Universidad Miguel Hernández de Elche

MASTER UNIVERSITARIO EN ROBÓTICA

“Modelos de difusión para generar movimientos 3D de un brazo robótico a partir de texto”

Trabajo de Fin de Máster

Curso académico 2023-2024

Autor: Álvaro Sáez Tonda
Tutor/es: Adrián Peidro Vidal

RESUMEN

El presente trabajo de investigación se centra en la implementación de un modelo para la generación de poses 3D de un brazo robótico con 6 grados de libertad, utilizando modelos de difusión a partir de prompts de texto. Inspirado en la reciente evolución de la inteligencia artificial y los modelos de generación de imágenes, este proyecto busca aplicar estas tecnologías en el ámbito de la robótica, específicamente en la manipulación robótica de precisión.

La investigación se estructura en varias fases: la primera fase consiste en la adaptación de un modelo de difusión especializado en la generación de poses 3D humanas, denominado *MotionDiffuse: Text-Driven Human Motion Generation with Diffusion Model* de Mingyuan Zhang, Zhongang Cai, Liang Pan, Fangzhou Hong, Xinying Guo, Lei Yang, y Ziwei Liu (2022), enfocándose únicamente en las poses de uno de los brazos. A partir de estas poses, se procederá a ajustar las posiciones de cada articulación del brazo robótico, considerando los parámetros físicos necesarios como masas y momentos de inercia.

En la segunda fase, se desarrollará el cálculo de la cinemática del brazo robótico, abordando tanto la cinemática directa como inversa para asegurar una correcta implementación de las poses generadas. Este paso es crucial para entender cómo cada movimiento impacta en la posición final del brazo y para permitir el control preciso del mismo a través de comandos de texto.

La tercera fase se dedicará al análisis dinámico del brazo, donde se simularán las fuerzas y torques necesarios para realizar las tareas definidas. Este análisis incluirá la selección de motores adecuados que puedan cubrir las necesidades operativas del brazo, asegurando que los actuadores elegidos sean capaces de proporcionar el rendimiento requerido.

Finalmente, se implementará una serie de simulaciones y pruebas prácticas para validar el sistema desarrollado. Se evaluará la precisión y eficiencia del modelo de difusión en la generación de poses útiles y su impacto en la operatividad del brazo robótico.

Este trabajo no solo pretende avanzar en la integración de técnicas avanzadas de IA en la robótica, sino también proporcionar una herramienta versátil que pueda adaptarse a diferentes tareas y escenarios mediante simples comandos de texto. La combinación de modelos de difusión con la robótica promete abrir nuevas posibilidades en la programación

y control de robots, facilitando su uso en diversas aplicaciones industriales y de investigación.

Palabras clave: Modelos de difusión; Generación de poses 3D; Brazo robótico; Cinemática; Dinámica; Inteligencia artificial; Robótica.

RESUM

El present treball d'investigació se centra en la implementació d'un model per a la generació de posicions 3D d'un braç robòtic amb 6 graus de llibertat, utilitzant models de difusió a partir de prompts de text. Inspirat en la recent evolució de la intel·ligència artificial i els models de generació d'imatges, aquest projecte busca aplicar aquestes tecnologies en l'àmbit de la robòtica, específicament en la manipulació robòtica de precisió.

La investigació s'estructura en diverses fases: la primera fase consisteix en l'adaptació d'un model de difusió especialitzat en la generació de posicions 3D humanes, denominat *MotionDiffuse: Text-Driven Human Motion Generation with Diffusion Model* de Mingyuan Zhang, Zhongang Cai, Liang Pan, Fangzhou Hong, Xinying Guo, Lei Yang, i Ziwei Liu (2022), centrant-se únicament en les posicions d'un dels braços. A partir d'aquestes posicions, es procedirà a ajustar les posicions de cada articulació del braç robòtic, considerant els paràmetres físics necessaris com masses i moments d'inèrcia.

En la segona fase, es desenvoluparà el càlcul de la cinemàtica del braç robòtic, abordant tant la cinemàtica directa com inversa per assegurar una correcta implementació de les posicions generades. Aquest pas és crucial per entendre com cada moviment impacta en la posició final del braç i per permetre el control precís d'aquest mitjançant comandaments de text.

La tercera fase es dedicarà a l'anàlisi dinàmica del braç, on se simularan les forces i torques necessaris per realitzar les tasques definides. Aquesta anàlisi inclourà la selecció de motors adequats que puguen cobrir les necessitats operatives del braç, assegurant que els actuadors escollits siguen capaços de proporcionar el rendiment requerit.

Finalment, s'implementarà una sèrie de simulacions i proves pràctiques per validar el sistema desenvolupat. S'avaluarà la precisió i eficiència del model de difusió en la generació de posicions útils i el seu impacte en l'operativitat del braç robòtic.

Aquest treball no només pretén avançar en la integració de tècniques avançades d'IA en la robòtica, sinó també proporcionar una eina versàtil que puga adaptar-se a diferents tasques i escenaris mitjançant simples comandaments de text. La combinació de models de difusió amb la robòtica promet obrir noves possibilitats en la programació i control de robots, facilitant el seu ús en diverses aplicacions industrials i de recerca.

Paraules clau: Models de difusió; Generació de posicions 3D; Braç robòtic; Cinemàtica; Dinàmica; Intel·ligència artificial; Robòtica.

ABSTRACT

This research work focuses on the implementation of a model for generating 3D poses of a robotic arm with 6 degrees of freedom using diffusion models based on text prompts. Inspired by the recent evolution of artificial intelligence and image generation models, this project aims to apply these technologies in the field of robotics, specifically in precision robotic manipulation.

The research is structured in several phases: the first phase involves adapting a diffusion model specialized in generating human 3D poses, called *MotionDiffuse: Text-Driven Human Motion Generation with Diffusion Model* by Mingyuan Zhang, Zhongang Cai, Liang Pan, Fangzhou Hong, Xinying Guo, Lei Yang, and Ziwei Liu (2022), focusing only on the poses of one arm. Based on these poses, the positions of each joint of the robotic arm will be adjusted, considering the necessary physical parameters such as masses and moments of inertia.

In the second phase, the kinematics of the robotic arm will be calculated, addressing both direct and inverse kinematics to ensure the correct implementation of the generated poses. This step is crucial for understanding how each movement impacts the final position of the arm and for allowing precise control of the arm through text commands.

The third phase will be dedicated to the dynamic analysis of the arm, where the necessary forces and torques to perform the defined tasks will be simulated. This analysis will include the selection of suitable motors that can cover the operational needs of the arm, ensuring that the chosen actuators are capable of providing the required performance.

Finally, a series of simulations and practical tests will be implemented to validate the developed system. The precision and efficiency of the diffusion model in generating useful poses and its impact on the operability of the robotic arm will be evaluated.

This work not only aims to advance the integration of advanced AI techniques in robotics but also to provide a versatile tool that can adapt to different tasks and scenarios through simple text commands. The combination of diffusion models with robotics promises to open new possibilities in robot programming and control, facilitating their use in various industrial and research applications.

Keywords: Diffusion models; 3D pose generation; Robotic arm; Kinematics; Dynamics; Artificial intelligence; Robotics.

AGRADECIMIENTOS

A mis padres, por el apoyo incondicional durante los años de estudio.

ÍNDICE

1. INTRODUCCIÓN

1.1. Objeto de estudio

El objeto de estudio de esta investigación es el desarrollo de un sistema que integre un modelo de difusión para la generación de poses 3D de un brazo robótico con 6 grados de libertad a partir de prompts de texto. Este estudio se centrará en adaptar un modelo de difusión especializado en la generación de poses 3D humanas, enfocándose exclusivamente en las poses de un brazo, y en cómo estas pueden ser traducidas y aplicadas al control de un brazo robótico. El objetivo es diseñar un sistema que permita el cálculo preciso de la cinemática y dinámica del brazo robótico, incluyendo la selección de motores adecuados. Además, se pretende implementar simulaciones y pruebas prácticas para validar la efectividad del sistema desarrollado, proporcionando así una herramienta versátil y adaptativa para la programación y control de robots mediante comandos de texto.

1.2. Justificación

El interés personal en la investigación se sustenta en la dedicación profesional del autor a la robótica, con un vínculo especialmente fuerte con la inteligencia artificial. La generación de movimiento en sistemas robóticos es uno de los principales retos actuales y específicos del trabajo actual del autor, por lo que este proyecto no solo representa una oportunidad para introducirle más profundamente en este campo, sino también para reforzar y expandir el conocimiento que ya tiene sobre la generación de acciones mediante sistemas robóticos impulsados por la IA.

El interés académico de la temática radica en la posibilidad de contribuir al avance del conocimiento en la integración de técnicas de IA en la robótica, específicamente en la generación de poses y movimientos precisos. Al desarrollar y validar un modelo de difusión capaz de traducir comandos de texto en poses 3D útiles para un brazo robótico, este trabajo pretende proporcionar una herramienta innovadora y versátil que pueda adaptarse a diversas aplicaciones industriales y de investigación. Asimismo, busca facilitar el control y programación de robots, promoviendo una mayor eficiencia y precisión en la ejecución de tareas robóticas complejas.

1.3. Estado de la cuestión

Previo al desarrollo del cuerpo de la investigación, se ha llevado a cabo una revisión de los principales trabajos vinculados al corazón conceptual del proyecto, en concreto, los modelos de difusión, sus aplicaciones en robótica, el cálculo de la cinemática y dinámica de mecanismos robóticos y las aplicaciones de la IA a nivel general en robótica.

Sobre los modelos de difusión en inteligencia artificial:

Denoising Diffusion Probabilistic Models de Jonathan Ho, Ajay Jain y Pieter Abbeel (2020): Este artículo presenta un nuevo enfoque para la generación de imágenes de alta calidad basándose en los ampliamente reconocidos datasets CIFAR-10 y Celeba-HQ. Se introduce un proceso de difusión basado en el comportamiento termodinámico del movimiento de conjuntos de partículas que tienden a estabilizarse entre estados de alta y baja concentración. De esta manera se añade ruido a los datos y el modelo aprende a revertir este proceso de cara a obtener una mayor estabilidad en el entrenamiento y una mayor calidad de las imágenes generadas.

Anteriormente, el estado del arte en la generación de imágenes, las Generative Adversarial Networks, presentaban altos problemas de estabilidad en la generación que se vieron resueltos por estos sistemas, por lo que han supuesto desde entonces una alternativa reemplazante.

Se introduce con este trabajo la generación de datos mediante modelos de difusión, que va a ser el concepto fundamental de la generación de poses para las articulaciones del brazo robótico sujeto de los experimentos durante la presente investigación.

Score-Based Generative Modeling through Stochastic Differential Equations de Yang Song y Stefano Ermon (2021): Este artículo enfoca el proceso de denoising ya introducido a través del uso de ecuaciones diferenciales estocásticas (SDE) y el proceso de Langevin para definir una trayectoria continua que conecta el ruido generado con los datos reales que se pretenden replicar en la generación de imágenes.

Esta generación de datos se realiza mediante un modelo de puntuaciones. En lugar de agregar ruido en pasos discretos, calcula una puntuación o gradiente que define cómo modificar los datos para acercarlos a una distribución objetivo en cada paso. El proceso de Langevin en este contexto rescata la definición del movimiento de una partícula en un medio viscoso

sometida a fuerzas aleatorias y fricción para simular la estabilización hacia un estado de menor energía. El uso de las ecuaciones diferenciales del movimiento del sistema ofrece un control más refinado y una mayor precisión general.

