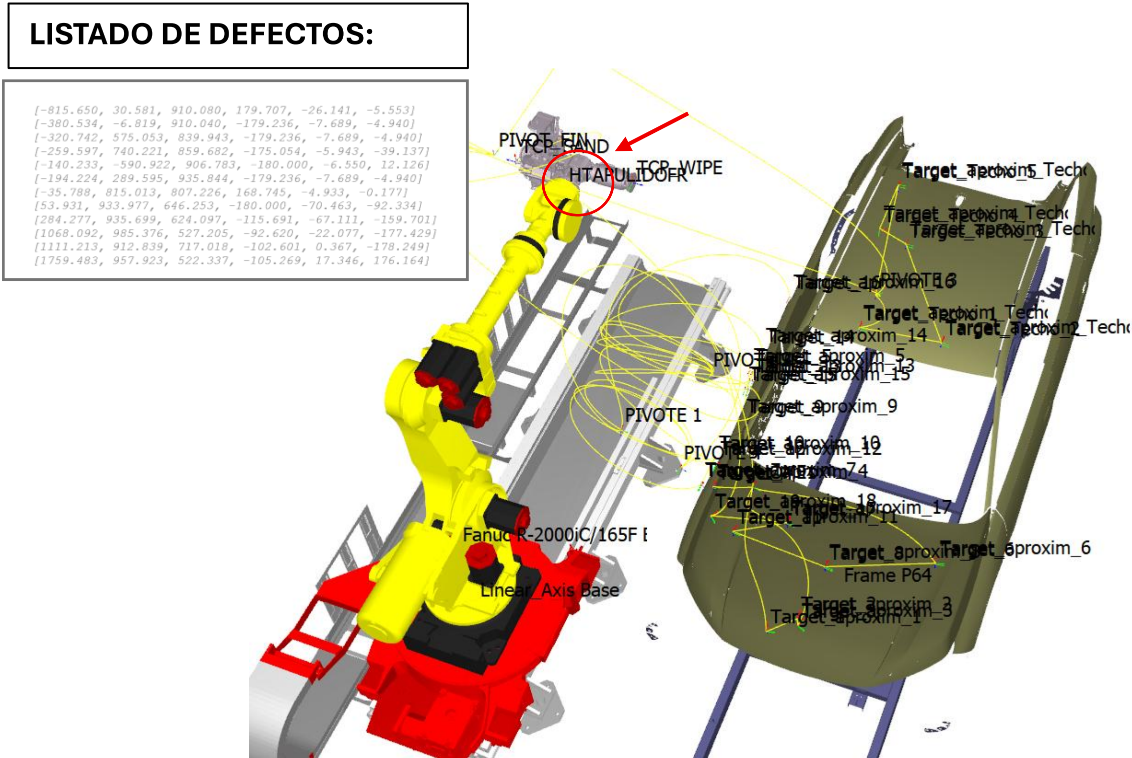
La metodología para la inspección y el tratamiento de defectos, tal como se ha ejemplificado para la estructura principal del chasis, se extrapola de manera análoga a todas las demás superficies del vehículo.

Esto incluye componentes específicos como las ruedas, las puertas y el techo, aplicando los mismos criterios y procedimientos de corrección tratado de manera conjunta en el apartado posterior.

V. Tratamiento de la célula en conjunto.

Finalmente, en este apartado se expone el resultado consolidado del proyecto. Esta solución engloba todas las metodologías analizadas a lo largo del trabajo, culminando en la definición de la trayectoria global que ejecuta el robot.

Como se muestra a continuación, esta trayectoria optimizada garantiza que el efector final recorra todos los puntos designados para la detección de defectos sobre la superficie del chasis, validando el enfoque completo de la simulación.



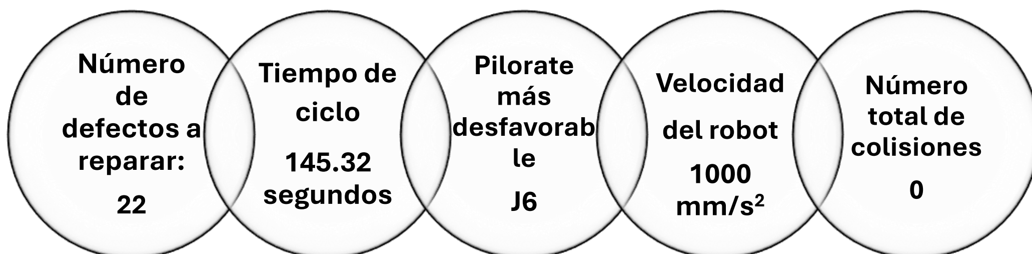


Ilustración 23: Escenario final de simulación: Ejecución de trayectorias de reparación completas y visualización de métricas de desempeño.

Una vez realizada la trayectoria global, debemos destacar que el proceso partió de un listado inicial de 54 targets potenciales, de los cuales se filtraron los 22 defectos que conforman la secuencia final de reparación. Para esta operación de 22 puntos, la simulación ha validado un tiempo de ciclo total de 145.32 segundos, un valor que se ajusta eficientemente a la cadencia de la línea de producción al optimizar los movimientos de desplazamiento entre defectos. Por supuesto, se ha cumplido el objetivo principal de viabilidad del trabajo, confirmando que el número de colisiones es cero. Adicionalmente, se ha implementado un script en Python para medir la suma total en grados de los pilotajes, de cada eje, permitiendo determinar de manera previsora cuál es la articulación más desfavorable. Este eje crítico señalado con un círculo rojo en la imagen anterior es de mayor recorrido

acumulado, necesitará un plan de mantenimiento futuro más exhaustivo. Este caso de aplicación en concreto presenta, por tanto, una solución validada, segura y eficiente, demostrando la viabilidad de la trayectoria obtenida (vista en la imagen anterior) para su implementación industrial.

VI. Conclusiones

El desarrollo de este capítulo ha permitido verificar la **alcanzabilidad cinemática** del sistema propuesto, demostrando que la configuración adoptada para el robot y el eje lineal cubre la totalidad del volumen de trabajo necesario sobre la carrocería. La correcta ubicación de los puntos pivote ha sido un factor determinante en este resultado, ya que ha facilitado la generación de un movimiento fluido y eficiente, logrando disminuir significativamente los *loops*[1] de reorientación del manipulador. Además, se ha comprobado que aquellos movimientos de reorientación inevitables se ejecutan estrictamente dentro del perímetro de seguridad de la célula, garantizando que ninguna parte del brazo robótico invada zonas de riesgo durante la operación.

En cuanto a la eficiencia del proceso, es importante destacar los resultados obtenidos en los tiempos de ciclo. Si bien el tiempo estándar de reparación manual suele estimarse en torno a los 40 segundos por defecto, la simulación arroja valores inferiores. Esta reducción se debe a que, en esta fase de validación de trayectorias, no se ha contabilizado el tiempo intrínseco de la operación física de reparación, sino el desplazamiento y posicionamiento del robot. No obstante, esta **trayectoria segura** y optimizada sienta las bases para un ciclo total altamente competitivo una vez se integren los tiempos de proceso reales.

En definitiva, la validación de estos parámetros confirma la **consolidación de un modelo cinemático viable**. La ausencia de colisiones, sumada a la optimización de los movimientos y el respeto a los límites de seguridad, demuestra que el gemelo digital desarrollado no es solo una representación teórica, sino una herramienta robusta preparada para su futura implementación física en un entorno industrial real.

Loop[1]: Es un movimiento de reorientación en el que el robot debe rotar ampliamente sus ejes para cambiar de configuración o evitar singularidades. Se manifiesta como una trayectoria curva y envolvente que desvía al brazo de la línea directa entre dos puntos.

:Capítulo 5

7. Estudio económico.

Una vez resuelto el núcleo de este trabajo, se pretende demostrar en este apartado que la propuesta no solo es viable desde el punto de vista ingenieril, sino también desde el punto de vista económico.

El objetivo es contrastar los costes asociados al proceso manual tradicional frente a los de la implantación de la célula robotizada propuesta en este trabajo, donde se pretende cuantificar el ahorro operativo y determinar la rentabilidad del proyecto, validando la propuesta como una inversión estratégica eficiente.

7.1. Definición del escenario de estudio.

Para el desarrollo del presente análisis, se plantea como punto de partida un escenario de estudio representativo de la industria actual; una empresa que ejecuta el proceso de reparación de defectos de pintura de manera íntegramente manual. Esta estructura operativa se sustenta en una plantilla de cuatro operarios contratados a jornada completa (40 horas semanales), organizados en dos turnos de trabajo, e incluye todo el equipamiento, vestuario y EPI necesarios para su desempeño.

Bajo esta premisa, el estudio aborda, desde una perspectiva financiera, el beneficio económico que supondría la sustitución de dicho proceso manual tradicional por la célula automatizada diseñada en el presente trabajo. El objetivo fundamental es cuantificar el impacto de esta transición tecnológica en la estructura de costes de la compañía y determinar su viabilidad mediante la aplicación de indicadores financieros clave: el Valor Actual Neto (VAN), la Tasa Interna de Retorno (TIR) y el plazo de recuperación (Payback).

En este contexto, es importante destacar que, si bien en la fase de simulación de trayectorias se obtuvieron los tiempos de ciclo específicos para la operación de pulido, el presente estudio económico adopta un enfoque integral y de la realidad industrial. Por consiguiente, para el cálculo de costes no se considerará únicamente el pulido, sino que se tendrá en cuenta el ciclo de trabajo completo, integrando tanto la fase de pulido como la de limpieza posterior de la zona, siguiendo el *“Diagrama 2: Desglose de las fases de la trayectoria de reparación.”*

A continuación, se presenta la tabla resumen con los costes anuales totales derivados de esta estructura operativa.

Condiciones de partidas homólogas para el proceso manual y el proceso automatizado	
Operación:	Pulido y limpieza
Numero de defectos a reparar:	22 defectos
Tiempo de la operación:	40 segundos por defecto
Número de horas efectivas al año:	3840 horas
Horizonte temporal:	10 años

Tabla 1: Parámetros iniciales operativos comunes a los procesos manual y automatizado.

7.2. Análisis económico de la situación actual (Proceso Manual).

Para establecer el escenario base del análisis, se define un entorno industrial estándar dedicado a la reparación manual de defectos de pintura en carrocerías de automóviles. La operativa actual se sustenta en una plantilla total de cuatro operarios contratados a jornada completa (40 horas semanales). La organización del trabajo se distribuye en dos turnos (mañana y tarde), con una asignación de dos trabajadores por turno para cubrir la demanda productiva.

El objetivo de este apartado es cuantificar el gasto anual integral que asume la empresa para mantener activa esta fase del proceso. Para determinar dicho coste, se han estimado y considerado las siguientes partidas de gasto fundamentales:

1. **Cargas Sociales:** Se contemplan los costes obligatorios de seguridad social a cargo de la empresa, desglosando los porcentajes de cotización por contingencias comunes, desempleo, formación profesional y accidentes de trabajo.
2. **Coste Operario a la Empresa:** Se parte del salario bruto anual por trabajador para calcular el coste total real que supone para la compañía mantener la plantilla activa, multiplicando el coste unitario por los cuatro operarios, divididos en dos turnos (mañana y tarde) necesarios para el proceso.
3. **Costes Indirectos y Vestuario:** Se incluyen los gastos recurrentes en dotación de uniformidad y Equipos de Protección Individual (EPIs) necesarios para garantizar la seguridad en el puesto (mascarillas P3, gafas de protección, mandiles de cuero y guantes), considerando su frecuencia de reposición y vida útil.

Cabe destacar que, para facilitar la lectura y fluidez de la memoria, el desglose numérico pormenorizado, las tablas salariales aplicadas completa de estos costes se encuentran detallados en el *" ANEXO 2- MEMORIA ECONÓMICA Y DESGLOSE DE COSTES."*

Costes del proceso manual de la reparación de los defectos	
Coste operarios	186.578,00€
Costes indirectos	555,53€
Herramientas	15.565,45€
Total costes del proceso manual	**202.698,98€**

Tabla 2: Sumatorio de los costes anuales totales pertenecientes al proceso manual.

7.3. Análisis económico de la propuesta (Célula robotizada)

Tras haber caracterizado económicamente la situación actual, se procede a analizar el escenario propuesto en este trabajo: la implantación de la célula robotizada. Para determinar con precisión los costes asociados a este nuevo escenario, se ha estructurado el análisis financiero en dos grandes bloques: la Inversión Inicial y los Costes operativos anuales.

A continuación, se describe el contenido de cada bloque, cuyo desglose cuantitativo se encuentra resumido en las tablas posteriores.

I. <u>Inversión inicial</u>

Esta categoría agrupa todos los desembolsos únicos necesarios en el "Año 0" para adquirir, instalar y arrancar el sistema. Las partidas contempladas son:

- **Costes de Adquisición** : Incluye la compra del brazo robótico (Fanuc R-2000iC/165F), las herramientas específicas de lijado y pulido y todos los elementos descritos en el apartado 4: *Descripción de la célula.*

Costes de adquisición	
Fanuc R-2000iC/165F	65.000€
Herramienta de SAND	35.000€
Herramienta WIPE	35.000€
7 eje lineal	15.000€
Carrito de traslación	7.000€
Costes de adquisición	**132.000€**

Tabla 3: Resumen de los costes de adquisición para la célula estudiada.

- **Costes de Instalación**: Engloba la mano de obra especializada necesaria para el montaje mecánico y eléctrico, incluyendo gestores de proyecto e ingenieros de robótica, teniendo en cuenta que la duración del proyecto de instalación y puesta en marcha suele tener una curación de 2 semanas, con el siguiente equipo técnico.

Costes de instalación	
1 Gestor del proyecto	4.000€
6 Técnicos mecánicos	2.800€
7 Técnicos electrónicos	2.800€
1 Ingeniero de robótica y puesta	2.800€
Costes de personal para la instalación.	**43.200€**

Tabla 4: Costes de personal para la instalación de la célula.

- **Costes de Adecuación**: Se contabiliza la inversión en seguridad industrial necesaria para integrar la célula en la planta, específicamente el vallado perimetral de seguridad (Serie 2000) , que incluye no solo el perimetral sino también la instalación.