La implicación fundamental en la generación de trayectoria y movimiento aplicable a mecanismos robóticos subyace en el ajuste más fino de las poses, especialmente útil en aplicaciones que requieran detalle y ajuste dinámico en tiempo real.

Sobre las aplicaciones de modelos de difusión en robótica:
Diffusion Models for Reinforcement Learning: A Survey de Zhengbang Zhu, Hanye Zhao, Haoran He, Yichao Zhong, Shenyu Zhang, Haoquan Guo (2024): Este artículo recoge la teoría presentada por los trabajos fundacionales de los modelos de difusión para el entrenamiento de acciones llevadas a cabo por sistemas robóticos, concretamente para las posiciones en el tiempo de sus articulaciones, centros de masa, de inercia, etc. Los datos generados por los modelos de difusión sirven de input de sistemas de aprendizaje por refuerzo que, a través de sus políticas de premio-castigo permiten la asimilación de diferentes comportamientos a los sistemas en función de las situaciones que el entorno pueda ofrecer.

Esta integración permite generar movimientos más naturales y precisos en comparación con las aproximaciones anteriores basadas en enfoques autoregresivos.

Model-Based Diffusion for Trajectory Optimization de Chaoyi Pan, Zeji Yi, Guanya Shi, Guannan Qu (2024): Este trabajo introduce una aproximación a la optimización de trayectorias utilizando modelos de difusión basados en el modelo del sistema, lo que lo diferencia de enfoques anteriores que eran *model-free*. Aprovecha la información del propio sistema dinámico del robot para calcular la función de puntuación, mejorando la capacidad de generar trayectorias más precisas sin necesidad de datos externos. Para sistemas robóticos, esto implica movimientos optimizados incluso en tareas complejas, como en robots de contacto con múltiples puntos de interacción.

La integración del modelo dinámico del robot permite una mayor generalización y adaptabilidad a nuevas configuraciones de robots, mientras que los enfoques anteriores se limitaban a entornos preentrenados o simulaciones sin explotar completamente el sistema dinámico del robot. Esto representa un avance significativo para la implementación de modelos de difusión en robótica, ya que permite optimizar movimientos en tiempo real y con datos incompletos o imperfectos.

Sobre la Cinemática y Dinámica de Brazos Robóticos:

Robot Dynamics and Control de Mark W. Spong y M. Vidyasagar (2005): Este libro supone un documento de referencia esencial para comprender los principios de la dinámica y el control de robots. Obra ampliamente reconocida y utilizada en el campo de la robótica en universidades, laboratorios de investigación y proyectos industriales a nivel global. Proporciona una base matemática rigurosa para modelar y controlar los movimientos robóticos, cubriendo temas como cinemática, dinámica de cuerpos rígidos y algoritmos de control. En el contexto de los modelos de difusión aplicados a la generación de trayectorias robóticas, este libro es crucial para entender cómo aplicar principios de control y optimización de movimientos, asegurando trayectorias precisas y eficientes basadas en las características del sistema físico del robot.

Este marco teórico ayuda a diseñar algoritmos de control que puedan beneficiarse de técnicas avanzadas como los modelos de difusión, optimizando el comportamiento de robots en entornos reales.

Introduction to Robotics: Mechanics and Control de John J. Craig (2018): Supone una lectura más enfocada en la mecánica y los fundamentos del control desde un punto de vista más accesible y práctico.

Su relevancia en el contexto de la aplicación de modelos de difusión a mecanismos robóticos radica en su enfoque en algoritmos de control, planificación de trayectorias y estabilidad. Proporciona una base para comprender mejor cómo los robots responden al control dinámico, lo que es esencial cuando se integran modelos de difusión para optimizar el movimiento.

Sobre la integración de IA en la robótica:

Artificial Intelligence: A Modern Approach de Stuart Russell y Peter Norvig (2021): Al igual que *Robot Dynamics and Control*, este libro supone una referencia internacional en cuanto a la concepción de la inteligencia artificial y su transversalidad en la implementación de diferentes soluciones, entre ellas la robótica. Proporciona conceptos clave acerca del aprendizaje automático, búsqueda, toma de decisiones y algoritmos de optimización.

Aunque su redacción es muy anterior a la introducción y estandarización de los modelos de difusión, sus principios teóricos sustentan los diseños de las arquitecturas actuales y el

enfoque de la algoritmia que recoge permite mejorar la toma de decisiones y el control de trayectorias robóticas en entornos dinámicos.

Deep Learning for Robot Perception and Cognition de Alexandros Iosifidis y Anastasios Tefas (2022): Ahonda en la investigación de la IA aplicada en robótica desde puntos de vista complementarios a la moción del mecanismo, abordando técnicas de deep learning aplicables a la percepción y la cognición. Cómo percibir el entorno de manera más eficiente y la repercusión en la toma de decisiones de manera autónoma basada en el procesamiento de datos sensoriales complejos son algunos de sus principales pilares.

1.4. Objetivos

El objetivo principal de esta investigación es:

- Integrar un modelo de difusión para la generación de poses 3D de un brazo robótico con 6 grados de libertad a partir de prompts de texto.

Como objetivos secundarios:

- Adaptar un modelo de difusión especializado en la generación de poses 3D humanas, enfocándose exclusivamente en las poses de un brazo y su traducción al control de un brazo robótico.
- Desarrollar el cálculo preciso de la cinemática del brazo robótico, abordando tanto la cinemática directa como inversa.
- Realizar un análisis dinámico del brazo robótico para simular las fuerzas y torques necesarios en la ejecución de tareas, incluyendo la selección de motores adecuados.
- Implementar una serie de simulaciones y pruebas prácticas para validar la efectividad y precisión del sistema desarrollado en la generación de poses y su impacto en la operatividad del brazo robótico.
- Proporcionar una herramienta versátil y adaptativa para la programación y control de robots mediante comandos de texto, facilitando su uso en diversas aplicaciones industriales y de investigación.

1.5. Metodología

La metodología para la elaboración del proyecto de investigación se divide en los siguientes bloques:

Revisión Bibliográfica: Se inspeccionarán bases de datos académicas como IEEE, Google Scholar y ArXiv para identificar estudios clave y textos sobre modelos de difusión y su integración en la cinemática y dinámica de los brazos robóticos.

Modelado Cinemático: Se empleará el método de Denavit-Hartenberg para la cinemática inversa, debido a la sencillez de su aplicación respecto a otros métodos y a su representación gráfica estructurada. Contamos además con la configuración del robot de tres grados de libertad sujeto de estudio del trabajo realizado en la asignatura de Cinemática de Robots. Las simulaciones se realizarán en Python.

Análisis Dinámico: Se implementará la formulación de Newton-Euler para modelar la dinámica del brazo, simulando fuerzas y torques en Python.

Integración de Modelos de Difusión: Se adaptará el modelo *MotionDiffuse*, extraído de la librería con el mismo nombre, basada en el paper *MotionDiffuse: Text-Driven Human Motion Generation with Diffusion Model* de Mingyuan Zhang, Zhongang Cai, Liang Pan, Fangzhou Hong, Xinying Guo, Lei Yang, y Ziwei Liu (2022), para la extracción de poses concretas de esqueletos humanoides para su adaptación a la distribución de articulaciones de un brazo robótico. Se extraerá el código base directamente de la librería para su adaptación a las condiciones del problema.

Validación: Las pruebas finales se realizarán en simulaciones con la librería de visualización de Python Matplotlib, y se evaluará el movimiento tomando como referencia le movimiento generado por la pose de referencia original al correr el código de la librería *MotionDiffuse*.

1.6. Estructura

Este trabajo de investigación está organizado en diez capítulos principales, cada uno con un propósito y contenido específico que contribuye al desarrollo y análisis del modelo de difusión para la generación de poses 3D en un brazo robótico con 6 grados de libertad.

Capítulo 1: Introducción.

Este capítulo establece el marco general del estudio y se subdivide en seis secciones:

- Objeto de estudio: Define el tema central de la investigación.
- Justificación: Explica el interés, la relevancia y la necesidad del estudio.
- Estado de la cuestión: Revisa la literatura existente sobre el tema.
- Objetivos: Especifica las metas principales y secundarias de la investigación.

- Metodología: Detalla los métodos y recursos utilizados para llevar a cabo el análisis.
- Estructura: Describe la organización del trabajo.

Capítulo 2: Contextualización. Los modelos de difusión para el control robótico.

- Los modelos de difusión: Contextualización de la arquitectura de generación de datos mediante técnicas de difusión y sus actuales aplicaciones.
- Cinemática de brazos robóticos: Modela el comportamiento cinemático de los brazos robóticos que se utilizarán durante el desarrollo del proyecto para obtener los movimientos de las articulaciones dadas unas posiciones definidas por el modelo de difusión.
- Dinámica de brazos robóticos: Modela el comportamiento dinámico de los brazos robóticos que también se usarán durante el desarrollo del proyecto a partir de la cinemática para caracterizar los recursos para la puesta en movimiento hacia las posiciones definidas por el modelo de difusión.
- La integración de la IA en la robótica: Comenta diferentes aplicaciones actuales de la IA en los sectores en los que la robótica refleja un mayor impacto.

Capítulo 3: De texto a acción mediante brazo robótico de tres grados de libertad.

Aquí se detalla el desarrollo práctico del modelo y su implementación:

- De texto a animación 3D mediante modelos de difusión: Adaptación y personalización del modelo.
- Obtención de la cinemática y la dinámica del mecanismo: Desarrollo del código para cálculos cinemáticos y ejecución de simulaciones dinámicas.
- Diseño de Motores: Selección de motores que posibilitan el movimiento generado por el modelo de difusión.

Capítulo 4: Propuesta de diseño y control completa.

Este capítulo se dedica a la validación del sistema desarrollado mediante la ejecución de simulaciones para verificar el ajuste del modelo, precisión y eficiencia.

Capítulo 5: Conclusiones y próximos pasos.

Resumen de los hallazgos y aprendizajes del estudio, incluyendo:

- Resultados Clave: Principales resultados obtenidos y su análisis.

- Implicaciones: Impacto y relevancia de los hallazgos en el campo de la robótica.
- Futuras Líneas de Investigación: Sugerencias para investigaciones futuras basadas en los resultados obtenidos.

Capítulo 6: Bibliografía

Listado de las fuentes bibliográficas consultadas y citadas.

Capítulo 7: Webgrafía

Listado de las fuentes webgráficas consultadas.

Capítulo 8: Índice de ilustraciones

Guía visual de las figuras utilizadas.

Capítulo 9: Índice de tablas

Registro de tablas incluidas en el trabajo.

Capítulo 10: Anexo

Recopilación del código utilizado para desarrollar end-to-end el sistema, desde la obtención del movimiento a partir de texto hasta la simulación y del movimiento del brazo robótico y diseño de este.

2. CONTEXTUALIZACIÓN. LOS MODELOS DE DIFUSIÓN PARA EL CONTROL ROBÓTICO.

2.1. Los modelos de difusión

Los modelos de difusión han emergido como una clase poderosa de modelos generativos, captando una atención significativa en los últimos años debido a su capacidad para generar muestras de alta calidad y su estabilidad durante el entrenamiento. Estos modelos utilizan un marco de eliminación de ruido que puede invertir eficazmente un proceso ruidoso en múltiples pasos para generar nuevos datos (Ho, Jain, & Abbeel, 2020). A diferencia de modelos generativos anteriores como los Autoencoders Variacionales (VAE) y las Redes Generativas Adversarias (GAN), los modelos de difusión muestran capacidades superiores en la generación de muestras de alta calidad y demuestran una mayor estabilidad en el entrenamiento (Song & Ermon, 2021). Esta superioridad ha permitido que los modelos de difusión logren éxitos sustanciales en diversos dominios, incluyendo la visión por computador, el procesamiento de lenguaje natural, la generación de audio y el descubrimiento de fármacos (Austin et al., 2021; Schneuing et al., 2023).

Los modelos de difusión se basan en un proceso de eliminación de ruido que, de manera iterativa, refina muestras iniciales de ruido blanco hasta alcanzar distribuciones de datos deseadas. Este proceso se puede entender como una cadena de Markov inversa, donde cada paso de la cadena elimina una pequeña cantidad de ruido, recuperando así la estructura subyacente de los datos originales. Este enfoque permite que los modelos de difusión generen datos de manera más estable y con una mayor calidad en comparación con los métodos tradicionales (Ho et al., 2020).

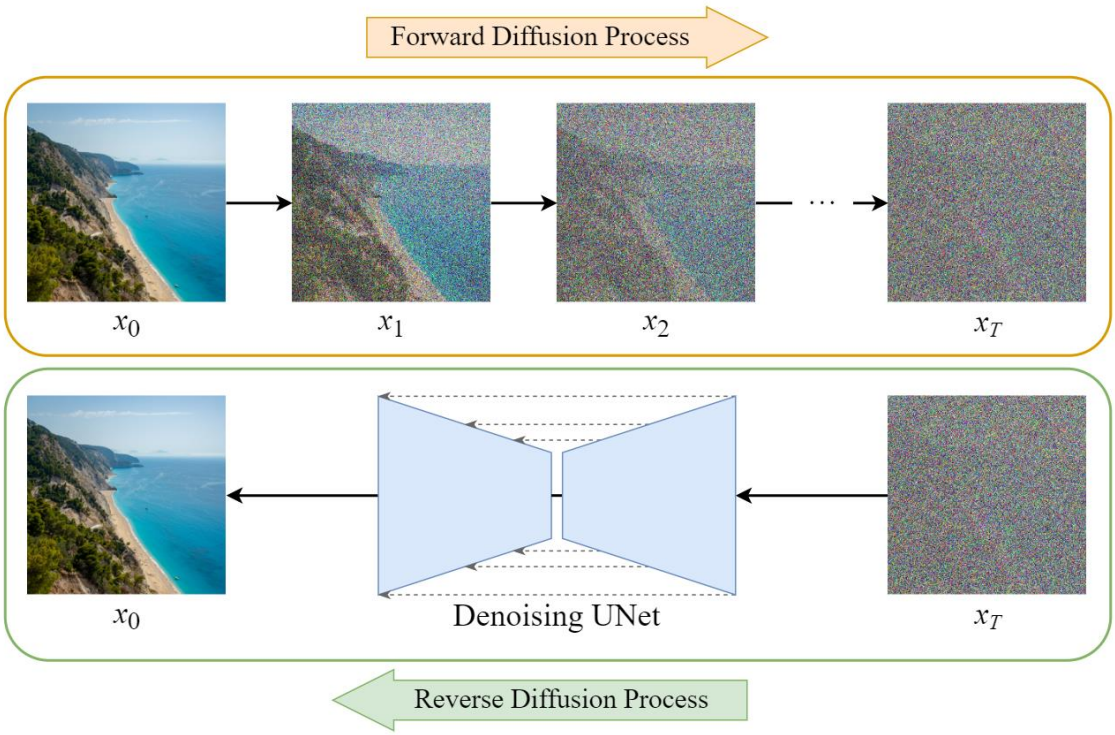

Figura 1. Proceso de Forward y Reverse Diffusion para la generación de imágenes con modelos de difusión.

A diferencia de los VAE, que tienden a generar muestras más borrosas debido a su naturaleza probabilística, los modelos de difusión pueden producir imágenes y datos con detalles finos. Además, en contraste con las GAN, que a menudo sufren problemas de estabilidad durante el entrenamiento y pueden generar muestras de baja calidad, los modelos de difusión han demostrado ser más estables y consistentes(Song & Ermon, 2021). Esta estabilidad proviene de su diseño iterativo, que permite un refinamiento gradual de las muestras en lugar de depender de un único paso generativo.

Aplicaciones en diversos dominios:

1. Visión por computador: Los modelos de difusión han logrado resultados sobresalientes en la generación de imágenes de alta resolución, demostrando su capacidad para captar detalles finos y texturas complejas (Ho et al., 2020).
2. Procesamiento de lenguaje natural: En el ámbito del lenguaje, estos modelos se han utilizado para generar textos coherentes y contextualmente relevantes, superando a otros modelos en tareas de generación de texto (Austin et al., 2021).

3. Generación de audio: Han sido aplicados con éxito en la síntesis de audio, produciendo muestras de sonido de alta fidelidad que son indistinguibles de las grabaciones humanas (Kong et al., 2020).
4. Descubrimiento de fármacos: En la biomedicina, los modelos de difusión han facilitado el descubrimiento de nuevas moléculas y compuestos, acelerando el proceso de desarrollo de fármacos (Schneuing et al., 2023).

A pesar de sus ventajas, los modelos de difusión también se enfrentan a ciertos desafíos. Uno de los principales es la necesidad de un gran poder computacional debido a su naturaleza iterativa. Además, aunque han demostrado una gran capacidad en la generación de datos de alta calidad, su aplicación en tareas con restricciones temporales o en tiempo real sigue siendo un área de investigación activa.

Las investigaciones actuales se centran en mejorar la eficiencia computacional de estos modelos y explorar su integración con otros enfoques generativos para aprovechar sus respectivas fortalezas. También es esencial investigar cómo los modelos de difusión pueden ser adaptados y aplicados en nuevos dominios, incluyendo aquellos con restricciones más estrictas y requerimientos de tiempo real, como es la generación de animación o movimiento en 3D para la realización de tareas generalistas por robots humanoides.

Los modelos de difusión, conocidos por su capacidad de generar datos de alta calidad y su estabilidad en el entrenamiento, han encontrado aplicaciones prometedoras en el ámbito de la robótica, especialmente en la robótica de brazos con múltiples grados de libertad (6 DoF, por sus siglas en inglés). Estos modelos están siendo integrados en soluciones de aprendizaje por refuerzo (RL) para abordar desafíos específicos en la robótica.

En el aprendizaje por refuerzo, los modelos de difusión se utilizan para generar trayectorias y políticas que mejoran la eficiencia y la generalización en tareas de control complejas. La integración de estos modelos en RL se ha mostrado efectiva para superar problemas comunes como la baja eficiencia de muestra, la escasez de datos en experiencias de repetición y los errores acumulativos en el modelado basado en planificación (Nagabandi et al., 2018; Schrittwieser et al., 2020; Xiao et al., 2019). Por ejemplo, en el trabajo de Janner et al. (2022), el modelo Diffuser genera trayectorias de alta calidad en un conjunto de datos offline, permitiendo la planificación de trayectorias futuras deseadas mediante muestreo guiado.

Desafíos abordados por los modelos de difusión:

1. Expresividad restringida en el aprendizaje offline: Los modelos de difusión pueden representar cualquier distribución normalizable, lo que les permite mejorar el ajuste en conjuntos de datos complejos, superando las limitaciones de las políticas gaussianas convencionales (Wang et al., 2023).
2. Escasez de datos en la repetición de experiencias: La capacidad de los modelos de difusión para generar datos sintéticos de alta fidelidad permite aumentar la eficiencia de aprendizaje en entornos con espacios de estado y acción de alta dimensionalidad y patrones de interacción complejos (Lu et al., 2023b).
3. Errores acumulativos en la planificación basada en modelos: Al operar a nivel de trayectorias, los modelos de difusión mejoran la consistencia temporal y reducen los errores acumulativos que surgen en enfoques autoregresivos (Xiao et al., 2019).
4. Generalización en el aprendizaje multitarea: Los modelos de difusión pueden manejar distribuciones multimodales en conjuntos de datos multitarea, adaptándose a nuevas tareas mediante la estimación de la distribución de tareas (He et al., 2023a).

Los modelos de difusión superan a otros modelos generativos como los VAE y GAN en términos de estabilidad del entrenamiento y calidad de las muestras generadas. Su capacidad para refinar iterativamente muestras desde una distribución inicial de ruido blanco hasta alcanzar la distribución de datos deseada les permite mantener detalles finos y texturas complejas en las muestras generadas (Ho et al., 2020).

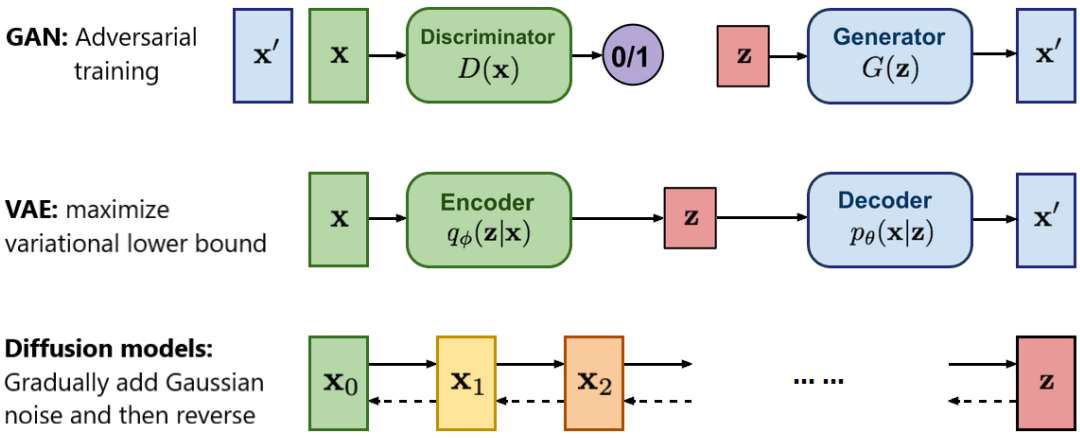

Figura 2. Comparativa entre el entrenamiento de GANs, VAEs y modelos de difusión.

En la robótica de brazos con 6 DoF, los modelos de difusión se están utilizando para:

1. Generación de trayectorias: Generan trayectorias de movimiento de alta calidad, esenciales para tareas de manipulación complejas y precisas (Chi et al., 2023).
2. Planificación y control: Mejoran la planificación de movimientos y el control adaptativo en tiempo real, abordando la variabilidad en las dinámicas del robot y las tareas (Ajay et al., 2022).
3. Aprendizaje por imitación: Permiten el aprendizaje de comportamientos complejos a partir de demostraciones humanas, capturando la multimodalidad del comportamiento humano y garantizando la consistencia temporal en las secuencias de acciones generadas (Li et al., 2023b).

Ejemplos de implementación:

1. Diffuser: Utilizado para generar trayectorias óptimas mediante el muestreo guiado por clasificadores en conjuntos de datos offline, demostrando mejoras en tareas de manipulación complejas (Janner et al., 2022).
2. Diffusion-QL: Combina políticas de difusión con el marco de aprendizaje Q, ajustándose perfectamente a conjuntos de datos recolectados por políticas de comportamiento multimodal fuertes, donde los enfoques anteriores fallaban (Wang et al., 2023).
3. SkillDiffuser: Empleado en el aprendizaje multitarea jerárquico, permite la generación de habilidades diversas y útiles para múltiples tareas de manipulación (Liang et al., 2023b).