Costes de adecuación local	
Vallado perimetral Serie 2000	25.000€
Costes de adecuación local	**25.000€**

Tabla 5: Costes de adecuación local.

- **Capacitación y puesta en marcha**: Incluye las licencias de software para el robot y la simulación, así como la asistencia técnica para el arranque de la reparación.

Costes de capacitación y puesta en marcha	
Software para el robot	5000€
Software para la simulación	5000€
Asistencia de arranque	4.500€
Costes de capacitación	**14.500€**

Tabla 6: Costes de capacitación y puesta en marcha.

II. Costes operativos.

Representan los gastos recurrentes necesarios para mantener la célula productiva a lo largo de su vida útil (estimada en 10 años). Las partidas consideradas son:

- **Energía Consumida:** Gasto eléctrico derivado del funcionamiento del robot y periféricos.
- **Mantenimiento**: Partidas para revisiones preventivas y reparaciones correctivas (variables según el año).
- **Consumibles:** Coste de los materiales fungibles (lijas y boinas) utilizados por el robot.
- **Seguro Técnico:** Póliza para cubrir posibles averías o incidentes del equipo.
- **Depreciación:** La amortización anual del valor del activo.
- **Salario de los Trabajadores:** Coste asociado a la supervisión de la línea y tareas auxiliares que requieren intervención humana .

Es importante señalar que la totalidad de los datos recogidos (ofertas de proveedores, fichas técnicas) y el desglose pormenorizado de los cálculos realizados se encuentran documentados exhaustivamente en el Anexo 2. A modo de síntesis, el sumatorio consolidado de los costes operativos anuales resultantes para el escenario robotizado se presenta a continuación:

Total costes operativos anuales			
AÑO 0	129.648,82€	**AÑO 6**	138.524,82€
AÑO 1	130.992,82€	**AÑO 7**	135.831,22€
AÑO 2	135.568,02€	**AÑO 8**	140.137,62€
AÑO 3	132.874,42€	**AÑO 9**	145.944,02€
AÑO 4	136,912,02€	**AÑO 10**	138.250,42€
AÑO 5	134.218,42€		

Tabla 7: Tabla resumen anual de los costes operativos asociados a la célula.

7.4. Evaluación de Rentabilidad e indicadores financieros.

Una vez definidos y cuantificados los costes asociados a los dos escenarios de estudio (proceso manual actual frente a la célula robotizada propuesta), este apartado tiene por objeto la evaluación financiera de la inversión. El objetivo es

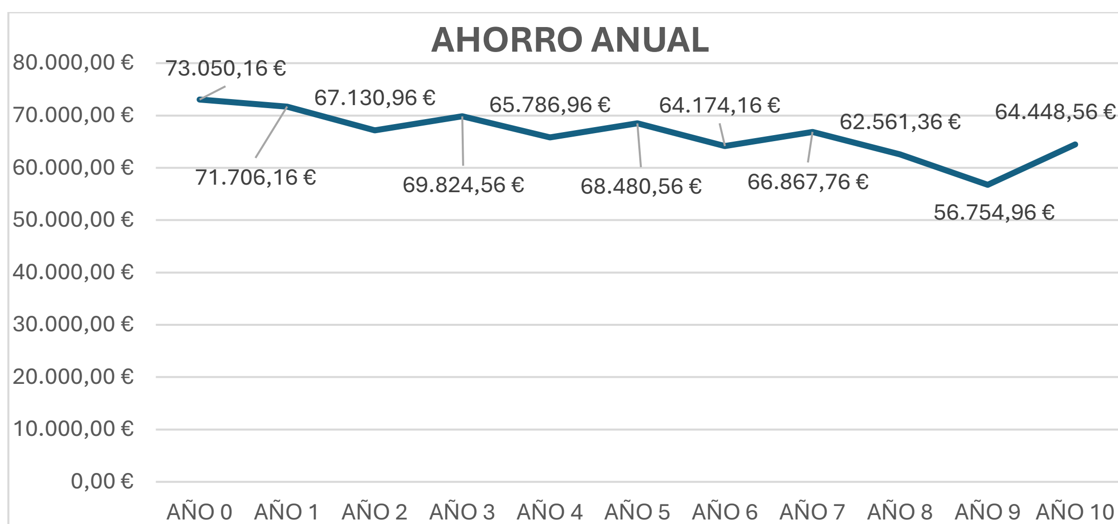
determinar si la sustitución del proceso manual por la automatización resulta rentable para la empresa en el horizonte temporal establecido.

El análisis se fundamenta en dos pilares: El Ahorro Anual Neto generado por la diferencia de costes operativos y el cálculo de los Indicadores Financieros dinámicos.

I. Ahorro anual.

La principal justificación económica del proyecto reside en la reducción drástica de los costes operativos. Al comparar los gastos del escenario manual con los del escenario automatizado, se obtiene un margen positivo anual o "ahorro neto". Este flujo de capital es el que permitirá amortizar la inversión inicial y generar beneficios a la empresa.

La evolución de este ahorro y su comportamiento a lo largo de los 10 años de estudio se representan visualmente en la Gráfica *1* que se muestra a continuación:



Gráfica 1: Evaluación de ahorro anual en un periodo de 10 años.

En cuanto al perfil temporal del ahorro, este no dibuja una trayectoria lineal, sino que exhibe una morfología en *'dientes de sierra'* típica de la gestión de activos industriales. Las variaciones observadas, concretamente los descensos en los años pares y la reducción más acusada del noveno año, son consecuencia directa del cronograma de mantenimiento preventivo y correctivo. Estos hitos, que implican la sustitución de componentes de desgaste, elevan puntualmente los costes operativos sin comprometer la rentabilidad global, Asimismo, es posible

identificar un menor ahorro en los últimos años provocado por el envejecimiento natural del equipo y la inflación de los consumibles.

Sin embargo, tal y como se aprecia en la gráfica, la implementación del sistema automático genera un beneficio neto positivo desde el primer año de operación, manteniéndose esta tendencia de manera ininterrumpida durante todo el horizonte temporal de diez años. Esta constancia en la generación de liquidez confirma la solidez del modelo de negocio, ya que en ningún año el ahorro desciende a valores negativos, situándose siempre por encima del umbral de los 56.000 € anuales incluso en los escenarios de mayor gasto operativo.

La magnitud de este rendimiento se hace evidente al observar el resultado global del proyecto. Tras finalizar la vida útil operativa estimada de la instalación, el **ahorro total acumulado asciende a la cifra de 730.786,21 €.** Este montante valida que la inversión inicial no solo se recupera con holgura, sino que la célula se convierte en un activo estratégico capaz de liberar más de medio millón de euros en recursos a lo largo de su vida útil , capital que la empresa puede redirigir a otras áreas productivas.

II. Indicadores Financieros.

Para validar la viabilidad del proyecto con rigor financiero, no basta con observar el ahorro bruto; es necesario aplicar indicadores que consideren el valor del dinero en el tiempo y la magnitud de la inversión inicial. Para este estudio se han calculado los tres indicadores estándar en ingeniería de proyectos, mostrados a continuación:

Ratios financieros de la implantación de la propuesta.	
VAN	269.754,51€
TIR	16,76 %
PAYBACK	3 años

- **Valor actual neto (VAN)**

El VAN es el indicador principal de rentabilidad, definido como la suma de los flujos de caja futuros actualizados al momento presente, descontados a la tasa de interés exigida al proyecto. Representa, en términos absolutos, la riqueza neta que la inversión genera por encima de la rentabilidad mínima requerida.

Para este proyecto, se ha obtenido un **VAN** de **269.754,51€**. Al ser este valor superior a cero, el resultado confirma matemáticamente que la inversión es

rentable. Esto significa que, tras cubrir todos los costes de inversión inicial y operación durante los 10 años, y tras satisfacer la tasa de rentabilidad exigida, el proyecto genera un excedente de riqueza para la empresa por lo que es una señal inequívoca de aceptación del proyecto desde el punto de vista financiero.

- **Tasa Interna de Retorno (TIR)**

Asimismo, para evaluar la eficiencia de la inversión se ha determinado la TIR, cuyo resultado del 16,76 % confirma la solidez financiera de la propuesta. Este índice no solo supera holgadamente el coste de oportunidad del capital, sino que establece un margen de seguridad, garantizando la viabilidad del proyecto incluso ante escenarios económicos menos favorables.

- **Periodo de retorno (Payback)**

Finalmente es necesario calcular el instante temporal donde vamos a recuperar la inversión inicial. El análisis determina un **Payback de 3 años**, esto supone que los recursos liberados durante los primeros cinco años financian la adquisición y puesta en marcha del equipo, mientras que, desde el comienzo del tercer año, el ahorro se consolida como ganancia neta. Teniendo en cuenta los tiempos de amortización característicos de los bienes de equipo industrial, este plazo de retorno confirma la viabilidad y sensatez de la inversión propuesta.

7.5. Conclusión del Estudio económico.

En definitiva, el análisis financiero ratifica la robustez de la transición hacia la célula robotizada. Los indicadores demuestran que la solución automatizada es económicamente superior al proceso manual, permitiendo no solo autofinanciar la inversión en un plazo razonable, sino también generar un valor neto significativo. Esta decisión estratégica optimiza la estructura de costes y libera recursos críticos, blindando la competitividad del proyecto a largo plazo.

Capítulo 6

8. Trabajo futuro.

El presente trabajo ha completado con éxito un recorrido integral por las fases críticas de la ingeniería de automatización. Se han alcanzado satisfactoriamente los hitos técnicos definidos al inicio: el modelado del entorno virtual en RoboDK, la implementación de un sistema de coordenadas dinámico para la localización de defectos, la programación algorítmica de trayectorias de aproximación y finalmente, la validación cinemática de la simulación para asegurar la ausencia de colisiones y singularidades.

No obstante, este proyecto no se ha limitado a la validación funcional. La inclusión de un estudio de viabilidad económica operativa que ha permitido cerrar el ciclo del proyecto, transformando un diseño técnico en una propuesta de negocio sólida.

La validación técnica y económica aquí expuesta no representa una conclusión definitiva, sino la consolidación de una base tecnológica robusta sobre la cual proyectar evoluciones más complejas. La arquitectura de la célula ha demostrado una alta escalabilidad, ofreciendo un escenario propicio para la innovación continua. En consecuencia, y capitalizando la infraestructura digital ya desarrollada, se plantean las siguientes líneas futuras de investigación:

8.1. Futuras líneas de investigación y desarrollo.

I. Compensación Dinámica del "Offset" de Entrada.

En la simulación actual, se asume que la carrocería llega en una posición ideal. En un entorno real, el sistema de transporte introduce pequeñas desviaciones posicionales. Una línea futura prioritaria es la integración de sistemas de visión artificial o palpado que calculen este desfase (*offset*) en tiempo real y recalibren el sistema de coordenadas del robot antes de iniciar el ciclo, garantizando que la herramienta actúe exactamente sobre el defecto.

II. Simulación del paquete (*Dresspack*)

Uno de los riesgos técnicos identificados es la rotura de cableado. Para mitigar esto en la fase de diseño, se propone implementar la simulación del comportamiento físico del *Dresspack* (tubos y cables externos). Esto permitiría prever tensiones

mecánicas, roces o enrollamientos excesivos del cableado durante los movimientos complejos de la muñeca del robot, optimizando la longitud y disposición de los suministros antes del montaje físico.

III. Integración de elementos periféricos de la célula

Tal y como se describió en la configuración del “4.2 Sistema robótico de reparación.“ La presente simulación ejemplifica el proceso utilizando una única unidad robótica para validar la calidad del pulido. Sin embargo, para completar el ciclo productivo bajo las exigencias de cadencia real, sería necesaria la integración de un segundo robot trabajando en paralelo. Una línea de investigación futura clave será la configuración de esta segunda unidad, replicando la metodología de este trabajo para gestionar operaciones simultáneas, sincronizar trayectorias y evitar interferencias entre ambos brazos, logrando así la cobertura del proceso completo.

IV. Ejecución mediante programas externos y optimización de patrones de trayectoria

En la configuración actual de la célula, la estrategia de reparación se basa en una aproximación posicional: el robot se desplaza a la coordenada del defecto, lo interpreta como un punto discreto y activa la herramienta. Sin embargo, la realidad industrial del tratamiento de superficies exige una actuación sobre la zona de carácter no puntual. La operación de pulido efectiva requiere que el robot ejecute patrones cinemáticos específicos alrededor del defecto, habitualmente configuraciones en espiral, dependiendo de la tipología del daño.

Por tanto, una línea de investigación futura prioritaria será la implementación de programas externos que no solo indiquen "dónde" está el defecto, sino "cómo" repararlo. El desafío técnico reside en generar estas espirales proyectando la sección de la curva sobre la geometría 3D de la carrocería, asegurando que el robot siga dicha trayectoria compleja manteniendo la orientación normal a la superficie en todo momento.

Capítulo 7

9. Conclusiones.

El desarrollo del presente Trabajo de Fin de Grado ha permitido cumplir satisfactoriamente con los objetivos planteados al inicio, cerrando el ciclo completo de diseño, simulación y viabilidad de una célula robotizada para la automatización del proceso de reparación de defectos en la carrocería de los automóviles. A modo de síntesis, se extraen las siguientes conclusiones principales:

En primer lugar, desde una perspectiva técnica, se ha logrado validar el diseño de la célula mediante un entorno de simulación 3D donde el valor diferencial del proyecto reside en la implementación de la generación de trayectorias mediante programación en Python. Gracias al uso de scripts programados, se han podido obtener tiempos de ciclo precisos y detectar de forma anticipada los puntos débiles de la instalación.

En concreto, el análisis ha permitido identificar y validar el comportamiento del robot en situaciones críticas, asegurando la alcanzabilidad, la ausencia de singularidades y la fluidez de los movimientos antes de cualquier implementación física.

En segundo lugar, este trabajo ha trascendido la barrera de la ingeniería virtual para adentrarse en la validación económica. No nos hemos limitado al diseño y la simulación técnica, sino que se ha completado el estudio con un análisis financiero exhaustivo, comparándolo con el proceso homólogo manual, se ha demostrado la robustez económica de la propuesta. Los indicadores obtenidos confirman la viabilidad del proyecto, justificando la inversión inicial mediante el ahorro significativo y la rentabilidad alcanzada.