La integración de modelos de difusión en la robótica de brazos con 6 grados de libertad está demostrando pues ser una estrategia efectiva para mejorar la generación de trayectorias, la planificación y el control en tareas de manipulación complejas. A medida que esta tecnología continúa evolucionando, se espera que su aplicación en la robótica siga expandiéndose, abordando desafíos adicionales y mejorando la eficiencia y la precisión de los robots en diversas tareas.

Una vez se obtiene la información de movimiento 3D, es necesario implementar algoritmos que extraigan la cinemática y dinámica del mecanismo para poder accionarlo de siguiendo la trayectoria definida por los modelos de difusión.

2.2. Cinemática de brazos robóticos

La cinemática de brazos robóticos se refiere al estudio del movimiento de los eslabones y las articulaciones del robot sin considerar las fuerzas y los momentos que causan este movimiento. Existen dos tipos principales de problemas cinemáticos: la cinemática directa y la cinemática inversa (Craig, 2005; Siciliano et al., 2010).

La cinemática directa (o forward kinematics) se encarga de determinar la posición y orientación del extremo efector del brazo robótico dados los valores de las variables articulares (ángulos de las articulaciones o desplazamientos lineales). Este proceso implica la construcción de una cadena cinemática desde la base hasta el extremo efector, utilizando las transformaciones homogéneas para describir la posición y orientación de cada eslabón en función de las variables articulares (Murray, Li, & Sastry, 1994).

Las ecuaciones de la cinemática directa se derivan utilizando matrices de transformación homogénea, que incluyen la rotación y la traslación en un solo sistema de referencia. Para un brazo robótico con *n* articulaciones, la transformación homogénea desde el sistema de referencia de la base hasta el sistema de referencia del extremo efector se expresa como:

$$T = T_1 T_2 \dots T_n$$

donde T_i es la matriz de transformación homogénea para el *i*-ésimo eslabón, que se puede descomponer en una matriz de rotación R_i y un vector de traslación d_i (Craig, 2005):

$$T_i = \begin{pmatrix} R_i & d_i \\ 0 & 1 \end{pmatrix}$$

Cada matriz de transformación homogénea T_i se define en términos de los parámetros de Denavit-Hartenberg (D-H), que son:

- θ_i: ángulo de rotación alrededor del eje z_{i-1}
- d_i: desplazamiento a lo largo del eje z_{i-1}
- a_i : longitud del eslabón a lo largo del eje x_i
- α_i: ángulo de rotación alrededor del eje x_i

Usando estos parámetros, la matriz de transformación homogénea T_i se puede expresar como (Siciliano et al., 2010):

$$T_i = \begin{pmatrix} \cos\theta_i & -\sin\theta_i \cos\alpha_i & \sin\theta_i \sin\alpha_i & a_i \cos\theta_i \\ \sin\theta_i & \cos\theta_i \cos\alpha_i & -\cos\theta_i \sin\alpha_i & a_i \sin\theta_i \\ 0 & \sin\alpha_i & \cos\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Al multiplicar estas matrices para todas las articulaciones n, se obtiene la posición y orientación final del extremo efector del brazo robótico.

La cinemática inversa (o inverse kinematics) se encarga de determinar los valores de las variables articulares que corresponden a una posición y orientación deseadas del extremo efector. A diferencia de la cinemática directa, la cinemática inversa puede tener múltiples soluciones, soluciones infinitas o no tener solución, dependiendo de la configuración del robot y las restricciones impuestas (Craig, 2005).

Existen varios métodos para resolver problemas de cinemática inversa, incluyendo:

- Métodos analíticos: Se utilizan para robots con estructuras simples y un número reducido de grados de libertad. Estos métodos implican derivar ecuaciones algebraicas explícitas para las variables articulares.
- Métodos numéricos: Utilizan técnicas iterativas para aproximar las soluciones. Ejemplos de estos métodos incluyen el método de Newton-Raphson y el gradiente descendente. Estos métodos son útiles para robots con estructuras complejas y múltiples grados de libertad (Siciliano et al., 2010).
- Métodos geométricos: Aprovechan las propiedades geométricas del robot para encontrar soluciones a los problemas de cinemática inversa. Son especialmente útiles para robots con configuraciones planas o esféricas (Murray, Li, & Sastry, 1994).

2.3. Dinámica de brazos robóticos

En lo que respecta a la dinámica de brazos robóticos, esta estudia las fuerzas y momentos que causan el movimiento de los eslabones y articulaciones del robot. Se divide en dinámica directa e inversa (Craig, 2005).

La dinámica directa (o forward dynamics) se encarga de determinar la aceleración de las articulaciones dadas las fuerzas y momentos aplicados. Las ecuaciones de movimiento se derivan utilizando el principio de Newton-Euler o el método de Lagrange (Siciliano et al., 2010).

Las ecuaciones de movimiento de un brazo robótico con *n* articulaciones se pueden expresar como:

$$M(q)\ddot{q} + C(q,\dot{q})\dot{q} + G(q) = \tau$$

donde:

- M(q) es la matriz de inercia.
- C(q,q˙) es la matriz de Coriolis y fuerzas centrífugas.
- G(q) es el vector de fuerzas gravitatorias.
- τ es el vector de torques aplicados en las articulaciones.
- q, q˙ y q¨ son los vectores de posición, velocidad y aceleración de las articulaciones, respectivamente (Murray, Li, & Sastry, 1994).

La dinámica inversa (o inverse dynamics) se encarga de determinar las fuerzas y momentos necesarios para generar un movimiento deseado. Este problema se resuelve utilizando las mismas ecuaciones de movimiento que en la dinámica directa, pero conociendo las posiciones, velocidades y aceleraciones deseadas de las articulaciones (Craig, 2005).

El conocimiento de la dinámica inversa es fundamental para el diseño de controladores de robots. Los controladores basados en la dinámica inversa, como los controladores PD con compensación de gravedad o los controladores robustos, permiten generar movimientos precisos y estables (Siciliano et al., 2010).

2.4. La integración de la IA en la robótica

Para concluir, la robótica ha avanzado de manera significativa en las últimas décadas, impulsada por desarrollos tecnológicos en inteligencia artificial, sensores avanzados y capacidades de computación. La integración de estas tecnologías ha permitido que los robots no solo realicen tareas repetitivas, sino que también se adapten a entornos dinámicos y ejecuten actividades complejas con un alto grado de autonomía y precisión (Craig, 2005; Siciliano et al., 2010).

La inteligencia artificial, en particular, ha jugado un papel crucial en estos avances. Las técnicas avanzadas de IA, como el aprendizaje profundo, el aprendizaje por refuerzo y los modelos generativos, han permitido que los robots aprendan de sus experiencias y se adapten

a nuevas situaciones (Ho, Jain, & Abbeel, 2020; Song & Ermon, 2021). Estos avances en IA han permitido a los robots generar trayectorias y movimientos precisos a partir de descripciones textuales, lo que representa un avance significativo en la programación y control de robots (Austin et al., 2021; Kong et al., 2020).

La capacidad de los robots para generar movimientos precisos y adaptarse a entornos cambiantes tiene aplicaciones amplias tanto en la industria como en la investigación.

1. Industria automotriz y manufactura: En líneas de producción, los robots con brazos articulados pueden realizar tareas como soldadura, ensamblaje y pintura con una precisión inigualable. La integración de modelos de IA permite que estos robots se adapten rápidamente a nuevos modelos de productos sin necesidad de un reentrenamiento extensivo (Chi et al., 2023).
2. Salud y cirugía: En el ámbito médico, los robots quirúrgicos pueden beneficiarse de estos avances al realizar procedimientos delicados con mayor precisión y menor riesgo de error. La capacidad de generar movimientos precisos a partir de descripciones textuales puede facilitar la programación de procedimientos quirúrgicos personalizados (Ajay et al., 2023).
3. Logística y almacenes: En centros de distribución y almacenes, los robots pueden manejar una variedad de tareas, desde el picking de productos hasta la organización de inventarios. La adaptabilidad que proporcionan los modelos de IA permite a estos robots manejar diferentes tipos de productos y adaptarse a cambios en la disposición del almacén (He et al., 2023a).
4. Investigación y desarrollo: En laboratorios de investigación, los robots con capacidades avanzadas de generación de movimientos pueden asistir en experimentos que requieren alta precisión y repetibilidad. Además, pueden ser utilizados para explorar nuevas áreas de investigación en robótica y automatización (Liang et al., 2023b).

A pesar de estos avances, la integración de técnicas avanzadas de IA en la robótica se enfrenta a varios desafíos:

1. Seguridad y confiabilidad: Es crucial asegurar que los robots operen de manera segura en entornos con humanos. Los modelos de IA deben ser robustos y confiables para evitar errores que podrían causar daños (Wang et al., 2023).

2. Interacción hombre-robot: Mejorar la interacción entre humanos y robots sigue siendo un área de investigación activa. Los robots deben ser capaces de comprender y anticipar las acciones humanas para trabajar de manera colaborativa y eficiente (Xiao et al., 2019).
3. Adaptabilidad en tiempo real: Aunque los modelos de IA pueden generar movimientos precisos, la capacidad de adaptarse en tiempo real a cambios en el entorno sigue siendo un desafío importante. El desarrollo de algoritmos que permitan esta adaptabilidad en tiempo real es una dirección futura clave (Janner et al., 2022).
4. Computación y recursos: La implementación de modelos avanzados de IA en robots requiere una considerable capacidad de procesamiento y recursos. Optimizar estos modelos para funcionar en hardware con recursos limitados es un desafío técnico significativo (Schneuing et al., 2023).

3. DE TEXTO A ACCIÓN MEDIANTE BRAZO ROBÓTICO DE TRES GRADOS DE LIBERTAD

3.1. De texto a animación 3D mediante modelos de difusión

Actualmente, uno de los proyectos más destacados en la generación de animaciones 3D a partir de texto con modelos de difusión es *MotionDiffuse*. Este enfoque es particularmente útil para aplicaciones que requieren movimientos precisos y adaptables, como animaciones, simulaciones y control de robots.

Este proyecto se destaca por varias características clave:

1. Mapeo probabilístico: A diferencia de los métodos deterministas, *MotionDiffuse* introduce variaciones en cada paso de denoising, permitiendo una mayor diversidad y realismo en los movimientos generados.
2. Síntesis realista: El modelo es capaz de manejar distribuciones de datos complejas, generando secuencias de movimiento vívidas y detalladas.
3. Manipulación multinivel: Responde a instrucciones detalladas sobre partes específicas del cuerpo y permite la síntesis de movimientos de longitud arbitraria basados en prompts textuales que varían en el tiempo.

Estos avances permiten que *MotionDiffuse* supere a gran parte de los métodos de generación de movimiento existentes, ofreciendo mejoras significativas en la precisión y control de los movimientos generados.

Dado que vamos a buscar el control de una zona concreta del esqueleto humano, como es el brazo, y que es un proyecto que no requiere un entrenamiento o reentrenamiento, ni la definición de una arquitectura concreta que se deba entrenar, se alza como una opción ideal para estructurar la generación de las poses del brazo robótico para este proyecto.

Los pasos a seguir incluyen:

1. Explicación del funcionamiento del proyecto: Se describirá cómo *MotionDiffuse* genera movimientos a partir de descripciones textuales, destacando las ventajas de su enfoque basado en modelos de difusión.
2. Clonación del repositorio: Se clonará el repositorio de GitHub para obtener el código y los datos necesarios para la ejecución del modelo.