Finalmente, este documento se constituye como el *"Paso 0"* estratégico y la hoja de ruta integral necesaria para viabilizar la transformación tecnológica de cualquier entorno industrial. Al abarcar todas las fases críticas ,desde la ingeniería de software y el diseño hasta el análisis de viabilidad, el trabajo ofrece una base sólida y verificada para cualquier empresa que desee instalar una célula de estas características. Se cierra así el ciclo del proyecto, entregando no solo una solución técnica funcional en simulación, sino una estrategia de despliegue a seguir para asegurar la transición industrial del proceso .

Capítulo 8

10. Bibliografía.

[1] RoboDK. Software de simulación de robots industriales y programación fuera de línea. https://robodk.com/es/, 2024. Accessed: 2025-11-25.

[2] RoboDK. RoboDK API for C++: Class RoboDK_API:RoboDK. https://robodk.com/doc/en/CppAPI/class_robo_d_k___a_p_i_1_1_robo_d_k.html#a3d1eae80846504319c14be64f91daec2, 2024. Accessed: 2025-11-25.

[3] RoboDK. Guía Básica de RoboDK. https://robodk.com/doc/es/Basic-Guide.html, 2024. Accessed: 2025-11-25.

[4] RoboDK. Programas de Robot: Instrucciones de programa. https://robodk.com/doc/es/Robot-Programs-Instrucciones-programa.html, 2024. Accessed: 2025-11-25.

[5] RoboDK. Consejos y Trucos (Tips & Tricks). https://robodk.com/doc/es/Tips-and-Tricks.html, 2024. Accessed: 2025-11-25.

[6] RoboDK. Reference Frames (Sistemas de Referencia). https://robodk.com/doc/en/Basic-Guide-Reference-Frames.html, 2024. Accessed: 2025-11-25.

[7] RoboDK. Getting Started: Create Targets. https://robodk.com/doc/en/Getting-Started-Create-Targets.html, 2024. Accessed: 2025-11-25.

[8] RoboDK. Interfaz: Configuración de objetos. https://robodk.com/doc/es/Interface-Configuraci%C3%B3n-objetos.html, 2024. Accessed: 2025-11-25.

[9] RoboDK. Simulación de programas de robot. https://robodk.com/doc/es/Robot-Programs-Simular-programa.html, 2024. Accessed: 2025-11-25.

[10] RoboDK. Programación Fuera de Línea con Python. https://robodk.com/doc/es/RoboDK-API-Programaci%C3%B3n-l%C3%ADnea-Python.html, 2024. Accessed: 2025-11-25.

[11] RoboDK. Documentación de la API de Python. https://robodk.com/doc/es/RoboDK-API-API-Python.html, 2024. Accessed: 2025-11-25.

[12] Aeterno Robotics. Fanuc R-2000iB/100P - Especificaciones técnicas. https://aeterno-robotics.com/robots/fanuc/r-2000-series/r-2000ib-100p/, 2024. Accessed: 2025-11-25.

[13] Ferrobotics. Active Angular Kit (AAK) - Automated sanding solution. https://www.ferrobotics.com/en/services/products/active-angular-kit/, 2024. Accessed: 2025-11-25.

[14] Vention. Heavy Duty 3.3 m 7th Axis for UR10e/UR16e. https://vention.io/designs/heavy-duty-33-m-7th-axis-for-ur10eur16e-44241, 2024. Accessed: 2025-11-25.

[15] Ferax. Folleto de Vallado Perimetral de Seguridad. https://www.ferax.es/images/folletos/Folleto-Vallado-Perimetral-FERAX.pdf, 2024. Accessed: 2025-11-25.

[16] Teamwork Protección. Gafa de seguridad panorámica Coverall de Bollé. https://teamworkproteccion.es/es/gafas-panoramicas/202-gafa-de-seguridad-panoramica-coverall-de-bolle-3660740000837.html, 2024. Accessed: 2025-11-25.

[17] A.T. Protección. Media Máscara Reutilizable 3M 6200. https://www.atproteccion.com/mascarillas-3m/4061-media-mascara-6200-3m.html, 2024. Accessed: 2025-11-25.

[18] Adypa. Guante antivibración A790. https://adypa.com/guantes-antivibracion/1571-guante-antivibracion-a790.html, 2024. Accessed: 2025-11-25.

[19] Prolaboral. Filtro de bayoneta para partículas 3M 2135 P3 R. https://www.prolaboral.com/es/1512-filtro-bayoneta-particulas-3m-2135.html, 2024. Accessed: 2025-11-25.

[20] O. Ramos. Fundamentos de Robótica: Cinemática Inversa. https://oramosp.epizy.com/teaching/201/fund-robotica/clases/5_Cinematica_Inversa.pdf?i=1, 2024. Accessed: 2025-11-25.

[21] Robotnik. Requisitos de seguridad para robots industriales: garantía de colaboración y eficiencia. https://robotnik.eu/es/requisitos-de-seguridad-para-robots-industriales-garantia-de-colaboracion-y-eficiencia/, 2024. Accessed: 2025-11-25.

[22] UPV/EHU (OpenCourseWare). T5 Cinemática - Revisión. https://ocw.ehu.eus/pluginfile.php/50445/mod_resource/content/8/T5%20CINEMATICA%20OCW_Revision.pdf, 2024. Accessed: 2025-11-25.

[23] Slideshare. Rotación y traslación por matrices homogéneas. https://es.slideshare.net/slideshow/rotacion-traslacion-por-matrices-homogeneas-2/46408431, 2015. Accessed: 2025-11-25.

Listado de ilustraciones

Ilustración 1:Célula de inspección de defectos "Eagle Eye" de J3D Vision. 11
Ilustración 2: Descripción de detección de defectos mediante la célula de inspección. 12
Ilustración 3: Posicionamiento y cota del skid en la herramienta de simulación. 14
Ilustración 4: Cota horizontal del modelo de estudio "Peugeot 3008". 15
Ilustración 5: Posicionamiento del skid y el chasis en la herramienta de simulación. 15
Ilustración 6: Cota longitudinal del séptimo eje y visualización en la herramienta de simulación. .. 16
Ilustración 7: Posicionamiento y anclaje del carro de traslación al 7 º eje. 16
Ilustración 8: Robot Fanuc R-2000iC/165F, representación en entorno de simulación frente a la unidad industrial real. 17
Ilustración 9: Herramienta "AAK 505/605" y su homólogo digital en la célula diseñada en RoboDK. 17
Ilustración 10: Esquema del proceso y acotación de las fases automatizadas en este estudio. 19
Ilustración 11: Orientación del sistema de referencia del robot (Linear_Axis_Base), respecto al sistema de referencia global de la célula (Frame P64). 21
Ilustración 12: Disposición y configuración de la célula de reparación para el tratamiento de la carrocería, emulada en la herramienta de simulación. 22
Ilustración 13: Creación de los targets(Target_n) y los targets de aproximación (Target_aproxim_n) para el listado de defectos otorgados. 25
Ilustración 14: Ejemplificación de la función , con parámetro de entrada Target_1 y salida la matriz de trasformación homogénea a dicho target. 28
Ilustración 15: Detención completa de la célula al colisionar el Robot con el chasis. 30
Ilustración 16: Resultado de la implementación programa en Python , que devuelve la existencia de colisón entre dos Targets, con el siguiente mensaje " Hay colision entre el target_n y el target_n+1 en el segmento i. 31
Ilustración 17: Implementación del Programa y Verificación Exitosa de Ausencia de Colisiones entre Targets. 31
Ilustración 18: Orden de ejecución de las trayectorias por secciones del objeto de estudio. 32
Ilustración 20:Fase inicial creación de los targets sobre el capó. 33
Ilustración 19: Fase de ejecución de las trayectorias. 33
Ilustración 21:Reperación de los defectos sobre el capó completada. 33
Ilustración 22: Trayectoria libre de colisiones (líneas amarillas) y evasión de las trayectorias que generan colisión (línea discontinua roja). 34
Ilustración 23: Escenario final de simulación: Ejecución de trayectorias de reparación completas y visualización de métricas de desempeño. 35

Universidad Zaragoza 1474

Listado de tablas

Tabla 1: Parámetros iniciales operativos comunes a los procesos manual y automatizado. 38
Tabla 2: Sumatorio de los costes anuales totales pertenecientes al proceso manual. 39
Tabla 3: Resumen de los costes de adquisición para la célula estudiada. 39
Tabla 4: Costes de personal para la instalación de la célula. 40
Tabla 5: Costes de adecuación local teniendo en cuenta que una empresa externa es la encargada de montar el perímetro. 40
Tabla 6: Costes de capacitación y puesta en marcha. 40
Tabla 7: Tabla resumen anual de los costes operativos asociados a la célula. 41

ANEXOS

ANEXO 1- PARÁMETROS DE ENTRADA Y CÓDIGOS PARA EL MODELADO Y LA SIMULACIÓN..54

ANEXO 2- MEMORIA ECONÓMICA Y DESGLOSE DE COSTES............................88

ANEXO 1- PARÁMETROS DE ENTRADA Y CÓDIGOS PARA MODELADO Y SIMULACIÓN DE LA CÉLULA.

1. Introducción

En el presente apartado se van a exponer y detallar todos los códigos necesarios que se han desarrollado para construir la solución completa.

tos scripts principales son:

1. Código final 1: *"Tratamiento de datos, creación de los targets y los targets de aproximación"*: El código responsable de definir y generar tanto los targets principales como los targets de aproximación necesarios para el guiado.

2. Código final 2: "Creación de trayectorias y simulación del movimiento"

   El algoritmo final que, basándose en los puntos creados por el script anterior, **genera la trayectoria** óptima.

De la misma manera, y con el fin de documentar todo el proceso de desarrollo, se presentarán los **scripts auxiliares** utilizados. Estas herramientas, aunque no forman parte del producto final, han sido fundamentales para **agilizar las pruebas** y la validación, incluyendo utilidades como los scripts de **eliminación de targets**, reseteo de entorno, etc.

2. Parámetros de entrada

I. Listado de defectos

[-800.917419, 190.705452, 759.694097, 172.623447, -21.981269, 14.111257]

[-754.423363, 24.628495, 940.270200, 179.707000, -26.141000, -5.553000]

[-710.082, 37.008, 770.443, -170.921, -12.540, -21.660]

[1588.531, -450.781, 188.468, -45.187, 88.315, -120.751]

[-35.787, 815.013, 807.226, 168.745, -4.933, -0.177]

[1068.091, 985.376, 527.205, -92.620, -22.077, -177.429]

[2811.415, 1055.275, 755.908, 12.834, -175.001, 8.529]

[-140.232, -590.921, 906.783, -180.000, -6.550, 12.126]

[53.931, 933.977, 646.253, -180.000, -70.463, -92.334]

[-380.534, -6.819, 910.040, -179.236, -7.689, -4.940]

[2005.183, 1151.018, 977.301, -150.598, 110.274, 33.918]

[776.169, 926.802, 365.662, -89.510, -14.831, -178.624]

[1255.870, -388.900, 1450.552, 98.712, -2.155, -88.093]

[284.276, 935.699, 624.097, -115.691, -67.111, -159.701]

[-326.285, 574.980, 880.810, -179.236, -7.689, -4.940]

[-985.116, 88.542, 550.173, 55.011, -133.628, 178.204]

[186.309, 882.840, 722.802, -160.158, -51.712, -111.915]

[295.107, 1221.391, 1488.005, -178.889, 44.110, -66.521]

[1444.621, 1445.897, 22.150, 77.294, 155.076, -101.378]

[986.710, 952.385, 536.099, -102.761, -69.535, -166.331]

[210.375, -810.150, 111.982, -20.447, -77.590, 4.331]

[1111.213, 912.838, 717.018, -102.601, 0.367, -178.249]

[-444.919, 1333.176, 450.772, 101.520, 29.817, 139.006]

[840.065, 926.738, 641.924, -112.878, -66.543, -156.868]

[2451.309, -210.887, 444.670, 31.995, 69.011, -155.280]

[3101.002, 1305.518, 988.225, 166.301, -99.447, 72.809]

[1759.482, 957.923, 522.337, -105.269, 17.346, 176.164]

[-194.224, 289.595, 935.844, -179.236, -7.689, -4.940]

[277.816, 1119.529, 215.100, -88.003, -118.995, 11.227]

[555.011, 1301.774, 955.336, 120.505, 7.881, -40.198]

[2999.557, -777.101, 1211.890, 8.765, 140.033, 160.773]

[-259.596, 740.220, 859.682, -175.054, -5.943, -39.137]

[160.119, -500.678, 1455.991, 144.478, -33.109, -133.007]

[2668.019, 133.477, 1449.201, -60.817, 100.910, 55.744]

[1333.882, -911.550, 677.029, 42.176, -166.700, -28.515]

[2201.173, -66.993, 201.445, 177.019, 122.903, 99.660]

[-1100.340, 1100.812, 100.256, 70.189, -50.761, 145.338]

[-655.002, -955.711, 100.992, -130.225, 160.311, -77.011

[1400.188, 1488.770, 900.510, 25.671, -80.419, 115.820]

[-990.517, 1450.220, 1100.753, -95.003, 15.998, -165.110]

[600.810, -100.150, 250.771, 15.201, 175.880, 22.405]

[2700.410, 1400.995, 400.100, -111.050, -44.771, 130.607]

[3050.600, 10.500, 700.315, 88.915, 99.011, -88.502]

[2105.523000, 882.621000, 0.000000, 0.000, 0.000, 90.000]

[450.778, 1450.880, 100.612, 135.700, -122.019, 45.188]

[10.112, 1490.300, 1480.910, -5.311, 66.820, -140.205]

[-1050.721, -100.815, 1477.001, 110.887, -177.530, 77.018]

[2300.915, 1450.118, 1490.550, -160.772, 88.009, -20.150]

[1700.337, -988.500, 10.188, 60.339, 22.710, 105.990]

[1085.194782, -228.687093, 1323.713855, -176.310, -13.843, -5.648]

[1741.022871, 197.831451, 1360.666990, 178.165, 1.370725, -3.616]

[2235.344061, 399.459955, 1347.687020, 174.177, 6.686567, -4.344]

[-10.995, -1011.880, 550.670, -145.008, -10.115, -177.300]

II. Puntos pivote utilizados:

4. Coordenadas de "HOME 1":

[-258.060882, 473.317677, 1319.723031, -179.236000, -7.689000, -7.689000]

- **Coordenadas de "PIVOTE 2"**

  [830.981774, 1190.677600, 683.292932, -112.877782, -66.542639, -156.868009]

- **Coordenadas de "PIVOTE 3"**

  [1291.229967, 252.768956, 1357.118273, 175.494770, -2.596637, 38.905182]

- **Coordenadas de " PIVOTE 4"**

  [176.755247, 1133.968761, 563.619285, -180.000000, -70.463000, -92.334000]

- **Coordenadas de " PIVOTE FIN"**

  [1069.868259, 2648.428749, 1941.929005, 103.384003, -88.499510, 162.837948]

3. Códigos finales:

I. Código final 1 para la **creación de targets y targets de aproximación**: Tratamiento de datos, creación de los targets y los targets de aproximación.

```
from robodk import robolink, robomath # Importar la API de RoboDK.
import math # Biblioteca matemA3@etica estA3@endar

# --- ConfiguraciA3in e InicializaciA3in ---

# busca una instancia de RoboDK que estA3© abierta en el ordenador y
establece un canal de comunicaciA3in con mi cA3©lula.
RDK = robolink. Robolink()

# DefiniciA3in de los identificadores  para los A3-tems de la estaciA3in.
nombre_robot = "Fanuc R-2000iC/165F"
nombre_referencia = "Frame P64" # Sistema de coordenadas base para los
targets.


# Ruta donde estA3@en el archivo de los defectos.
archivo_targets =
r"C:\Users\Martuki\Desktop\TFG\TFG_Marta\TextosImportantes\lista_targets.
txt"

# definiciA3in del offset en z-local e y-local(mm) para los puntos de
aproximaciA3in. Estos tendrA3@en la misma orientaciA3in
#que el defecto asociado para optimizar el cA3@elculo, debido a la
variabilidad geomA3©trica del chasis
# se definen dos offsets: para las superficies generales (capA3i,
puertas, techo) el target de aproximaciA3in estarA3@e a 20 mm y en el
#tramo de la rueda, por su complejidad geomA3©trica, se definirA3@e a 6
mm.
#a cotas superiores en esa zona, el target se ha validado como
cinemA3@eticamente inalcanzable.

VALOR_APROX_DEFAULT_MM = 20.0 #  (capA3i, puertas, techo)
VALOR_APROX_RUEDA_MM = 6.0     #   (ruedas)

# --- ValidaciA3in de la EstaciA3in de RoboDK ---

# Obtiene el A3-tem del robot desde la estaciA3in por su nombre
robot = RDK.Item(nombre_robot, robolink.ITEM_TYPE_ROBOT)
if not robot.Valid(): # Validar que el A3-tem existe y es accesible
    # Lanzar una excepciA3in si el robot no se encuentra
    raise Exception(f"El robot '{nombre_robot}' no se ha encontrado en la
estaciA3in.")

# Obtener el A3-tem del sistema de referencia (frame)
referencia = RDK.Item(nombre_referencia, robolink.ITEM_TYPE_FRAME)
if not referencia.Valid(): # Validar que el frame existe
    raise Exception(f"La referencia '{nombre_referencia}' no se ha
encontrado en la estaciA3in.")

# --- InicializaciA3in de Contadores y Estructuras de Datos ---
contador_creados = 0    # Acumulador para targets generados
contador_omitidos = 0   # Acumulador para targets descartados por el
filtro
contador_target = 1
targets_validos = []    # Lista para almacenar los targets que superen el
```

```
# --- 1.  Prefiltrado de Datos de Entrada ---

print(f"Iniciando lectura y filtrado del archivo: {archivo_targets}")
# Apertura del archivo en modo lectura ('r')
with open(archivo_targets, "r") as archivo:
    # Iterar sobre cada lA3-nea del archivo de texto
    for linea in archivo:
        # Normalizar la lA3-nea: eliminar espacios/saltos de lA3-nea y
caracteres ('[', ']')
        linea_limpia = linea.strip().replace("[", "").replace("]", "")

        try:
            # Cortar por las comas y crea una lista de string y
convierte cada string en un nA3emero
            datos = [float(x) for x in linea_limpia.split(",")]
        except ValueError as e:
            # Omitir la lA3-nea si el parsing falla (formato incorrecto,
lA3-nea vacA3-a)
            continue

        # Validar que la lA3-nea contiene 6 componentes (X,Y,Z,
Rx,Ry,Rz)
        if len(datos) != 6:
            continue

        # Asignar coordenadas cartesianas para facilitar la legibilidad
del filtro
        x, y, z = datos[0], datos[1], datos[2]
        # Flag booleano para gestionar la creaciA3in del target
        crear_target = False
        # String para almacenar la regla de filtrado aplicada
        tramo_aplicado = None

        # --- LA3gica de Bounding Box (Volumen de Trabajo A3љtil) ---

        # 1. Zona de ExclusiA3in (Techo)
        if (1040 < x < 2236) and (-230 < y < 400) and (z > 1300):
            print(f"Omitiendo (Zona Techo): x={x:.1f}, y={y:.1f},
z={z:.1f}")
            crear_target = False # Descartar explA3-citamente

        # 2. Zona CAPA3"
        elif -854 < x < 386:
            if y < 945 and (600 < z < 960): # Sub-zona superior
                crear_target = True
                tramo_aplicado = "CAPO (y < 945)"
            elif y >= 945 and (573 < z < 719): # Sub-zona lateral/rueda
                crear_target = True
                tramo_aplicado = "RUEDA (y >= 945)"

        # 3. Zona 1PUERTA
        elif 385 < x < 1050:
            if y < 940 and (300 < z < 1400):
                crear_target = True
                tramo_aplicado = "1PUERTA (y < 940)"
            elif y >= 940 and (300 < z < 850):
                crear_target = True
                tramo_aplicado = "1PUERTA (y >= 940)"
```

```
# 4. Zona 2PUERTA
        elif 1050 < x < 2500:
            if (-500 < y < 965) and (500 < z < 1400):
                crear_target = True
                tramo_aplicado = "2PUERTA (y < 965)"
            elif y >= 965 and (50 < z < 842.50):
                crear_target = True
                tramo_aplicado = "2PUERTA (y >= 965)"

        # 5. Zona FINAL
        elif x > 2500:
            if y < 965 and (890 < z < 920):
                crear_target = True
                tramo_aplicado = "FINAL (y < 965)"
            elif y >= 965 and (890 < z < 929.918):
                crear_target = True
                tramo_aplicado = "FINAL (y >=965)"

        # --- AcumulaciA3in de resultados del filtro ---

        if crear_target:
            # Si el flag es True, almacena los datos del target para
crearlos despuA3©s.
            targets_validos.append((datos, tramo_aplicado))
        else:
            # Si el flag es False, incrementar el contador de
omisiones
            contador_omitidos += 1
            # Logica de omisiA3in para el caso del techo.
            if not ((1040 < x < 2236) and (-230 < y < 400) and (z >
1300)):
                print(f"Omitiendo: x={x:.1f}, y={y:.1f}, z={z:.1f}
(Fuera de Bounding Box)")

# --- 2. CreaciA3in de Targets en la EstaciA3in ---

print(f"\nIniciando creaciA3in de {len(targets_validos)} targets y sus
aproximaciones...")

# Iterar sobre la lista de targets filtrados
for datos, tramo_aplicado in targets_validos:
    # Extraer coordenadas
    x, y, z = datos[0], datos[1], datos[2]

    # Generar nombres A3enicos para el par de targets
    nombre_target = f"Target_{contador_target}"
    nombre_aproxim = f"Target_aproxim_{contador_target}"

    # Convertir el vector [x,y,z,rx,ry,rz] a una matriz de
transformaciA3in homogA3©nea (Pose 4x4)
    #ya que ROBODK , espera recibir una matriz 4x4 no una lista de 6
nA3emeros
    pose_target = robomath.xyzrpw_2_pose(datos)

    # --- LA3igica de Offset para AproximaciA3in ---
    offset_mm = 0.0
    offset_info = "" # String para logging

    # Aplicar offset especA3-fico para la zona de la RUEDA
    if (-960 < x < 369) and (y > 860):
```

```
retroceder en Z local
        offset_info = f"Offset Rueda ({offset_mm}mm)"
    else:
        # Aplicamos el offset por defecto para el resto de las zonas
        offset_mm = -VALOR_APROX_DEFAULT_MM
        offset_info = f"Offset Default ({offset_mm}mm)"

    # Calcular la pose de aproximaciA3in:
    # Se post-multiplica la pose del target por una traslaciA3in en su
propio eje Z (local)
    pose_aproxim = pose_target * robomath.transl(0, 0, offset_mm)


    # Target Principal (Defecto)
    target = RDK.AddTarget(nombre_target, referencia) # Crear A3-tem
    target.setPose(pose_target)                        # Asignar su pose
    print(f"Creado: {nombre_target} (Regla: '{tramo_aplicado}')")
    contador_creados += 1 # Incrementar contador

    # Target de AproximaciA3in
    target_ap = RDK.AddTarget(nombre_aproxim, referencia) # Crear A3-tem
    target_ap.setPose(pose_aproxim)                        # Asignar la
pose con offset
    print(f" -> Creado Aprox: {nombre_aproxim} ({offset_info})")

    # Incrementar el A3-ndice numA3©rico para el siguiente par
    contador_target += 1


# --- 4. Reporte Final ---
print("\n--- Resumen de OperaciA3in ---")
print(f"Targets principales creados: {contador_creados}")
print(f"Targets de aproximaciA3in creados: {contador_creados}")
print(f"Targets omitidos (filtrados): {contador_omitidos}")

# Generar mensaje para la ventana emergente de RoboDK
mensaje_final = f"Proceso finalizado. {contador_creados} targets
principales y {contador_creados} aproximaciones generadas."
RDK.ShowMessage(mensaje_final, True) # Mostrar ventana emergente
print(f"\n{mensaje_final}") # Imprimir en consola
```