3. Configuración del entorno: Se instalarán las dependencias requeridas y se configurará el entorno de desarrollo.
4. Ejecución de inferencia: Se utilizarán varios prompts textuales para generar movimientos humanoides y se evaluará la calidad de los resultados.
5. Extracción de datos del brazo derecho: Se explicará cómo extraer los puntos de datos correspondientes al brazo derecho de las animaciones generadas.
6. Visualización de los datos: Se describirá cómo visualizar los movimientos del brazo derecho para evaluar su precisión y realismo.

Explicación del proceso de obtención de poses mediante MotionDiffuse

En el capítulo sobre la contextualización del presente documento, se definieron los modelos de difusión como una nueva arquitectura de generación de datos que se servía de la generación de ruido y posterior eliminación en diferentes pasos para conseguir un realismo y calidad de resultados mayores que el que hasta entonces había supuesto el estado del arte, las redes generativas adversarias.

En primer lugar, se introduce el texto descriptivo y, para que el modelo pueda trabajar con él, se codifica mediante el modelo CLIP, que obtiene el significado semántico del texto y es capaz de apuntar con él a un vector en un espacio vectorial que lo represente con respecto a un contexto general lingüístico.

Una vez codificado, el modelo principal del sistema, un transformer de movimiento, procesa el vector de la entrada de texto y genera una secuencia limpia de movimiento. Se escoge una arquitectura de tipo transformer, ya que es capaz de procesar todos los elementos de una secuencia de datos en paralelo, lo que la hace mucho más eficiente con respecto a redes de tratamiento de series temporales clásicas como las RNN, LSTM o GRU.

Da comienzo entonces el proceso de difusión gaussiana, en el que se añade ruido poco a poco a la pose humanoide limpia. En este punto se introducen los conceptos de *alfas* y *betas*. Estos dos parámetros ponderan la mezcla de la señal original, la pose real, y el ruido en cada paso. De esta manera, el modelo es capaz de aprender a recuperar la señal original de los datos con ruido.

Es aquí cuando entra en juego el proceso de *denoising*, o eliminación de ruido, en el que el modelo aplica funciones que van reduciéndolo progresivamente mediante el uso de dos funciones clave, *p_sample* y *p_sample_loop*, comparando la pose generada en cada paso con la original para asegurarse que se está eliminando correctamente. Este proceso se lleva a cabo durante un número determinado de iteraciones, que en el proyecto de *MotionDiffuse* están configurados de forma estándar como 1000 *steps*, tras el cual se obtiene la animación final fluida y limpia.

A continuación, se añade un esquema gráfico en el que se muestra el proceso previamente explicado para una mejor comprensión:

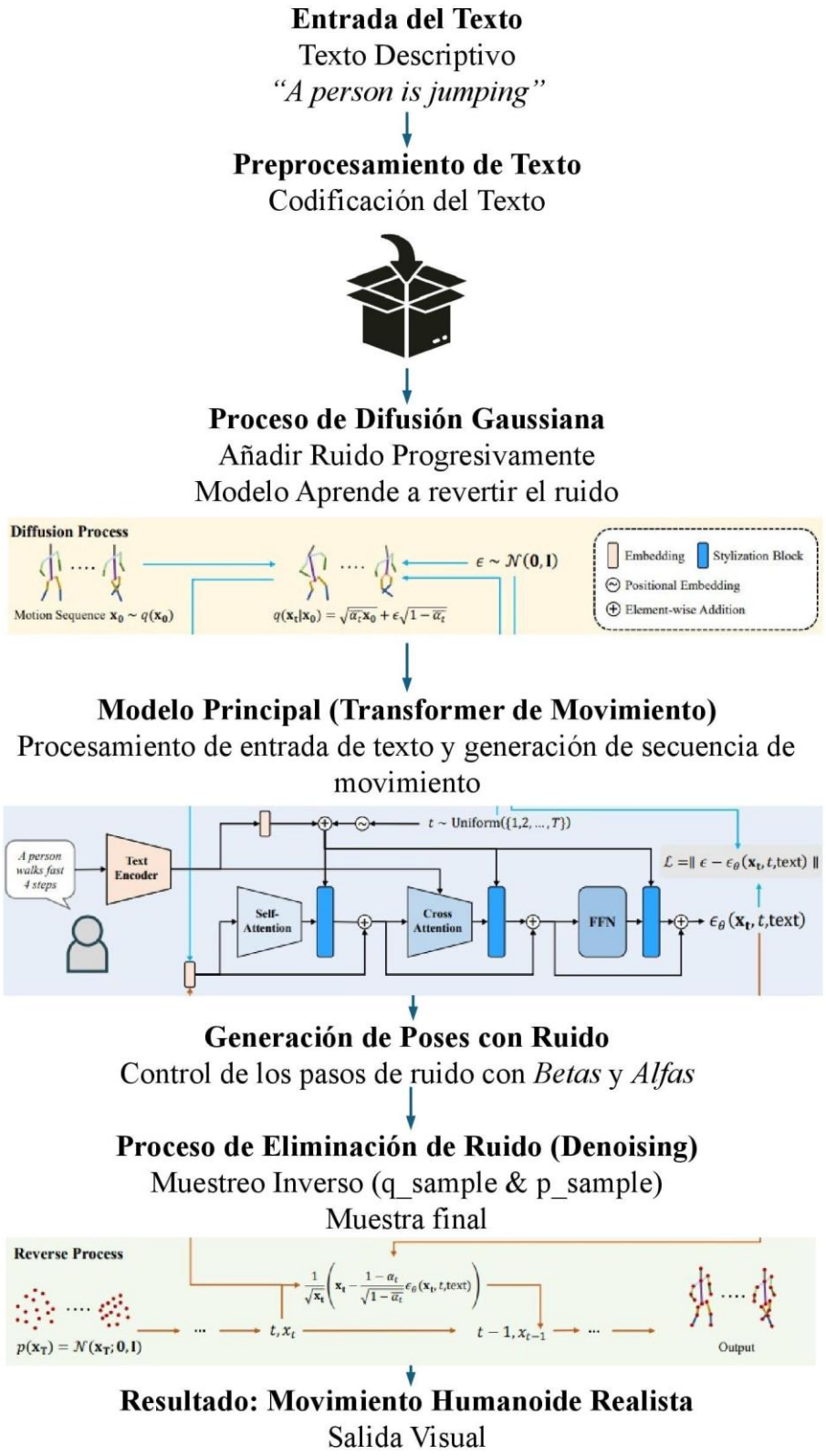

Figura 3. Esquema de la inferencia de *MotionDiffuse* para obtener una pose desde texto

Instalaciones Iniciales

Para empezar, es necesario instalar algunas dependencias clave:

```
!apt install ffmpeg
!pip install git+https://github.com/openai/CLIP.git
!pip install mmcv==1.7.1
!pip install matplotlib==3.3.1
```

Estas instalaciones proporcionan herramientas esenciales como *ffmpeg* para la manipulación de multimedia, la biblioteca *CLIP* de OpenAI para el procesamiento de texto, *mmcv* para el manejo de visión por computador y *matplotlib* para la visualización de datos.

Es especialmente relevante la instalación de la versión 1.7.1 para la biblioteca *mmcv*, ya que después de las actualizaciones que ha sufrido durante 2024 hacia la versión 2.2.0 muchos de los proyectos que hacían uso de ella han quedado inhabilitados.

Clonación del Proyecto

Luego, clonamos el repositorio del proyecto *MotionDiffuse* desde Google Drive y lo preparamos para su uso:

```
!gdown https://drive.google.com/uc?id=1vzBZ2rNCQWBQpYvC6hpyJfR3iK1O_FEG
!unzip MotionDiffuse.zip
!rm MotionDiffuse.zip
import os
os.chdir("MotionDiffuse")
```

Esto descarga y descomprime el proyecto en el entorno de trabajo, cambiando el directorio de trabajo actual a la carpeta del proyecto.

O alternativamente se puede clonar el repositorio oficial del proyecto en Github:

```
!git clone https://github.com/mingyuan-zhang/MotionDiffuse/tree/main
```

Imports Necesarios

A continuación, importamos las bibliotecas necesarias para el funcionamiento del proyecto:

```
import torch
import torch.nn.functional as F
import random
import time
from models.transformer import MotionTransformer
from torch.utils.data import DataLoader
import torch.optim as optim
from torch.nn.utils import clip_grad_norm_
from collections import OrderedDict
from utils.utils import print_current_loss
from os.path import join as pjoin
import codecs as cs
import torch.distributed as dist
from datasets import build_dataloader
import torch as th
import numpy as np
from utils.get_opt import get_opt
from utils.plot_script import *
import utils.paramUtil as paramUtil
from utils.motion_process import recover_from_ric
from mmcv.runner import get_dist_info
from models.gaussian_diffusion import (GaussianDiffusion,
get_named_beta_schedule, create_named_schedule_sampler, ModelMeanType,
ModelVarType, LossType)
```

La sección de carga de archivos y configuración de parámetros es fundamental para el correcto funcionamiento del modelo de difusión. A continuación, se detalla el propósito y funcionamiento de los componentes clave en este apartado:

1. Planificador de Betas

El parámetro `beta_scheduler` y el cálculo de las `betas` son esenciales para definir el proceso de difusión gaussiana. En el contexto de los modelos de difusión, las betas representan la tasa de ruido añadido en cada paso de la cadena de Markov inversa.

```
beta_scheduler = 'linear'
diffusion_steps = 1000
betas = get_named_beta_schedule(beta_scheduler, diffusion_steps)
```

- Planificador de betas (beta_scheduler): Determina cómo se distribuyen las betas a lo largo de los pasos de difusión. En este caso, se utiliza un planificador lineal

('linear'), lo que significa que el ruido añadido aumenta de manera lineal con cada paso.

- Pasos de difusión (diffusion_steps): Número total de pasos en el proceso de difusión. Aquí, se establecen 1000 pasos.
- Obtención de las betas (get_named_beta_schedule): Función que genera un *array* de valores beta basado en el planificador y los pasos especificados.

El *array* de betas define la cantidad de ruido que se introduce en cada paso de la cadena de difusión. Un planificador lineal asegura una introducción uniforme del ruido a lo largo de todos los pasos.

2. Configuración del Modelo de Difusión Gaussiana

El modelo de difusión gaussiana es el núcleo del proceso de generación de movimiento en *MotionDiffuse*. Se configura de la siguiente manera:

```
diffusion = GaussianDiffusion(
    betas=betas,
    model_mean_type=ModelMeanType.EPSILON,
    model_var_type=ModelVarType.FIXED_SMALL,
    loss_type=LossType.MSE
)
```

- Betas (betas): Los valores calculados anteriormente que determinan la cantidad de ruido en cada paso.
- Tipo de media del modelo (model_mean_type): Se establece como `EPSILON`, lo que significa que el modelo predice el ruido añadido en cada paso en lugar de directamente la señal original.
- Tipo de varianza del modelo (model_var_type): `FIXED_SMALL` indica que se utiliza una varianza fija pequeña en cada paso de difusión inversa.
- Tipo de pérdida (loss_type): `MSE` (Error Cuadrático Medio) se usa como la función de pérdida para entrenar el modelo, midiendo la diferencia entre las predicciones y los valores reales.