II. Código 2: Simulación de la trayectoria.

```
from robodk import robolink, robomath
import math
import os
import re

#
# CONFIGURACIA3"N DEL SISTEMA Y PARA3@3METROS DE ENTRADA

# Nombres de los items en el A3@erbol de RoboDK
ROBOT_NAME = "Fanuc R-2000iC/165F"
FRAME_NAME = "Frame P64"
HOME_NAME = "HOME1" # PosiciA3in de reposo segura
PIVOTE_1_NAME = "PIVOTE 1" # Puntos de paso intermedios (seguridad)
PIVOTE_2_NAME = "PIVOTE 2"
PIVOTE_3_NAME = "PIVOTE 3"
PIVOTE_4_NAME = "PIVOTE 4"
PIVOT_FIN_NAME = "PIVOT_FIN" # PosiciA3in final antes de recoger el
robot
IGNORE_NAMES = ['HTAPULIDOFR'] # Herramientas a ignorar en el cA3@elculo
de colisiones

# ParA3@emetros para la FASE 1 (InspecciA3in/Trabajo en CapA3i)
N_SAMPLES_CAPO = 50 # ResoluciA3in para discretizar la lA3-nea en
chequeo de colisiones
SHRINK_MM_CAPO = 0.5 # Margen de seguridad en los extremos del segmento

# ParA3@emetros para la FASE 2 (Rueda) - Registro de datos
OUTPUT_FILE_RUEDA =
r"C:\Users\Martuki\Desktop\TFG\TFG_Marta\TextosImportantes\REGISTRO_MOVI
MIENTO_RUEDA.txt"
Y_OFFSET_RUEDA = 24.85 # Offset para cA3@elculo matemA3@etico de
seguridad
Y_THRESHOLD_RUEDA = 904.0 # Umbral lA3-mite en Y para detectar posible
colisiA3in

# ParA3@emetros para la FASE 3 (Puerta)
OUTPUT_FILE_PUERTA =
r"C:\Users\Martuki\Desktop\TFG\TFG_Marta\TextosImportantes\REGISTRO_MOVI
MIENTO_PUERTA.txt"
Y_OFFSET_PUERTA = 24.85
Y_THRESHOLD_PUERTA = 907.0

# ParA3@emetros para la FASE 4 (GeneraciA3in dinA3@emica del Techo)
ARCHIVO_TARGETS_TECHO =
r"C:\Users\Martuki\Desktop\TFG\TFG_Marta\TextosImportantes\lista_targets
.txt"
VALOR_EJE_7_FINAL = 1700.0 # PosiciA3in del carril (track) para alcanzar
el techo
COORDS_PIVOTE_3 = [1291.229967, 252.768956, 1357.118273, 175.494770, -
2.596637, 38.905182]
Z_OFFSET_TECHO_APROX = 20.0 # Distancia de seguridad para los puntos de
aproximaciA3in (Z local)
```

```
#
=========================================================================
=
# FUNCIONES AUXILIARES


def sort_key(target_item):
    """
    Extrae el nA3ɛmero del nombre del target para ordenarlos
correctamente (1, 2, ... 10).
    Evita que 'Target 10' vaya antes que 'Target 2'.
    """
    match = re.search(r'\d+', target_item.Name())
    return int(match.group()) if match else 0

# Funciones matemA3@eticas para interpolaciA3in y comprobaciA3in
geomA3©trica
def to_xyz_list(pose): return [float(v) for v in pose.Pos()]
def interp(a, b, t): return [a[i] + (b[i] - a[i]) * t for i in
range(3)]

def shrink_segment(a, b, shrink_mm):
    """ Reduce ligeramente el segmento a comprobar para evitar falsos
positivos en los extremos """
    v = [b[i] - a[i] for i in range(3)]
    norm = math.sqrt(sum(vi*vi for vi in v))
    if norm == 0 or norm <= 2*shrink_mm: return a, b
    s = shrink_mm / norm
    return [a[i] + v[i]*s for i in range(3)], [b[i] - v[i]*s for i in
range(3)]

def parse_collision_result(res):
    """ Parsea el resultado que devuelve RoboDK dependiendo de la
versiA3in de la API """
    if isinstance(res, (list, tuple)) and len(res) >= 1:
        colliding_item = res[0]
        if hasattr(colliding_item, "Valid"):
            return colliding_item.Valid(), colliding_item
        return bool(colliding_item), (res[1] if len(res) > 1 else
None)
    return bool(res), None

def check_collision_geometrica(t1, t2):
    """
    Comprueba si la lA3-nea recta entre dos targets colisiona con el
entorno.
    Se discretiza la lA3-nea en 'N_SAMPLES_CAPO' puntos y se consulta
a RoboDK.
    """
    p1s, p2s = shrink_segment(to_xyz_list(t1.PoseAbs()),
to_xyz_list(t2.PoseAbs()), SHRINK_MM_CAPO)
    points = [interp(p1s, p2s, i/float(N_SAMPLES_CAPO)) for i in
range(N_SAMPLES_CAPO+1)]
    for i in range(N_SAMPLES_CAPO):
        res = RDK.Collision_Line(points[i], points[i+1])
        collision, coll_item = parse_collision_result(res)
        if collision:
            if coll_item and coll_item.Valid():
```

```
  # Si choca con una herramienta definida en IGNORE, no cuenta como
colisiA3in
                if any(ignore in coll_item.Name() for ignore in
IGNORE_NAMES):
                    continue
            print(f"    [ALERTA] Deteccion Geometrica: Colision
encontrada en trayectoria.")
            return True
    print("    [OK] Deteccion Geometrica: Trayectoria directa
libre.")
    return False

def get_aprox_target(target_item, aprox_map):
    """
    Busca en el diccionario si existe un target de aproximaciA3in
asociado al target actual.
    Relaciona 'Target_X' con 'Target_aproxim_X'.
    """
    if not target_item or not target_item.Valid():
        return None

    # Extraemos el ID numA3©rico del nombre
    match = re.search(r'(\d+)$', target_item.Name())
    if match:
        num_str = match.group(1)
        if num_str in aprox_map:
            return aprox_map[num_str]
    return None

#
====================================================================
# SCRIPT PRINCIPAL

RDK = robolink.Robolink()
print(">>> Iniciando script de generacion de trayectorias y
targets...")
RDK.setCollisionActive(robolink.COLLISION_ON)

# --- 1. Carga y validaciA3in de objetos de la estaciA3in ---
robot = RDK.Item(ROBOT_NAME)
home = RDK.Item(HOME_NAME)
pivote_1 = RDK.Item(PIVOTE_1_NAME)
pivote_2 = RDK.Item(PIVOTE_2_NAME)
# pivote_3 se gestionarA3@e dinA3@emicamente mA3@es adelante
pivote_4 = RDK.Item(PIVOTE_4_NAME)
pivot_fin = RDK.Item(PIVOT_FIN_NAME)
frame_ref = RDK.Item(FRAME_NAME)

# VerificaciA3in de integridad: si falta algo, detenemos el script
if not robot.Valid(): raise Exception(f"Error: No se encuentra el
robot '{ROBOT_NAME}'.")
if not home.Valid(): raise Exception(f"Error: No se encuentra
'{HOME_NAME}'.")
if not pivote_1.Valid(): raise Exception(f"Error: No se encuentra
'{PIVOTE_1_NAME}'.")
if not pivote_4.Valid(): raise Exception(f"Error: No se encuentra
'{PIVOTE_4_NAME}'.")
if not pivot_fin.Valid(): raise Exception(f"Error: No se encuentra
```

```
'{PIVOT_FIN_NAME}'.")
if not frame_ref.Valid(): raise Exception(f"Error: No se encuentra
'{FRAME_NAME}'.")

print(">>> Objetos cargados correctamente. Iniciando mapeo.")

all_targets = RDK.ItemList(robolink.ITEM_TYPE_TARGET)

# --- 2. CreaciA3in del Mapa de Aproximaciones ---
# Creamos un diccionario para acceso rA3@epido (O(1)) a los targets
de aproximaciA3in
aprox_map = {}
for target in all_targets:
    if target.Name().startswith("Target_aproxim_"):
        match = re.search(r'(\d+)$', target.Name())
        if match:
            num_str = match.group(1)
            aprox_map[num_str] = target

print(f">>> Mapeados {len(aprox_map)} targets de aproximacion
existentes.")

frame_pose_inv = robomath.invH(frame_ref.PoseAbs())
pivot_names = [HOME_NAME, PIVOTE_1_NAME, PIVOTE_2_NAME,
PIVOTE_4_NAME, PIVOT_FIN_NAME]


#
======================================================================
==
# FASE 1: PROCESADO ZONA CAPA3"
print("\n=============== FASE 1: ZONA CAPO ===============")

# Filtrado espacial de targets correspondientes al capA3i
capo_targets = []
for target in all_targets:
    if target.Name().startswith("Target_aproxim_"): continue #
Ignoramos aprox en la lista de trabajo
    if target.Name() in pivot_names: continue

    # Coordenadas relativas al frame de referencia
    x, y, z = (frame_pose_inv * target.PoseAbs()).Pos()
    # Caja de limitaciA3in (Bounding Box) aproximada del capA3i
    if (-960 < x < 386) and (y < 816) and (700 < z < 960):
        if len(robot.SolveIK(target.Pose()).list()) > 0: #
Verificamos alcanzabilidad
            capo_targets.append(target)

if not capo_targets:
    print("[INFO] No hay targets en la zona Capo.")
else:
    capo_targets.sort(key=sort_key)
    print(f">>> {len(capo_targets)} targets a procesar en Capo.")

    # Movimiento inicial a Home
    robot.MoveJ(home)

    # AproximaciA3in al primer punto
```

```
first_target = capo_targets[0]
    first_aprox = get_aprox_target(first_target, aprox_map)

    if first_aprox:
        print(f"- FASE 1 | Entrada segura via aproximacion:
{first_aprox.Name()}")
        robot.MoveJ(first_aprox) # MoveJ rA3@epido por aire
        robot.MoveL(first_target) # MoveL lento/preciso al punto de
trabajo
        robot.MoveL(first_aprox)  # Retirada lineal
    else:
        print(f"[AVISO] Sin aproximacion para {first_target.Name()}.
Movimiento directo.")
        robot.MoveJ(first_target)

    # Bucle de trabajo
    for i in range(len(capo_targets) - 1):
        current_target = capo_targets[i]
        next_target = capo_targets[i+1]
        next_aprox = get_aprox_target(next_target, aprox_map)

        # Caso sin target de aproximaciA3in (Legacy)
        if not next_aprox:
             print(f"[AVISO] {next_target.Name()} no tiene
aproximacion. Usando logica estandar.")
             if check_collision_geometrica(current_target,
next_target):
                 print(f"    [COLISION] Detectada. Evasion via
{home.Name()}.")
                 robot.MoveJ(home)
             else:
                 print(f"    [OK] Ruta directa viable.")
             robot.MoveL(next_target)
             continue

        # Caso CON target de aproximaciA3in (LA3gica Robusta)
        print(f"- FASE 1 | Procesando: {current_target.Name()} ->
{next_target.Name()}")

        # 1. ComprobaciA3in de colisiA3in en trayectoria directa
entre puntos de trabajo
        if check_collision_geometrica(current_target, next_target):
            print(f"    [COLISION] Ruta directa bloqueada. Evasion
via HOME.")
            robot.MoveJ(home) # EvasiA3in segura a Home
            print(f"    -> Recuperando posicion en
{next_aprox.Name()}")
            robot.MoveJ(next_aprox)
        else:
            print(f"    [OK] Ruta libre. Transicion via
aproximacion.")
            robot.MoveL(next_aprox) # Subimos al punto de seguridad
local

        # 2. Bajada al punto de trabajo y retorno
        robot.MoveL(next_target)
        robot.MoveL(next_aprox)

    # Salida de la zona
    print(f"- FASE 1 | Salida hacia {pivote_1.Name()}")
    if check_collision_geometrica(capo_targets[-1], pivote_1):
```

```
print(f"    [COLISION] Salida bloqueada. Evasion via HOME.")
        robot.MoveJ(home)

    robot.MoveJ(pivote_1)
    # Movimiento de transiciA3in a la siguiente zona
    robot.MoveJ(pivote_4)


#
======================================================================
# FASE 2: PROCESADO ZONA RUEDA
print("\n=============== FASE 2: ZONA RUEDA ===============")

rueda_targets = []
for target in all_targets:
    if target.Name().startswith("Target_aproxim_"): continue
    if target.Name() in pivot_names: continue
    x, y, z = (frame_pose_inv * target.PoseAbs()).Pos()
    # Filtro espacial para Rueda
    if -960 < x < 369 and y > 860:
        rueda_targets.append(target)

if not rueda_targets:
    print("[INFO] No hay targets en la zona Rueda.")
else:
    rueda_targets.sort(key=sort_key)
    print(f">>> {len(rueda_targets)} targets a procesar en Rueda.")

    # AproximaciA3in inicial
    first_target = rueda_targets[0]
    first_aprox = get_aprox_target(first_target, aprox_map)

    if first_aprox:
        print(f"- FASE 2 | Entrada segura via: {first_aprox.Name()}")
        robot.MoveJ(first_aprox)
        robot.MoveL(first_target)
        robot.MoveL(first_aprox)
    else:
        robot.MoveJ(first_target)

    # Registro de datos en fichero de texto
    os.makedirs(os.path.dirname(OUTPUT_FILE_RUEDA), exist_ok=True)
    with open(OUTPUT_FILE_RUEDA, "w", encoding="utf-8") as f_out:

        for i in range(len(rueda_targets) - 1):
            target_n, target_n1 = rueda_targets[i], rueda_targets[i+1]
            y_n, y_n1 = (frame_pose_inv *
target_n.PoseAbs()).Pos()[1], (frame_pose_inv *
target_n1.PoseAbs()).Pos()[1]
            y_to_check = min(y_n, y_n1) # Tomamos la coordenada Y
mA3@es crA3-tica

            next_aprox = get_aprox_target(target_n1, aprox_map)

            # Chequeo matemA3@etico de colisiA3in basado en
coordenadas Y (SimplificaciA3in eficiente)
            collision_risk = (y_to_check - Y_OFFSET_RUEDA) <
```

```
Y_THRESHOLD_RUEDA
            resultado_log = ""

            if not next_aprox:
                # Logica Legacy
                if collision_risk:
                    resultado_log = f"Colision matematica detectada.
Evasion via {pivote_4.Name()}."
                    print(f"    [COLISION] {resultado_log}")
                    robot.MoveJ(pivote_4) # EvasiÃ3in articulada
                else:
                    resultado_log = "Trayectoria limpia."
                    print(f"    [OK] {resultado_log}")

                robot.MoveL(target_n1)
                f_out.write(f"{target_n.Name()} ->
{target_n1.Name()}: {resultado_log}\n")
                continue

            # Logica Nueva con Aproximaciones
            print(f"- FASE 2 | Analizando: {target_n.Name()} ->
{target_n1.Name()}")

            if collision_risk:
                resultado_log = f"Colision matematica. Evasion via
{pivote_4.Name()}."
                print(f"    [COLISION] {resultado_log}")
                robot.MoveJ(pivote_4) # Salida a pivote seguro
                print(f"    -> Reingreso a {next_aprox.Name()}")
                robot.MoveJ(next_aprox)
            else:
                resultado_log = "Trayectoria limpia."
                print(f"    [OK] {resultado_log}")
                print(f"    -> Transicion via {next_aprox.Name()}")
                robot.MoveL(next_aprox)

            robot.MoveL(target_n1)
            robot.MoveL(next_aprox)

            f_out.write(f"{target_n.Name()} -> {target_n1.Name()}:
{resultado_log}\n")


#
======================================================================
# FASE 3: PROCESADO ZONA PUERTA

print("\n=============== FASE 3: ZONA PUERTA ===============")
print(f">>> Posicionando en punto de espera: {pivote_2.Name()}")
robot.MoveJ(pivote_2)

puerta_targets = []
for target in all_targets:
    if target.Name().startswith("Target_aproxim_"): continue
    if target.Name() in pivot_names: continue
    x, y, z = (frame_pose_inv * target.PoseAbs()).Pos()
    # Filtro espacial Puerta
    if 385 < x < 1759.483 and y > 910:
        puerta_targets.append(target)

if not puerta_targets:
    print("[INFO] No hay targets en zona Puerta.")
```

```
puerta_targets.append(target)

if not puerta_targets:
    print("[INFO] No hay targets en zona Puerta.")
else:
    puerta_targets.sort(key=sort_key)
    print(f">>> {len(puerta_targets)} targets a procesar en 
Puerta.")

    # Entrada inicial
    first_target = puerta_targets[0]
    first_aprox = get_aprox_target(first_target, aprox_map)

    if first_aprox:
        robot.MoveJ(first_aprox)
        robot.MoveL(first_target)
        robot.MoveL(first_aprox)
    else:
        robot.MoveJ(first_target)

    # Registro y ejecuciA3in
    os.makedirs(os.path.dirname(OUTPUT_FILE_PUERTA), exist_ok=True)
    with open(OUTPUT_FILE_PUERTA, "w", encoding="utf-8") as f_out:
        for i in range(len(puerta_targets) - 1):
            target_n, target_n1 = puerta_targets[i], 
puerta_targets[i+1]
            y_n = (frame_pose_inv * target_n.PoseAbs()).Pos()[1]

            next_aprox = get_aprox_target(target_n1, aprox_map)

            # Chequeo matemA3@etico umbral Y
            collision_risk = (y_n - Y_OFFSET_PUERTA) < 
Y_THRESHOLD_PUERTA
            resultado_log = ""

            if not next_aprox:
                if collision_risk:
                    resultado_log = f"Colision matematica. Evasion 
via {pivote_2.Name()}."
                    print(f"    [COLISION] {resultado_log}")
                    robot.MoveJ(pivote_2)
                else:
                    resultado_log = "Trayectoria limpia."
                    print(f"    [OK] {resultado_log}")
                robot.MoveL(target_n1)
                f_out.write(f"{target_n.Name()} -> 
{target_n1.Name()}: {resultado_log}\n")
                continue

            # Logica Nueva
            print(f"- FASE 3 | Analizando: {target_n.Name()} -> 
{target_n1.Name()}")

            if collision_risk:
                resultado_log = f"Colision matematica. Evasion via 
{pivote_2.Name()}."
                print(f"    [COLISION] {resultado_log}")
                robot.MoveJ(pivote_2)
                robot.MoveJ(next_aprox)
            else:
                resultado_log = "Trayectoria limpia."
                print(f"    [OK] {resultado_log}")
```

```
                robot.MoveL(next_aprox)

            robot.MoveL(target_n1)
            robot.MoveL(next_aprox)

            f_out.write(f"{target_n.Name()} -> {target_n1.Name()}:
{resultado_log}\n")

    print(f">>> Regresando a punto seguro: {pivote_2.Name()}")
    robot.MoveJ(pivote_2)


#
====================================================================
# TRANSICIA3"N AL FINAL DE LA3@cNEA
print("\n=============== TRANSICION ===============")
print(f">>> Moviendo a {pivot_fin.Name()} antes de iniciar Techo.")
robot.MoveJ(pivot_fin)

====================================================================
# FASE 4: PREPARACIA3"N DEL TECHO (Eje Externo + GeneraciA3in de
Targets)
print("\n=============== FASE 4: CONFIGURACION TECHO
===============")

# 1. Posicionamiento del 7A2e Eje (Rail/Track)
print(f">>> Ajustando eje externo a {VALOR_EJE_7_FINAL} mm para
alcanzar el techo...")
try:
    joints = robot.Joints().list()
    if len(joints) < 7:
        print("[AVISO] El robot no tiene eje externo configurado. Se
omite.")
    else:
        joints[6] = VALOR_EJE_7_FINAL
        robot.MoveJ(joints)
        print("    [OK] Eje externo posicionado.")
except Exception as e:
    print(f"[ERROR] Fallo al mover eje externo: {e}")
    RDK.ShowMessage(f"ERROR eje externo: {e}", True)

# 2. GestiA3in dinA3@emica del Pivote 3
print(f">>> Verificando existencia de {PIVOTE_3_NAME}...")
pose_pivote_3 = robomath.xyzrpw_2_pose(COORDS_PIVOTE_3)
pivote_3 = RDK.Item(PIVOTE_3_NAME, robolink.ITEM_TYPE_TARGET)

if not pivote_3.Valid():
    print(f"    [INFO] Creando target nuevo: {PIVOTE_3_NAME}")
    pivote_3 = RDK.AddTarget(PIVOTE_3_NAME, frame_ref)
else:
    print(f"    [INFO] Actualizando coordenadas de {PIVOTE_3_NAME}")
```

```
pivote_3.setPose(pose_pivote_3)

# 3. Lectura de fichero y generaciA3in de targets virtuales
contador_creados_techo = 0
contador_omitidos_techo = 0
contador_target_techo = 1
targets_validos_techo = []
lista_de_items_techo = []
lista_de_items_techo_aprox = []

try:
    print(f">>> Leyendo coordenadas desde: {ARCHIVO_TARGETS_TECHO}")
    with open(ARCHIVO_TARGETS_TECHO, "r") as archivo:
        for linea in archivo:
            # Limpieza de caracteres del formato lista de Python
'[x, y, ...]'
            linea_limpia = linea.strip().replace("[",
"").replace("]", "")
            try:
                datos = [float(x) for x in linea_limpia.split(",")]
            except ValueError: continue

            if len(datos) != 6: continue
            x, y, z = datos[0], datos[1], datos[2]

            # Filtro espacial para asegurar que pertenecen al Techo
            if (1040 < x < 2236) and (-230 < y < 400) and (z >
1300):
                targets_validos_techo.append((datos, "TECHO"))
            else:
                contador_omitidos_techo += 1

    # Ordenamos por coordenada X para optimizar barrido
    targets_validos_techo.sort(key=lambda t: t[0][0])

    print(f">>> Generando {len(targets_validos_techo)} targets en el
simulador...")

    for datos, tramo in targets_validos_techo:

        # Nomenclatura sistemA3@etica
        nombre_target = f"Target_Techo_{contador_target_techo}"
        nombre_target_aprox =
f"Target_aproxim_Techo_{contador_target_techo}"
        contador_target_techo += 1

        # A) Crear Target de Trabajo
        target = RDK.AddTarget(nombre_target, frame_ref)
        pose = robomath.xyzrpw_2_pose(datos)
        target.setPose(pose)
        lista_de_items_techo.append(target)

        # B) Crear Target de AproximaciA3in (Offset en Z local)
        datos_aprox = list(datos)
        datos_aprox[2] += Z_OFFSET_TECHO_APROX # Elevamos en Z

        target_aprox = RDK.AddTarget(nombre_target_aprox, frame_ref)
        pose_aprox = robomath.xyzrpw_2_pose(datos_aprox)
        target_aprox.setPose(pose_aprox)
        lista_de_items_techo_aprox.append(target_aprox)
```

```
if match_num:
            aprox_map[match_num.group(1)] = target_aprox

        contador_creados_techo += 1

    print("[RESUMEN] Generacion Techo:")
    print(f"    - Targets Trabajo: {contador_creados_techo}")
    print(f"    - Targets Aprox:
{len(lista_de_items_techo_aprox)}")
    print(f"    - Omitidos:        {contador_omitidos_techo}")

except Exception as e:
    print(f"[ERROR CRITICO] Fallo leyendo fichero targets: {e}")
    RDK.ShowMessage(f"Error fichero targets: {e}", True)


#
======================================================================
# FASE 5: EJECUCIA3“N TRAYECTORIA TECHO
print("\n=============== FASE 5: EJECUCION TECHO ===============")

if not lista_de_items_techo:
    print("[INFO] No hay targets de techo disponibles.")
else:
    print(f">>> Iniciando barrido de {len(lista_de_items_techo)}
puntos.")
    robot.MoveJ(pivote_3)

    # Entrada inicial
    first_techo_target = lista_de_items_techo[0]
    first_aprox = get_aprox_target(first_techo_target, aprox_map)

    if first_aprox:
        print(f"- FASE 5 | Aproximacion inicial:
{first_aprox.Name()}")
        robot.MoveJ(first_aprox)
        robot.MoveL(first_techo_target)
        robot.MoveL(first_aprox)
    else:
        robot.MoveJ(first_techo_target)

    # Barrido
    for i in range(len(lista_de_items_techo) - 1):
        current_target = lista_de_items_techo[i]
        next_target = lista_de_items_techo[i+1]
        next_aprox = get_aprox_target(next_target, aprox_map)

        if not next_aprox:
            # Fallback a metodo antiguo
            if check_collision_geometrica(current_target,
next_target):
                print(f"    [COLISION] Evasion via
{pivote_3.Name()}.")
                robot.MoveJ(pivote_3)
            else:
                print(f"    [OK] Directo.")
            robot.MoveL(next_target)
```

```
                continue

            # Metodo con aproximacion
            print(f"- FASE 5 | {current_target.Name()} ->
{next_target.Name()}")

            if check_collision_geometrica(current_target, next_target):
                print(f"    [COLISION] Evasion via {pivote_3.Name()}.")
                robot.MoveJ(pivote_3)
                robot.MoveJ(next_aprox)
            else:
                print(f"    [OK] Transicion via aproximacion.")
                robot.MoveL(next_aprox)

            robot.MoveL(next_target)
            robot.MoveL(next_aprox)

        print(f">>> Fin barrido. Retorno a {pivote_3.Name()}")
        last_techo_target = lista_de_items_techo[-1]

        # Comprobacion final antes de volver al pivote
        if check_collision_geometrica(last_techo_target, pivote_3):
            print(f"    [ALERTA] Riesgo de colision en retorno.
Precaucion.")

        robot.MoveL(pivote_3)

# ===================================================================
# FINAL DEL PROCESO
print("\n=============== FIN DE PROCESO ===============")
print(f">>> Robot a posicion de repliegue: {pivot_fin.Name()}")
robot.MoveJ(pivot_fin)

# Reset a posiciA3in cero (Home mecA3@enico)
print(">>> Enviando robot a CERO absoluto
(Mantenimiento/Calibracion)...")
try:
    num_axes = len(robot.Joints().list())
    joints_cero = [0.0] * num_axes
    robot.MoveJ(joints_cero)
except Exception as e:
    print(f"[ERROR] No se pudo ir a CERO: {e}")

print("\n>>> Script finalizado correctamente.")
RDK.ShowMessage("Ciclo completo finalizado.", True)
```

4. Códigos preliminares : Creación de las trayectorias

I. Trayectoria sobre el capó.

```
from robodk import robolink, robomath
import math
import re

RDK = robolink.Robolink()
print(">>> Script de movimiento interactivo iniciado correctamente")

# --- Reiniciar colisiones ---
RDK.setCollisionActive(robolink.COLLISION_OFF)
RDK.setCollisionActive(robolink.COLLISION_ON)
RDK.Update()

# ---------- CONFIGURACIA3"N ----------
ROBOT_NAME = "Fanuc R-2000iC/165F"  # <-- IMPORTANTE: Pon el nombre
exacto de tu robot
FRAME_NAME = "Frame P64"
HOME_NAME = "HOME1"
PIVOTE_NAME = "PIVOTE 1" # <-- AA3'ADIDO: Nombre del target final
N_SAMPLES = 50
SHRINK_MM = 0.5
IGNORE_NAMES = ['HTAPULIDOFR']

# ---------- Funciones ----------
def to_xyz_list(pose): return [float(v) for v in pose.Pos()]
def interp(a, b, t): return [a[i] + (b[i] - a[i]) * t for i in
range(3)]
def shrink_segment(a, b, shrink_mm):
    v = [b[i] - a[i] for i in range(3)]
    norm = math.sqrt(sum(vi*vi for vi in v))
    if norm == 0 or norm <= 2*shrink_mm: return a, b
    s = shrink_mm / norm
    return [a[i] + v[i]*s for i in range(3)], [b[i] - v[i]*s for i in
range(3)]
def parse_collision_result(res):
    if isinstance(res, (list, tuple)) and len(res) >= 1:
        # Si el resultado es una tupla/lista, el primer elemento indica
la colisiA3n
        # (puede ser un booleano o un objeto Item)
        colliding_item = res[0]
        if hasattr(colliding_item, "Valid"):
            return colliding_item.Valid(), colliding_item
        return bool(colliding_item), (res[1] if len(res) > 1 else None)
    # Si es un simple booleano
    return bool(res), None
def sort_key(target_item):
    match = re.search(r'\d+', target_item.Name())
    return int(match.group()) if match else 0
```

```
def check_collision_between(t1, t2):
    """FunciA3in que encapsula la lA3gica de detecciA3in de
colisiones entre dos targets."""
    p1s, p2s = shrink_segment(to_xyz_list(t1.PoseAbs()),
to_xyz_list(t2.PoseAbs()), SHRINK_MM)
    points = [interp(p1s, p2s, i/float(N_SAMPLES)) for i in
range(N_SAMPLES+1)]

    for i in range(N_SAMPLES):
        res = RDK.Collision_Line(points[i], points[i+1])
        collision, coll_item = parse_collision_result(res)

        if collision:
            if coll_item and coll_item.Valid():
                name = coll_item.Name()
                if any(ignore in name for ignore in IGNORE_NAMES):
                    continue # Es una colisiA3in ignorada, seguimos
comprobando
            print(f"    -> DetecciA3in: A2@eColisiA3in encontrada en
el segmento {i}!")
            return True # ColisiA3in real detectada

    print("    -> DetecciA3in: Trayectoria directa libre.")
    return False

# ---------- Cargar Objetos Principales ----------
robot = RDK.Item(ROBOT_NAME, robolink.ITEM_TYPE_ROBOT)
home_target = RDK.Item(HOME_NAME)
frame_ref = RDK.Item(FRAME_NAME, robolink.ITEM_TYPE_FRAME)
pivote_target = RDK.Item(PIVOTE_NAME) # <-- AA3'ADIDO: Cargar el
target pivote

if not robot.Valid(): raise Exception(f"No se encontrA3i el robot:
{ROBOT_NAME}")
if not home_target.Valid(): raise Exception(f"No se encontrA3i el
target de evasiA3in: {HOME_NAME}")
if not frame_ref.Valid(): raise Exception(f"No se encontrA3i el frame
de referencia: {FRAME_NAME}")
if not pivote_target.Valid(): raise Exception(f"No se encontrA3i el
target final: {PIVOTE_NAME}") # <-- AA3'ADIDO: VerificaciA3in

# ---------- Obtener, filtrar y ordenar targets ----------
all_targets_in_station = RDK.ItemList(robolink.ITEM_TYPE_TARGET)
frame_pose_inv = robomath.invH(frame_ref.PoseAbs())
filtered_targets = []

print("\n>>> Analizando y filtrando targets por coordenadas...")
for target in all_targets_in_station:
    # Excluimos el target pivote del proceso de filtrado y
ordenaciA3in principal
    if target.Name() == PIVOTE_NAME:
        continue

    relative_pose = frame_pose_inv * target.PoseAbs()
    x, y, z = relative_pose.Pos()
```

```
# Aplicamos la condiciA3in de filtrado
    if (-960 < x < 386) and (y < 816) and (700 < z < 960):
        # Comprobamos si el target es alcanzable antes de aA3±adirlo
        if len(robot.SolveIK(target.Pose()).list()) > 0:
            filtered_targets.append(target)
        else:
            print(f"    C2љ Cfë@f  Target '{target.Name()}' descartado
por ser inalcanzable.")

if not filtered_targets:
    raise Exception("NingA3єn target cumpliA3i con las condiciones de
filtrado o ninguno era alcanzable.")

filtered_targets.sort(key=sort_key)

print(f">>> {len(filtered_targets)} targets pasaron el filtro y han sido
ordenados:")
for t in filtered_targets:
    print(f"  - {t.Name()}")


# ---------- Ejecutar Trayectoria con EvasiA3in de Colisiones ----------
print("\n>>> INICIANDO EJECUCIA3"N DE TRAYECTORIA:")

# --- Movimiento inicial a HOME ---
print(f"1. Moviendo a la posiciA3in inicial de seguridad:
{home_target.Name()}")
robot.MoveJ(home_target)

# --- Movimiento al primer target de la secuencia ---
first_target = filtered_targets[0]
print(f"2. Moviendo al inicio de la secuencia: {first_target.Name()}")
robot.MoveJ(first_target)

# --- Bucle principal de movimiento ---
for idx in range(len(filtered_targets) - 1):
    current_target = filtered_targets[idx]
    next_target = filtered_targets[idx+1]

    print(f"\n--- Analizando movimiento: {current_target.Name()} ->
{next_target.Name()} ---")

    # Verificamos si hay colisiA3in en la ruta directa
    hay_colision = check_collision_between(current_target, next_target)

    if hay_colision:
        print(f"    D0џ'Ґ A2@eColisiA3in! Tomando ruta de evasiA3in
vA3-a '{home_target.Name()}'")
        # Movimiento de evasiA3in
        robot.MoveJ(home_target)
        # Movimiento al siguiente target
        robot.MoveJ(next_target)
    else:
        print(f"    C2њ… Ruta libre. Moviendo directamente a
{next_target.Name()}")
        # Movimiento directo
        robot.MoveJ(next_target)
```

```
        robot.MoveJ(next_target)

# ========== SECCIA3“N AA3‘ADIDA: MOVIMIENTO FINAL A PIVOTE 1 ==========
print("\n>>> TRAYECTORIA PRINCIPAL FINALIZADA. MOVIENDO A PUNTO FINAL:")

last_target_in_sequence = filtered_targets[-1]
print(f"\n--- Analizando movimiento final:
{last_target_in_sequence.Name()} -> {pivote_target.Name()} ---")

# Verificamos si hay colisiA3in en la ruta directa final
hay_colision_final = check_collision_between(last_target_in_sequence,
pivote_target)

if hay_colision_final:
    print(f"    D0џ’Ґ A2@eColisiA3in! Tomando ruta de evasiA3in vA3-a
'{home_target.Name()}'")
```