3. Betas y Alfas en el Proceso de Difusión

Los siguientes cálculos utilizan los valores de betas para obtener las alfas y otros parámetros necesarios para el proceso de muestreo:

```
betas = np.array(betas, dtype=np.float64)
alphas = 1.0 - betas
alphas_cumprod = np.cumprod(alphas, axis=0)
sqrt_alphas_cumprod = np.sqrt(alphas_cumprod)
sqrt_one_minus_alphas_cumprod = np.sqrt(1.0 - alphas_cumprod)
```

- Alfas (alphas): Se calculan como `1 - betas`, representando la cantidad de la señal original que se conserva en cada paso.
- Producto acumulativo de alfas (alphas_cumprod): Multiplicación acumulativa de los valores de alfas, lo que ayuda a determinar la evolución de la señal a lo largo de los pasos de difusión.
- Raíz cuadrada de alfas acumulativas (sqrt_alphas_cumprod): Utilizada en el proceso de muestreo para escalar la señal original.
- Raíz cuadrada de uno menos alfas acumulativas (sqrt_one_minus_alphas_cumprod): Utilizada para escalar el ruido añadido en cada paso.

Estos cálculos permiten controlar cómo la señal y el ruido se combinan a lo largo de la cadena de difusión, asegurando que la señal original pueda ser recuperada efectivamente durante el proceso de denoising.

4. Función q_sample

La función `q_sample` utiliza los valores de `sqrt_alphas_cumprod` y `sqrt_one_minus_alphas_cumprod` para generar una versión ruidosa de los datos de entrada en un paso específico de la difusión:

```
def q_sample(x_start, t, noise=None):
    if noise is None:
        noise = th.randn_like(x_start)
    assert noise.shape == x_start.shape
    return (
```

```
        _extract_into_tensor(sqrt_alphas_cumprod,  t,  x_start.shape)  *
x_start
        +     _extract_into_tensor(sqrt_one_minus_alphas_cumprod,     t,
x_start.shape) * noise
    )
```

- x_start: Los datos originales (en este caso, las poses humanoides).
- t: El paso de difusión actual.
- noise: Ruido añadido a los datos. Si no se proporciona, se genera aleatoriamente.

Esta función crea una representación ruidosa de los datos originales para un paso dado en la cadena de difusión, esencial para el proceso de denoising posterior.

MotionDiffuse: Text-Driven Human Motion Generation with Diffusion Model de Mingyuan Zhang, Zhongang Cai, Liang Pan, Fangzhou Hong, Xinying Guo, Lei Yang, y Ziwei Liu (2022) explica que los modelos de difusión aplicados a la generación de movimiento humano se basan en un proceso de eliminación de ruido que itera a través de pasos definidos por los parámetros `betas` y `alphas`. Este enfoque permite la generación de movimientos más realistas y variados en comparación con los métodos deterministas.

El uso de modelos de difusión gaussiana, como se describe en el *paper*, permite modelar distribuciones de datos complejas y producir secuencias de movimiento detalladas y naturales a partir de descripciones textuales. La configuración cuidadosa de los parámetros de difusión y las funciones asociadas asegura que el proceso de denoising pueda recuperar la estructura subyacente de los datos originales, logrando así la síntesis de movimientos de alta calidad.

Las funciones `p_sample_loop` y `p_sample` son fundamentales en el proceso de generación de muestras del modelo de difusión. Estas funciones implementan el proceso iterativo de eliminación de ruido que caracteriza a los modelos de difusión, generando nuevas muestras a partir de ruido aleatorio. A continuación, se proporciona una explicación detallada de cada función y su conexión con el proceso descrito en el paper.

La función `p_sample_loop` es responsable de la generación de muestras completas a través de un proceso iterativo de denoising. Esta función llama repetidamente a `p_sample` para cada paso en la cadena de difusión, generando así las muestras de manera progresiva.

```
def p_sample_loop(
```

```
    model,
    shape,
    noise=None,
    clip_denoised=True,
    denoised_fn=None,
    cond_fn=None,
    model_kwargs=None,
    device=None,
    pre_seq=None,
    transl_req=None,
    progress=False,
):
"""
Generate samples from the model.

:param model: the model module.
:param shape: the shape of the samples, (N, C, H, W).
:param noise: if specified, the noise from the encoder to sample.
              Should be of the same shape as `shape`.
:param clip_denoised: if True, clip x_start predictions to [-1, 1].
:param denoised_fn: if not None, a function which applies to the
    x_start prediction before it is used to sample.
:param cond_fn: if not None, this is a gradient function that acts
                similarly to the model.
:param model_kwargs: if not None, a dict of extra keyword arguments to
    pass to the model. This can be used for conditioning.
:param device: if specified, the device to create the samples on.
               If not specified, use a model parameter's device.
:param progress: if True, show a tqdm progress bar.
:return: a non-differentiable batch of samples.
"""
```

1. Inicialización:
    - Si se proporciona `noise`, se utiliza como la entrada inicial. De lo contrario, se genera ruido aleatorio con la forma especificada (`shape`).
2. Iteración a través de los pasos de difusión:
    - Se generan índices en orden inverso (`indices = list(range(num_timesteps))[::-1]`) para recorrer la cadena de difusión desde el ruido puro hasta la muestra final.

   - Si `progress` es *True*, se muestra una barra de progreso utilizando `tqdm`.
3. Llamada a `p_sample` en cada paso:
   - En cada paso, se llama a `p_sample` con los parámetros actuales, obteniendo una muestra intermedia (`sample`).
   - Se actualiza `img` con esta muestra para el siguiente paso.
4. Retorno de la muestra final:
   - Una vez completados todos los pasos, se retorna la muestra final generada.

La función `p_sample` se encarga de generar una muestra en un único paso de la cadena de difusión. Utiliza el modelo para predecir el estado de la muestra en el paso anterior y aplica ruido en consecuencia.

```
def p_sample(
    model,
    x,
    t,
    clip_denoised=True,
    denoised_fn=None,
    cond_fn=None,
    pre_seq=None,
    transl_req=None,
    model_kwargs=None,
):
    """
    Sample x_{t-1} from the model at the given timestep.

    :param model: the model to sample from.
    :param x: the current tensor at x_{t-1}.
    :param t: the value of t, starting at 0 for the first diffusion step.
    :param clip_denoised: if True, clip the x_start prediction to [-1, 1].
    :param denoised_fn: if not None, a function which applies to the
        x_start prediction before it is used to sample.
    :param cond_fn: if not None, this is a gradient function that acts
                    similarly to the model.
    :param model_kwargs: if not None, a dict of extra keyword arguments to
        pass to the model. This can be used for conditioning.
    :return: a dict containing the following keys:
             - 'sample': a random sample from the model.
```

```
            - 'pred_xstart': a prediction of x_0.
    """
```

1. Concatenación de secuencias:
    - Si `pre_seq` se proporciona, se ajusta `x` para incluir `pre_seq` en los pasos de difusión. Esto permite la generación condicional basada en una secuencia previa.
2. Aplicación de ruido:
    - Si `transl_req` se proporciona, se añade ruido adicional a las partes especificadas de `x`.
3. Predicción del modelo:
    - Se llama a `p_mean_variance` para obtener la media (`mean`) y varianza (`variance`) de la distribución posterior, así como la predicción de la muestra original (`pred_xstart`).
4. Aplicación de gradientes de condición:
    - Si `cond_fn` se proporciona, se ajusta la media utilizando los gradientes de la función de condición (`condition_mean`).
5. Generación de la muestra:
    - Se genera una nueva muestra añadiendo ruido a la media predicha y escalándola con la varianza.
6. Retorno:
    - Se retorna un diccionario con la nueva muestra generada y la predicción de la muestra original.

Según el paper, el proceso de denoising involucra la predicción de la señal original y la adición controlada de ruido en cada paso, lo que se implementa en las funciones `p_sample_loop` y `p_sample`.

- Proceso iterativo: La iteración a través de los pasos de difusión en `p_sample_loop` refleja el enfoque de generación progresiva descrito en el paper, donde el modelo refina gradualmente una muestra ruidosa hasta obtener una muestra realista.
- Predicción y ruido: La función `p_sample` realiza la predicción de la señal original (`pred_xstart`) y la combina con ruido escalado, siguiendo el principio de denoising iterativo que es central en los modelos de difusión.

El modelo principal es un Transformer de Movimiento, que se configura para aceptar características de entrada específicas y un número determinado de capas y dimensiones latentes.

```
def build_models(opt):
    encoder = MotionTransformer(
        input_feats=opt.dim_pose,
        num_frames=opt.max_motion_length,
        num_layers=opt.num_layers,
        latent_dim=opt.latent_dim,
        no_clip=opt.no_clip,
        no_eff=opt.no_eff
    )
    return encoder
```

El modelo se configura para trabajar con datos de un conjunto específico, en este caso, "t2m" (Text-to-Motion), y se aseguran ciertas condiciones como el número de cuadros y la raíz de los datos.

```
opt = get_opt('checkpoints/t2m/t2m_motiondiffuse/opt.txt', device)
opt.do_denoise = True

assert opt.dataset_name == "t2m"
assert 60 <= 196
opt.data_root = './dataset/HumanML3D'
opt.motion_dir = pjoin(opt.data_root, 'new_joint_vecs')
opt.text_dir = pjoin(opt.data_root, 'texts')
opt.joints_num = 22
opt.dim_pose = 263

mean = np.load(pjoin(opt.meta_dir, 'mean.npy'))
std = np.load(pjoin(opt.meta_dir, 'std.npy'))
```

Parámetros específicos:

- `dataset_name`: Asegura que el modelo trabaje con el conjunto de datos correcto.

- `data_root`, `motion_dir`, `text_dir`: Especifican las ubicaciones de los datos de movimiento y texto.
- `joints_num`, `dim_pose`: Definen la configuración de las articulaciones y la dimensión de la pose.

En lo que se refiere a la inferencia, analizamos la estructura del *Main* con un prompt cualquiera, como es *una persona salta*, o *a person is jumping*:

1. Contexto de `torch.no_grad()`

```
with torch.no_grad():
        caption = ["a person is jumping"]
        m_lens = torch.LongTensor([60]).to(device)
```

- Descripción: `torch.no_grad()` desactiva la autograd, lo cual es crucial durante la inferencia para ahorrar memoria y mejorar la velocidad, ya que no se necesitan gradientes.
- Uso de `caption` y `m_lens`: Define la entrada textual (*a person is jumping*) y la longitud de la secuencia de movimiento correspondiente.

2. Configuración del Batch

```
batch_size = 1024
N = len(caption)
cur_idx = 0
encoder.eval()
all_output = []
```

- batch_size: Define el tamaño del lote de datos que se procesará en cada iteración.
- N: Número total de captions que se van a procesar.
- cur_idx: Índice actual dentro del conjunto de datos.
- encoder.eval(): Configura el modelo en modo evaluación, desactivando dropout y batch normalization.
- all_output: Lista para almacenar las salidas generadas.

3. Procesamiento por Lotes

```
while cur_idx < N:
```

```
if cur_idx + batch_size >= N:
    batch_caption = caption[cur_idx:]
    batch_m_lens = m_lens[cur_idx:]
else:
    batch_caption = caption[cur_idx: cur_idx + batch_size]
    batch_m_lens = m_lens[cur_idx: cur_idx + batch_size]

caption = batch_caption
dim_pose = opt.dim_pose
m_lens = batch_m_lens
xf_proj, xf_out = encoder.encode_text(caption, device)
```

- División en lotes: Divide el conjunto de datos en lotes manejables basados en `batch_size`.
- Variables de lote: `batch_caption` y `batch_m_lens` son las sublistas actuales de captions y longitudes que se procesarán en esta iteración.
- Codificación de texto: `encoder.encode_text` convierte las captions en representaciones latentes utilizables por el modelo de difusión.