5. Código 3: Códigos auxiliares.

4.1 Códigos para el tratamiento de los parámetros.

I. Código de cálculo de la matriz de transformación homogénea.

```
from robodk import robolink, robomath

# ---------- 1. CONFIGURACIA3"N INICIAL ----------
# Defino el target.
TARGET_ORIGEN = "Target_1"
# ---------------------------------------------

print(f">>> Iniciando script de verificaciA3in de pose para: {TARGET_ORIGEN}")

# 2. VINCULACIA3"N CON LA API DE ROBODK
# Establezco la conexiA3in con la instancia de RoboDK en ejecuciA3in.
RDK = robolink.Robolink()

# 3. LOCALIZACIA3"N Y VALIDACIA3"N DEL TARGET DE ORIGEN
# Procedo a buscar e la estaciA3in.
target_origen_item = RDK.Item(TARGET_ORIGEN, robolink.ITEM_TYPE_TARGET)

# Implemento una comprobaciA3in de validez. Si el item no es encontrado
#, lanzo una excepciA3in para detener la ejecuciA3in.
if not target_origen_item.Valid():
    mensaje_error = f"Error de adquisiciA3in: No se ha podido localizar el item
'{TARGET_ORIGEN}'."
    print(mensaje_error)
    RDK.ShowMessage(mensaje_error, True)
    raise Exception(mensaje_error)
# 4. EXTRACCIA3"N DEL VECTOR DE POSE [1x6]
# Para obtener los 6 valores de pose, primero adquiero la matriz [4x4]
# absoluta del target de origen.
matriz_origen = target_origen_item.PoseAbs()
# A continuaciA3in, utilizo la funciA3in 'pose_2_xyzrpw' para descomponer
# la matriz y obtener sus 6 componentes (x,y,z,rx,ry,rz).
try:
    datos_pose_extraidos = robomath.pose_2_xyzrpw(matriz_origen)
except Exception as e:
    mensaje_error = f"Error durante la conversiA3in Matriz -> Vector [1x6]:
{e}"
    print(mensaje_error); RDK.ShowMessage(mensaje_error, True)
    raise Exception(mensaje_error)

print(f"Vector [1x6] extraA3-do de '{TARGET_ORIGEN}': {datos_pose_extraidos}")

# 5. CA3@3LCULO DE LA MATRIZ DESDE EL VECTOR
# Tomo los 'datos_pose_extraidos' (el vector [1x6]) y los utilizo como
# argumento para la funciA3in 'robomath.xyzrpw_2_pose'.

try:
    pose_calculada_final = robomath.xyzrpw_2_pose(datos_pose_extraidos)
```

```
# 6. PRESENTACIA3“N DE RESULTADOS
# Convierto la matriz [4x4] final.
mensaje_matriz_str = str(pose_calculada_final)

# Formateo los 6 valores de pose (redondeando a 3 decimales).
datos_formateados = [round(v, 3) for v in datos_pose_extraidos]

# Construyo el mensaje.
mensaje_final_popup = (f"El {TARGET_ORIGEN} con las coordenadas {datos_formateados},
"
                       f"tiene la siguiente matriz de transformaciA3in
homogA3©nea:\n\n"
                       f"{mensaje_matriz_str}")

# Muestro el mensaje.
RDK.ShowMessage(mensaje_final_popup, True)

print(">>> Script finalizado con Exito.")
```

II. Creación de targets puntuales para buscar diferentes configuraciones de targets.

```
from robodk import robolink, robomath
import os, traceback

RDK = robolink.Robolink()
print(">>> Script iniciado correctamente")

OUTPUT_FILE =
r"C:\Users\Martuki\Desktop\TFG\TFG_Marta\TextosImportantes\targetsPu
ntuales.txt"

# Obtener todos los targets
try:
    targets = RDK.ItemList(robolink.ITEM_TYPE_TARGET)
except Exception as e:
    print("Error al obtener la lista de targets:", e)
    traceback.print_exc()
    targets = []

if not targets:
    raise Exception("No se encontraron targets en la cA3©lula")

print(f">>> Se encontraron {len(targets)} targets")

# Crear carpeta si no existe
os.makedirs(os.path.dirname(OUTPUT_FILE), exist_ok=True)

with open(OUTPUT_FILE, "w", encoding="utf-8") as f_out:
    for t in targets:
        try:
            pose = t.PoseAbs()

            # PosiciA3in XYZ
            xyz = pose.Pos()
            # Para orientaciA3in, probar varios mA3©todos
disponibles
            try:
                # Algunos APIs usan OriXyz o OriRxRyRz
                rxyz = pose.OriXyz()
            except Exception:
                try:
                    rxyz = pose.OriRxRyRz()
                except Exception:
                    # fallback: usar orientaciA3in como identidad si
no disponible
                    rxyz = [0.0, 0.0, 0.0]

            # xyz y rxyz deberA3-an ser listas de 3 floats cada uno
            coords = list(xyz) + list(rxyz)
            coords_str = "[" + ",".join(f"{c:.6f}" for c in coords)
+ "]"
            f_out.write(coords_str + "\n")
            print(f"Target {t.Name()} -> {coords_str}")
        except Exception as e:
            print(f"Error al procesar target {t.Name()}: {e}")
            traceback.print_exc()

mensaje_final = f">>> Lista de coordenadas guardada en
```

III. Eliminar Targets.

```
RDK = robolink.Robolink()
print(">>> Iniciando script de limpieza de targets...")

# --- Buscar todos los targets existentes ---
todos_los_targets = RDK.ItemList(filter=robolink.ITEM_TYPE_TARGET)
print(f">>> Se encontraron {len(todos_los_targets)} targets en total.")

# --- Contadores ---
contador_eliminados = 0
contador_conservados = 0

# --- Bucle de comprobaciA3in y eliminaciA3in ---
for target in todos_los_targets:
    nombre = target.Name()
    conservar = False

    # Regla 1: Comprobar si es un nombre exacto protegido
    if nombre in NOMBRES_PROTEGIDOS_EXACTOS:
        conservar = True

    # Regla 2: Comprobar si coincide con el patrA3in "PIVOTE N"
    elif re.match(PATRON_PIVOTE, nombre):
        conservar = True

    # --- Decidir acciA3in ---
    if conservar:
        print(f"  Conservando: {nombre}")
        contador_conservados += 1
    else:
        print(f"  Eliminando: {nombre}")
        target.Delete()
        contador_eliminados += 1

# --- Resumen final ---
print("\n----------------------------------")
print(">>> PROCESO DE LIMPIEZA COMPLETADO")
print(f"Targets conservados: {contador_conservados}")
print(f"Targets eliminados:  {contador_eliminados}")
print("----------------------------------")
RDK.ShowMessage(f"Limpieza finalizada. Targets eliminados:
{contador_eliminados}", True)
```

IV. Guardar la posición de los Targets que hay creados en ese momento en la célula.

```
from robodk import robolink, robomath
import os, traceback

RDK = robolink.Robolink()
print(">>> Script iniciado correctamente")

OUTPUT_FILE =
r"C:\Users\Martuki\Desktop\TFG\TFG_Marta\TextosImportantes\targetsPuntu
ales.txt"

# Obtener todos los targets
try:
    targets = RDK.ItemList(robolink.ITEM_TYPE_TARGET)
except Exception as e:
    print("Error al obtener la lista de targets:", e)
    traceback.print_exc()
    targets = []

if not targets:
    raise Exception("No se encontraron targets en la cA3©lula")

print(f">>> Se encontraron {len(targets)} targets")

# Crear carpeta si no existe
os.makedirs(os.path.dirname(OUTPUT_FILE), exist_ok=True)

with open(OUTPUT_FILE, "w", encoding="utf-8") as f_out:
    for t in targets:
        try:
            pose = t.PoseAbs()

            # PosiciA3in XYZ
            xyz = pose.Pos()
            # Para orientaciA3in, probar varios mA3©todos disponibles
            try:
                # Algunos APIs usan OriXyz o OriRxRyRz
                rxyz = pose.OriXyz()
            except Exception:
                try:
                    rxyz = pose.OriRxRyRz()
                except Exception:
                    # fallback: usar orientaciA3in como identidad si no
disponible
                    rxyz = [0.0, 0.0, 0.0]

            # xyz y rxyz deberA3-an ser listas de 3 floats cada uno
            coords = list(xyz) + list(rxyz)
            coords_str = "[" + ",".join(f"{c:.6f}" for c in coords) +
"]"
            f_out.write(coords_str + "\n")
            print(f"Target {t.Name()} -> {coords_str}")
        except Exception as e:
            print(f"Error al procesar target {t.Name()}: {e}")
            traceback.print_exc()

mensaje_final = f">>> Lista de coordenadas guardada en {OUTPUT_FILE}"
```

V. Mover el séptimo eje.

```
from robodk import robolink, robomath

# Crear conexiA3in con RoboDK
RDK = robolink.Robolink()

# ParA3@emetros
nombre_robot = "Fanuc R-2000iC/165F"

# Buscar el robot por nombre
robot = RDK.Item(nombre_robot, robolink.ITEM_TYPE_ROBOT)

# Verificar si se encontrA3i el robot
if not robot.Valid():
    print(f"C2ќ@a No se encontrA3i el robot con nombre
'{nombre_robot}'.")
else:
    # Obtener las articulaciones actuales
    joints = robot.Joints().list()

    if len(joints) < 7:
        print("C2љ Cfë@f Este robot no tiene un 7A2є eje (Aeë7).")
    else:
        # Mostrar joints originales
        print(f"C2њ… Articulaciones actuales antes del cambio:")
        for i, joint in enumerate(joints, start=1):
            print(f"Aeë{i}: {joint:.2f}")

        # Cambiar Aeë7 a 1000 mm
        joints[6] = 1000.0

        # Aplicar el cambio al robot
        robot.MoveJ(joints)

        # Confirmar el cambio
        print("\nC2њ… Nueva configuraciA3in aplicada al robot:")
        for i, joint in enumerate(joints, start=1):
            print(f"Aeë{i}: {joint:.2f}")
```

5.1 Códigos para la comprobación de colisiones.

I. Código para comprobar si entre dos targets existe una colisión:

```
from robodk import robolink, robomath
import math, os
import re

# Me conecto a la instancia de RoboDK que tengo abierta.

RDK = robolink.Robolink()
print(">>> Script iniciado correctamente")

# --- Bloque 2: Reiniciar Colisiones ---

RDK.setCollisionActive(robolink.COLLISION_OFF)
RDK.setCollisionActive(robolink.COLLISION_ON)
RDK.Update()
# Hago un "reinicio" del comprobador de colisiones.
# Lo apago y lo enciendo para asegurarme de que no se usen
resultados
# de colisiones anteriores.

# --- Bloque 3: ParA3@emetros de ConfiguraciA3in ---

FRAME_NAME = "Frame P64" # El nombre del sistema de coordenadas de
mi chasis.
N_SAMPLES = 20            # Voy a dividir cada lA3-nea (target a
target) en 20 sub-segmentos para comprobar colisiones.
SHRINK_MM = 0.5           # "Acorto" la lA3-nea 0.5mm por cada lado.
No quiero comprobar justo *en* el target, sino *entre* ellos.
IGNORE_NAMES = ['HTAPULIDOFR'] # Pongo mi herramienta en la lista de
ignorados. No me interesa si choca consigo misma.
# CAMBIO 1: Esta es la ruta donde guardarA3© mi archivo de
resultados.
OUTPUT_FILE =
r"C:\Users\Martuki\Desktop\TFG\TFG_Marta\TextosImportantes\colisione
s_capo.txt"


# --- Bloque 4: Funciones Auxiliares ---
# AquA3- defino mis propias "herramientas" para hacer el cA3idigo
principal mA3@es limpio.

def to_xyz_list(pose): return [float(v) for v in pose.Pos()]
# Una funciA3in simple que he creado para pasar de un objeto "Pose"
de RoboDK a una lista [X, Y, Z].

def interp(a, b, t): return [a[i] + (b[i]-a[i]) * t for i in
range(3)]
# Fi funciA3in de interpolaciA3in. La uso para calcular los 20
puntos intermedios entre un target y el siguiente.
```

```
def shrink_segment(a, b, shrink_mm):
# Esta es la funciA3in que implementa la lA3gica de 'SHRINK_MM'.
# Mueve el punto 'a' 0.5mm hacia 'b' y 'b' 0.5mm hacia 'a', acortando la
lA3-nea que voy a comprobar.
    v = [b[i] - a[i] for i in range(3)]
    norm = math.sqrt(sum(vi*vi for vi in v))
    if norm == 0 or norm <= 2*shrink_mm: return a, b
    s = shrink_mm / norm
    return [a[i] + v[i]*s for i in range(3)], [b[i] - v[i]*s for i in
range(3)]

def parse_collision_result(res):
# La funciA3in de colisiA3in de RoboDK puede devolver cosas diferentes.
# (True/False si hubo colisiA3in, y el Objeto con el que colisionA3i).
    if isinstance(res, (list, tuple)) and len(res) >= 1:
        return bool(res[0]), (res[1] if len(res) > 1 else None)
    return bool(res), None

def sort_key(target_item):
    # FunciA3in para ordenar targets por nA3emero
    match = re.search(r'\d+', target_item.Name())
    return int(match.group()) if match else 0

# --- Bloque 5: ObtenciA3in y Filtrado de Targets ---

# CAMBIO 2: Obtener, filtrar y ordenar targets
frame_ref = RDK.Item(FRAME_NAME, robolink.ITEM_TYPE_FRAME)
if not frame_ref.Valid():
    raise Exception(f"No se pudo encontrar el frame de referencia
'{FRAME_NAME}'")
# Primero, busco mi 'Frame P64' en la estaciA3in. Si no lo encuentra,
paro el script.

all_targets_in_station = RDK.ItemList(robolink.ITEM_TYPE_TARGET)
# Pillo *todos* los targets que existen en mi estaciA3in de RoboDK.

frame_pose_inv = robomath.invH(frame_ref.PoseAbs())
# Obtengo la Pose *Absoluta* de mi "Frame P64" y calculo su inversa.
# UsarA3© esta matriz inversa para convertir las coordenadas absolutas
# de los targets a coordenadas relativas al chasis.

filtered_targets = []
print("\n>>> Analizando y filtrando targets por coordenadas...")
for target in all_targets_in_station:
    relative_pose = frame_pose_inv * target.PoseAbs()
    # Para cada target, calculo su posiciA3in *relativa* al "Frame P64".
    # Es decir, "A2ïDA3inde estA3@e este target *respecto al chasis*?"

    x, y, z = relative_pose.Pos()

    # Compruebo si el target estA3@e dentro de este "cajA3in" (bounding
box) especA3-fico.
    # Solo los targets que estA3©n dentro de esta zona (X entre -960 y
386, Y menor que 945, Z entre 720 y 960)
    # *relativo al chasis*, serA3@en los que aA3±ada a mi lista.
```

```
# Compruebo si el target estA3@e dentro de este "cajA3in" (bounding box)
especA3-fico.
    # Solo los targets que estA3©n dentro de esta zona (X entre -960 y
386, Y menor que 945, Z entre 720 y 960)
    # *relativo al chasis*, serA3@en los que aA3±ada a mi lista.
    if (-960 < x < 386) and (y < 945) and (720 < z < 960):
        filtered_targets.append(target)

if not filtered_targets:
    raise Exception("NingA3єn target cumpliA3i con las condiciones de
filtrado. No se puede continuar.")

# --- Bloque 6: OrdenaciA3in de Targets ---

# Ordenamos los targets filtrados para asegurar la secuencia correcta
filtered_targets.sort(key=sort_key)
# AquA3- uso mi 'sort_key' que definA3- antes.
# 'filtered_targets' ahora contiene solo los targets del capA3i,
# y estA3@en ordenados numA3©ricamente (Target_1, Target_2, Target_3...).

print(f">>> {len(filtered_targets)} targets pasaron el filtro y han sido
ordenados:")
for t in filtered_targets:
    print(f"  - {t.Name()}") # Imprimo la lista ordenada para verificar
que estA3@e bien.

# --- Bloque 7: ComprobaciA3in de Colisiones (El nA3єcleo del script) ---

os.makedirs(os.path.dirname(OUTPUT_FILE), exist_ok=True)
# Me aseguro de que la carpeta donde guardarA3© el .txt exista (si no, la
crea).

with open(OUTPUT_FILE, "w", encoding="utf-8") as f_out:
# Abro el archivo de texto en modo "escritura" ('w').
# 'f_out' es como llamo al archivo mientras estA3© abierto.

    print("\n>>> Iniciando comprobaciA3in de colisiones en targets
filtrados...")
    for idx in range(len(filtered_targets)-1):
        # Itero sobre mi lista de targets, de 0 hasta el penA3єltimo.
        t1, t2 = filtered_targets[idx], filtered_targets[idx+1]

        # Llamo a mi funciA3in de "acortar" el segmento.
        p1s, p2s = shrink_segment(to_xyz_list(t1.PoseAbs()),
to_xyz_list(t2.PoseAbs()), SHRINK_MM)

        # Creo mis 'N_SAMPLES' (20) puntos intermedios a lo largo de esa
lA3-nea acortada.
        points = [interp(p1s, p2s, i/float(N_SAMPLES)) for i in
range(N_SAMPLES+1)]
        collision_detected = False # Reseteo la bandera para este par.
```

```
        for i in range(N_SAMPLES):
            # Ahora compruebo colisiones en los 20 pequeA3±os sub-
segmentos.

            res = RDK.Collision_Line(points[i], points[i+1])
            collision, coll_item = parse_collision_result(res)

            if collision and coll_item and coll_item.Valid():
                #   Si hay colilisiA3in, en alguno de los segmentos!
                name = coll_item.Name()

                # Compruebo si el objeto con el que choca estA3@e en mi
lista de ignorados.
                # Si es la herramienta ('HTAPULIDOFR'), la ignoro y sigo
comprobando.
                if any(ignore in name for ignore in IGNORE_NAMES): continue

                # Si es una colisiA3in REAL (ej. con el chasis):
                mensaje = f"Hay colisiA3in entre {t1.Name()} y {t2.Name()}
con {name} en segmento {i}"
                print(mensaje); f_out.write(mensaje + "\n") # Escribo en la
consola Y en el archivo.
                collision_detected = True # Activo la bandera.
                break # A2@eImportante! Dejo de comprobar este par (T1-T2)
y paso al siguiente (T2-T3).

        if not collision_detected:
            # Si el 'for' de 20 segmentos terminA3i y la bandera sigue en
False,
            # significa que la trayectoria entre T1 y T2 es limpia.
            mensaje = f"No hay colisiA3in entre el {t1.Name()} y el
{t2.Name()}"
            print(mensaje); f_out.write(mensaje + "\n")

# --- Bloque 8: FinalizaciA3in ---

mensaje_final = f">>> Resultados guardados en {OUTPUT_FILE}"
print(mensaje_final)
# Muestro un mensaje
# y dA3inde estA3@en el archivo de resultados.
RDK.ShowMessage(mensaje_final, True)
```

ANEXO 2- MEMORIA ECONÓMICA Y DESGLOSE DE COSTES.

I. Desglose de costes escenario manual

CARGAS SOCIALES PORCENTAJES			
Seguridad Social	0,3327	33,27	%
Contingencias comunes	0,236	23,6	%
Desempleo	0,055	5,5	%
MEI	0,0067	0,67	%
Formación Profesional	0,02	2	%
Accidentes	0,015	1,5	%

COSTE OPERARIOS	**186.578**	**€/año**
Salario operario	35000	€/año
Coste de 1 operario a la empresa	**46644,5**	€/año
N operarios	4	operarios
Coste de sueldos 4 operarios a la empresa	186578	€/año

COSTES INDIRECTOS	**555,53**	**€/año**	
Uniformes	Unidades	coste (€)	total (€)
Ropa	2	80€	160€
Calzado de seguridad	1	45€	45€
EPIS			0€
Tapones y cascos de seguridad			0€
Gafas	3	9,42€	28,26€
Mascarillas con filtro P3 (polvo fino)	5	20€	100€
Mandil de cuero	4	17,43€	69,72€
Guantes antivibración	3	19,05€	57,15€
Filtro P3	12	7,95€	95,4€

HERRAMIENTAS TOTAL	**155.65,45455**	**€/año**	
Herramienta	**unidades/año**	**Precio u (€)**	**total**
LIJADORA industrial de mano	2	222,2€	444,4€
Recambio hojas de pulido	57600	0,2€	11.520€
Horas de servicio disco lija (h)	0,066666667		
Horas de servicio lijadora de mano (h)	3840		

* NOTA CAMBIO CADA 4 MINUTOS			

PULIDORA			
Pulidora industrial	2	680€	1.360€
Recambio toallitas	349,0909091	5,99€	2.091,05€
Horas de servicio de la toallita (h)	11		
Horas de servicio de la pulidora industrial (h)	3840		
EXTRAS DE REPARACIÓN ANUAL			150€

3840		
Tiempo de ciclo	105,32	segundos
Defectos	22	defectos
Defectos hora	751,9939233	defectos/hora
Defectos en 1 año teniendo en cuenta que trabajan 3840	2887656,665	defectos al año

II. Costes asociados a la implantación de la célula robotizada.

COSTES DE ADQUISICIÓN			157.000€
Unidades	Ítem	Precio	Unidades
1	Fanuc R-2000iC/165F	65.000€	1
1	Herramienta de SAND	35.000€	1
1	Herramienta WIPE	35.000€	1
1	Mecanismo de fijación	7.000€	1
1	7 eje lineal	150.00€	1

COSTES DE LA INSTALACIÓN				43200€
	Operarios	Función	Desempeño	Desembolso
	1	Gestor del proyecto	2,5 semanas	4.000€
	6	Técnicos. Mecánicos	2,5 semanas	16.800€
	7	Técnicos eléctricos	2,5 semanas	19.600€
	1	Técnicos de Robótica	2,5 semanas	2.800€

COSTES DE ADECUACIÓN LOCAL				18.000€
MATERIALES DURANTE LA INSTALACIÓN		18000	€	
Unidades	Ítem	Precio/u		Total
1	Perimetral de seguridad Serie 2000, incluyendo transporte e instalación	18000	€	

CAPACITACIÓN Y PUESTA EN MARCHA	16.000€
Software para el robot	8.500€
Software para la simulación	3.000€
Asistencia de arranque	4.500€

5. Costes operativos

ENERGÍA CONSUMIDA											
	Año 0	Año 1	Año 2	Año 3	Año 4	Año 5	Año 6	Año7	Año 8	Año 9	Año 10
€ /KWh	0,15	0,155	0,159	0,162	0,164	0,167	0,17	0,173	0,176	0,179	0,182
kW/año	40.320	41.664	42.739,2	43.545,6	44.083,2	44.889,6	45.696	46.502,4	47.308,8	48.115,2	48.921,6

CONSUMIBLES	4.733,82	€/año	
CONSUMIBLES			
Tiempo de ciclo	145,32	segundos	55
Defectos	22	defectos	
Defectos hora	751,9939233	defectos/hora	
Defectos en 1 año teniendo en cuenta que trabajan 3840	2887656,665	defectos al año	
Vida Recambio lijadora h	0,75		
Wipe reutilizable h	18		
Lijad	5120		
Wipes	213,3333333		
Horas de trabajo de la máquina ponemos mismas que manual	3840		
Consumible	unidades	Precio	total
Lija	18	6,99	125,82€
Wipe	5120	0,9	4.608€

SALARIO TRABAJADORES	60.000€/año
Revisor de la célula 1	30.000 €
Revisor de la célula 2	30.000€

INVERSIÓN INICIAL	210.700 €

COSTES OPERATIVOS											
COSTE ASOCIADO	AÑO 0	AÑO 1	AÑO 2	AÑO 3	AÑO 4	AÑO 5	AÑO 6	AÑO 7	AÑO 8	AÑO 9	AÑO 10
ENERGÍA CONSUMIDA €/año	40.320,00	41.664,00	42.739,20	43.545,60	44.083,20	44.889,60	45.696,00	46.502,40	47.308,80	48.115,20	48.921,60
MANTENIMIENTO €/año	0,00	0,00	3.500,00	0,00	3.500,00	0,00	3.500,00	0,00	3.500,00	8.500,00	0,00
CONSUMIBLES €/año	4.733,82	4.733,82	4.733,82	4.733,82	4.733,82	4.733,82	4.733,82	4.733,82	4.733,82	4.733,82	4.733,82
SEGURO TÉCNICO €/año	3.675,00	3.675,00	3.675,00	3.675,00	3.675,00	3.675,00	3.675,00	3.675,00	3.675,00	3.675,00	3.675,00
DEPRECIACIÓN €/año	23.420,00	23.420,00	23.420,00	23.420,00	23.420,00	23.420,00	23.420,00	23.420,00	23.420,00	23.420,00	23.420,00
SALARIO DE LOS TRABAJADORES €/año	60000,00	60000,00	60000,00	60000,00	60000,00	60000,00	60000,00	60000,00	60000,00	60000,00	60000,00
TOTAL COSTES OPERATIVOS	**1321.48,82**	**133.492,82**	**138.068,02**	**135.374,42**	**139.412,02**	**136.718,42**	**141.024,82**	**138.331,22**	**142.637,62**	**148.444,02**	**140.750,42**

TOTAL COSTES OPERATIVOS€/año	1321.48,82	133.492,82	138.068,02	135.374,42	139.412,02	136.718,42	141.024,82	138.331,22	142.637,62	148.444,02	140.750,42

AHORRO ANUAL	70.550,16 €	69.206,16 €	64.630,96 €	67.324,56 €	63.286,96 €	65.980,56 €	61.674,16 €	64.367,76 €	60.061,36 €	54.254,96 €	61.948,56€

Valor actual Neto (VAN)	324.945,40 €
Ahorro total en 10 años	703.286,21 €
Tasa interna de retorno (TIR)	7%
Payback	3 años