4. Generación de secuencias

```
B = len(caption)
T = min(m_lens.max(), encoder.num_frames)
output = p_sample_loop(
    encoder,
    (B, T, dim_pose),
    clip_denoised=False,
    progress=True,
    model_kwargs={
        'xf_proj': xf_proj,
        'xf_out': xf_out,
        'length': m_lens
    })
```

- Dimensiones del lote y tiempo: `B` es el número de ejemplos en el lote actual. `T` es la longitud máxima de secuencia considerada en este lote.
- Llamada a `p_sample_loop`: Esta función genera las secuencias de movimiento a partir de las representaciones latentes del texto. Se pasa el `encoder`, las dimensiones

del lote y pose, y otros argumentos clave como `model_kwargs` que incluyen las representaciones latentes y las longitudes de las secuencias.

5. Almacenamiento de salidas

```
B = output.shape[0]
for i in range(B):
    all_output.append(output[i])
cur_idx += batch_size
```

- Obtención del tamaño de lote: `B` se actualiza para reflejar el tamaño del lote generado.
- Almacenamiento: Las salidas generadas se almacenan en `all_output`.
- Actualización del índice: `cur_idx` se incrementa por el tamaño del lote, avanzando en el conjunto de datos.

Este proceso asegura que el modelo de difusión pueda transformar descripciones textuales en secuencias de movimiento humanoide de manera eficiente y escalable.

MotionDiffuse incluye todos estos procesos unificados en una misma ejecución, pero esta línea de investigación ha requerido diseccionar el código para obtener, además de las visualizaciones finales, las coordenadas de las poses específicas para cada frame y correlativas a cada articulación, puesto que solo es interesante en este caso el movimiento correspondiente a los puntos del brazo derecho respectivos a las articulaciones del brazo robótico que queremos mover. De esta forma, no basta únicamente con realizar la inferencia de la librería para obtener los datos que requerimos, sino que es necesario ir un paso para atrás para obtener del código exactamente las posiciones 3D a las que deberá moverse el brazo robótico.

Tras ejecutar el código para el prompt de salto, se obtienen los siguientes resultados para un esqueleto humanoide 3D:

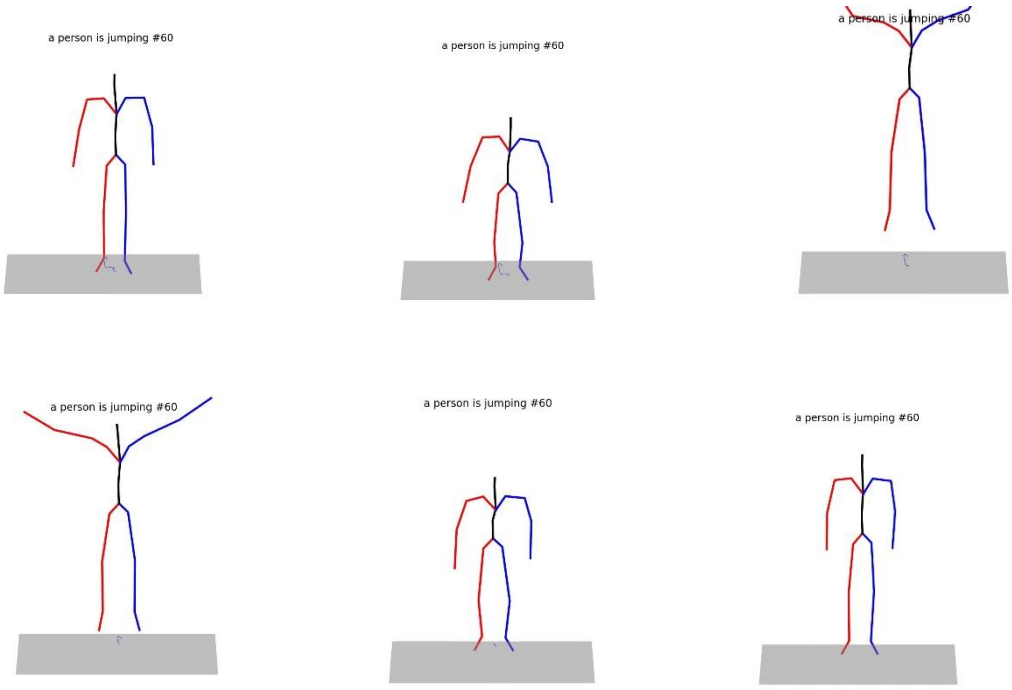

Figura 4. Secuencia de 6 poses de esqueleto humanoide 3D correspondientes al resultado de *MotionDiffuse* a la acción "una persona salta".

Se puede comprobar visualmente cómo el esqueleto realiza la acción demandada.

Si se desea extraer unos puntos concretos del esqueleto para realizar una integración concreta, en este caso con un brazo robótico de 6 grados de libertad, se tiene que estudiar la disposición de sus articulaciones para extraer aquellas correspondientes, por ejemplo, al brazo derecho.

En este caso, se busca realizar la asociación de las articulaciones del esqueleto que genera *MotionDiffuse* con el brazo robótico que se muestra en la Figura 5.

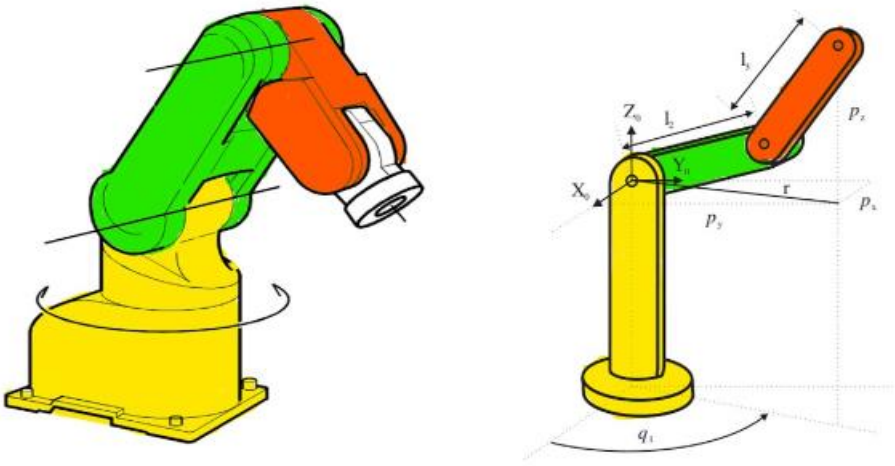

Figura 5. Brazo robótico de tres grados de libertad.

En ella, es posible identificar dos articulaciones fundamentales, la del hombro, compuesta por dos grados de libertad rotacionales, y la del codo, compuesta por otra rotacional. Aunque no es una articulación, se deberá considerar la posición del efector final, el extremo del brazo l_3, puesto que es una parte fundamental del brazo al ser el encargado de entrar en interacción con el elemento del entorno en cuestión dependiendo de la acción solicitada.

Así pues, se debe identificar dos articulaciones fundamentales en el esqueleto humanoide, hombro y codo, así como el efector final o extremo del antebrazo.

Esta información es posible encontrarla en el archivo *paramUtil.py* del proyecto, en el que se refleja la información de la cadena cinemática y los offsets necesarios para una correcta visualización del esqueleto.

Se encuentra que las articulaciones clave del brazo derecho del esqueleto humanoide son aquellas con los índices 9, 14, 17, 19 y 21 correspondientes al brazo representado en la Figura 6.

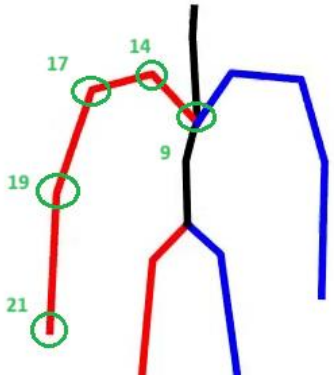

Figura 6. Asociación de índices de articulaciones del brazo derecho en el esqueleto generado por *MotionDiffuse*.

Como se puede apreciar, las tres últimas articulaciones corresponden a extremo del antebrazo, codo y hombro, y el esqueleto incluye otras dos para procurar la unión del brazo con la columna vertebral y darle movilidad adicional al hombro con otra articulación.

Se considerarán las dos articulaciones y el extremo definidos para el brazo de 3 grados de libertad aquellos con los índices 17, 19 y 21. Se muestra a continuación la serie de poses que toman estas articulaciones para el brazo derecho del esqueleto humanoide durante la animación de saltar.

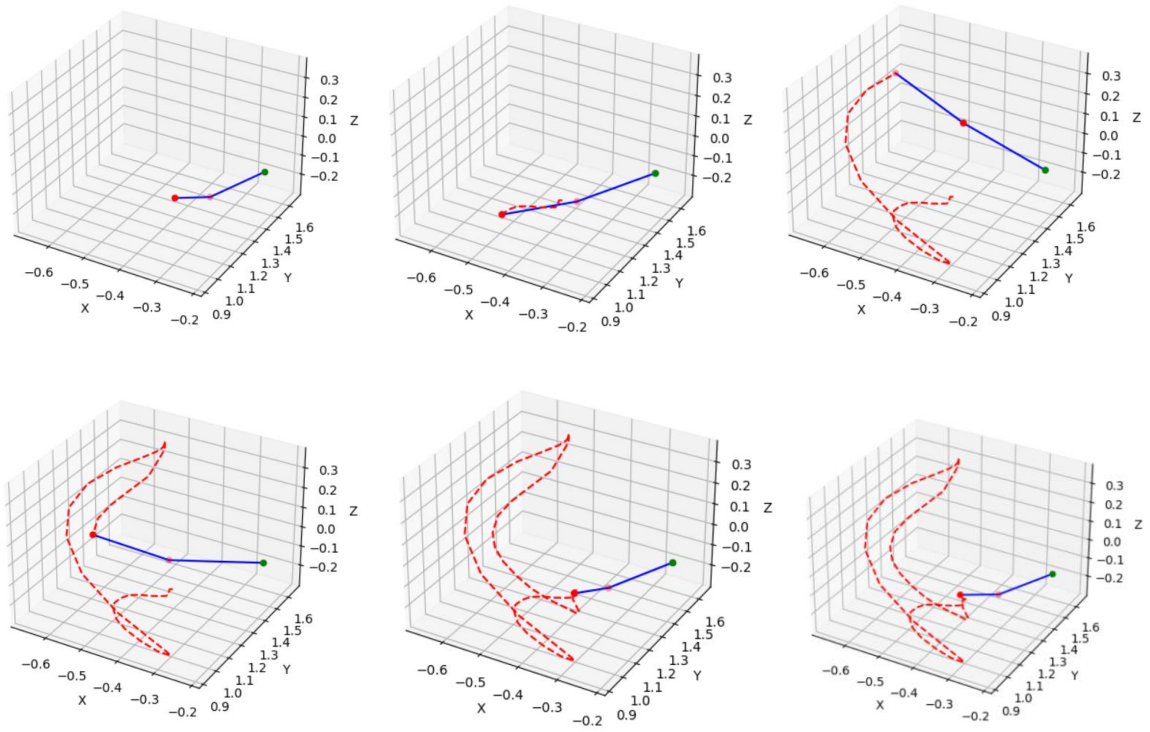

Figura 7. Secuencia de movimiento del brazo robótico generada por *MotionDiffuse* con 3 articulaciones principales para la acción "una persona salta".

El punto verde corresponde a la articulación que queda fijada en el origen de coordenadas del brazo robótico, la del hombro, en este caso el 17, mientras que quedan móviles las del codo y la muñeca, 19 y 21 respectivamente.

A nivel de código, guardamos como un array el archivo de *joints* generado para el ploteo del esqueleto y lo cargamos para obtener los índices adecuados:

```
import numpy as np

poses = np.load('…/element.py')

# Extraemos puntos clave del brazo derecho:

brazo_derecho_indices = [17, 19, 21]

brazo_derecho_poses = poses[:,brazo_derecho_indices,:]
```

Ajuste de Puntos del Brazo Derecho:

```
hombro_pos_fija = brazo_derecho_poses[0, 0, :]
```

```
for i in range(brazo_derecho_poses.shape[0]):
    desplazamiento = brazo_derecho_poses[i, 0, :] - hombro_pos_fija
    brazo_derecho_poses[i, :, :] -= desplazamiento
```

En la sucesión original extraída de poses humanoides, la posición del hombro es variable. No obstante, el punto correspondiente a dicha posición en el robot es fijo, por lo que debemos hacer que esta posición sea estática a lo largo de todos los frames en la secuencia de poses. Para ello, se calcula el desplazamiento del hombro en cada frame respecto a su posición en el primer frame y se ajustan las posiciones de todos los puntos del brazo en consecuencia.

Para ello, se crea la función *actualizar*, que plotea los puntos seleccionados del brazo derecho conectados por líneas y limpia el gráfico actualizándolo para la información de pose de cada frame:

```
def actualizar(num, brazo_derecho_poses, trayectorias, plot):

    ax.clear()

    # Graficamos el hombro en verde y los demás puntos del brazo en rojo

    puntos = brazo_derecho_poses[num]

       ax.scatter(puntos[0,  0],  puntos[0,  1],  puntos[0,  2],  c='g',
marker='o')  # Hombro (fijo)

      ax.scatter(puntos[1:,  0],  puntos[1:,  1],  puntos[1:,  2],  c='r',
marker='o')  # Brazo

    # Conectamos los puntos con líneas

    for i in range(len(puntos) - 1):

        ax.plot([puntos[i, 0], puntos[i+1, 0]],

                [puntos[i, 1], puntos[i+1, 1]],

                [puntos[i, 2], puntos[i+1, 2]], 'b-')
```

```
    # Graficamos la trayectoria del extremo del brazo (muñeca)

    trayectoria = np.array(trayectorias)

      ax.plot(trayectoria[:num+1,  2,  0],  trayectoria[:num+1,  2,  1],
trayectoria[:num+1, 2, 2], 'r--')

    # Fijamos los límites de los ejes para que sean constantes

    ax.set_xlim([limites_x[0], limites_x[1]])

    ax.set_ylim([limites_y[0], limites_y[1]])

    ax.set_zlim([limites_z[0], limites_z[1]])

    # Etiquetas de los ejes

    ax.set_xlabel('X')

    ax.set_ylabel('Y')

    ax.set_zlabel('Z')
```

Es posible representar la animación 3D de las poses seleccionadas del brazo derecho, tal y como se ha plasmado en la Figura 7, siguiendo los próximos pasos.

Se crea la función *animar_brazo_derecho* que genera figuras para cada frame. Esta llama a la función de actualización de cada uno de los frames:

```
def animar_brazo_derecho(brazo_derecho_poses):

    fig = plt.figure()

    global ax

    ax = fig.add_subplot(111, projection='3d')

    # Calculamos los límites fijos de los ejes

    global limites_x, limites_y, limites_z

          limites_x    =    (np.min(brazo_derecho_poses[:,    :,    0]),
np.max(brazo_derecho_poses[:, :, 0]))
```

```
        limites_y = (np.min(brazo_derecho_poses[:, :, 1]),
np.max(brazo_derecho_poses[:, :, 1]))

        limites_z = (np.min(brazo_derecho_poses[:, :, 2]),
np.max(brazo_derecho_poses[:, :, 2]))

    # Lista para almacenar las trayectorias del extremo del brazo (muñeca)

    trayectorias = brazo_derecho_poses[:, :, :]

    # Animamos los puntos del brazo derecho

        ani = animation.FuncAnimation(fig, actualizar,
frames=brazo_derecho_poses.shape[0],

                                      fargs=(brazo_derecho_poses, trayectorias,
None), interval=100)

    return ani
```

Por último, se ejecuta la animación:

```
ani = animar_brazo_derecho(brazo_derecho_poses)
HTML(ani.to_jshtml())
```

3.2. Obtención de la cinemática y dinámica del mecanismo

Una vez se ha obtenido la posición de las articulaciones del robot para cada instante de tiempo, se procede a la obtención de la cinemática inversa.

Para ello, elaboramos la matriz de Denavit-Hartenberg del brazo robótico de 3 grados de libertad que hemos presentado en el apartado previo. Utilizamos los parámetros de Denavit-Hartenberg para describir las relaciones entre los diferentes eslabones del robot. La tabla D-H define cuatro parámetros para cada eslabón: el ángulo de giro (θ), el desplazamiento a lo largo del eje z (d), la longitud del eslabón (a) y el ángulo de torsión (α).

Tabla de Denavit-Hartenberg del brazo robótico de 6 grados de libertad			
Θ	d	a	α
Θ_1 (rotación alrededor del eje Z_0) – Base	d_1 (altura de la base, l_1)	0	$\pi/2$
Θ_2 (rotación alrededor del eje Y_1) – Hombro	0	a_2 (longitud del brazo superior, l_2)	0
Θ_3 (rotación alrededor del eje Y_2) - Codo	0	a_3 (longitud del antebrazo, l_3)	0

Tabla 1. Tabla de Denavit-Hartenberg del brazo robótico de 3 grados de libertad.

Se elabora a continuación la matriz de transformación completa desde Θ_1 a Θ_3 multiplicando las matrices de transformación homogénea que se presentaron en el apartado 1.7. Contextualización, en el apartado sobre cinemática y dinámica.

Para ello, se han de definir las thetas iniciales, que podemos inicializar con valores aleatorios, y también las dimensiones de los eslabones. Los valores de a_2 y a_3 (l_2 y l_3) deben ser suficientes como para encajar en las distancias euclidianas entre los puntos 3D calculados mediante el citado modelo.

Las coordenadas 3D de las articulaciones podrían no conformar distancias constantes en el tiempo, lo que supondría un problema a nivel de estabilidad del sistema, por lo que nos disponemos a comprobar sus valores.

Tras medir las distancias para los 60 frames generados de la animación de saltar entre las articulaciones 17-19 y 19-21, obtenemos que:

Brazo	Medida media	Desviación estándar
17-19	0,261	0,003
19-21	0,266	0,007

Tabla 2. Medidas de los eslabones correspondientes al brazo robótico generado por *MotionDiffuse*.

Las medidas se encuentran en metros y el modelo de difusión entiende que son unas dimensiones similares para el brazo y el antebrazo del esqueleto. En el caso más conservador, las desviaciones estándar no llegan al centímetro, por lo que se puede considerar que el sistema es estable en cuanto a las dimensiones de los eslabones.

Si se decidiera tomar otras dimensiones sería posible escalar los puntos 3D generados por el modelo, y únicamente habría que calcular la relación final de distancias euclidianas entre ellos para volver a definir la longitud de los eslabones.

Se considerarán las dimensiones en metros estimadas por el modelo, se asignará un valor cualquiera para la altura de la base y se tomarán valores aleatorios para los tres ángulos theta iniciales.

Implementamos la matriz de transformación de Denavit-Hartenberg:

```
def matriz_DH(theta, d, a, alpha):
return np.array([ [np.cos(theta), -np.sin(theta) * np.cos(alpha), np.sin(theta) * np.sin(alpha), a * np.cos(theta)], [np.sin(theta), np.cos(theta) * np.cos(alpha), -np.cos(theta) * np.sin(alpha), a * np.sin(theta)], [0, np.sin(alpha), np.cos(alpha), d], [0, 0, 0, 1] ])
```

Generación de valores aleatorios para `theta` (valores entre 0 y 180 convertidos a radianes):

```
theta1, theta2, theta3 = np.radians(np.random.uniform(0, 180, 3))
```

Definimos la matriz de transformación completa:

```
def matriz_transformacion_completa(parametros_DH):
T = np.eye(4)
for param in parametros_DH:
      T = T @ matriz_DH(*param)
return T
T_completa = matriz_transformacion_completa(parametros_DH)
```

Se calcula la matriz de transformación completa multiplicando las matrices de transformación individuales de cada eslabón:

- `T = np.eye(4)`: Inicializa la matriz de transformación como una matriz identidad de 4x4.
- `for param in parametros_DH: T = T @ matriz_DH(*param)`: Para cada conjunto de parámetros D-H, calcula la matriz de transformación correspondiente y la multiplica con la matriz acumulada `T`.

Obtenemos la siguiente matriz de transformación completa para los *qrand* θ_1=0,837 radianes, θ_2=0,672 radianes y θ_3=1,484 radianes:

```
[[-3.69610358e-01 -5.58635114e-01  7.42505888e-01  3.85491767e-02]
 [-4.09706879e-01 -6.19237649e-01 -6.69839538e-01  4.27311155e-02]
 [ 8.33983487e-01 -5.51789401e-01  6.12323400e-17  7.84234028e-01]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  1.00000000e+00]]
```

Se calcula el jacobiano:

```
def calcular_jacobiano(thetas, parametros_DH):
      n = len(thetas)
      J = np.zeros((6, n))
      T = np.eye(4)
      z = [np.array([0, 0, 1])]
      p = [np.array([0, 0, 0])]
      for i, (theta, d, a, alpha) in enumerate(parametros_DH):
            T_i = matriz_DH(theta, d, a, alpha)
            T = T @ T_i
            z.append(T[:3, 2])
            p.append(T[:3, 3])
      p_end = p[-1]
      for i in range(n):
            J[:3, i] = np.cross(z[i], (p_end - p[i]))
            J[3:, i] = z[i]
      return J
```

Inicialización:

- `J` es la matriz Jacobiana de tamaño 6xN (donde N es el número de articulaciones).

- `T` es la matriz de transformación acumulada, comenzando como una matriz identidad de 4x4.
- `z` y `p` almacenan los vectores de dirección y las posiciones de cada eslabón respectivamente.

Cálculo del jacobiano:

- Se itera sobre los parámetros de Denavit-Hartenberg para calcular las matrices de transformación de cada eslabón.
- `T_i` es la matriz de transformación para el eslabón actual.
- `T` se actualiza multiplicando por `T_i` en cada iteración.
- `z` y `p` se actualizan para cada eslabón, almacenando el vector de dirección del eje z y la posición del extremo del eslabón.

Relleno de la matriz jacobiana:

- Para cada eslabón, se calcula la columna correspondiente de la matriz Jacobiana usando el producto vectorial y el vector de dirección del eje z.

Se construye la función que lleva a cabo la cinemática inversa:

```
def cinemática_inversa(pos_deseada, parametros_DH, thetas_iniciales, iteraciones=1000, tolerancia=1e-6):
      thetas = np.array(thetas_iniciales)
      for _ in range(iteraciones):
            parametros_DH_actualizados = [(thetas[i], d, a, alpha) for i, (theta, d, a, alpha) in enumerate(parametros_DH)]
            T_completa =
            matriz_transformacion_completa(parametros_DH_actualizados)
            pos_actual = T_completa[:3, 3]
            error = pos_deseada - pos_actual
            if np.linalg.norm(error) < tolerancia:
            break
            J = calcular_jacobiano(thetas, parametros_DH)
            d_thetas = np.linalg.pinv(J) @ np.concatenate((error, np.zeros(3))) thetas += d_thetas
      return thetas
```

Inicialización:

- Se inicializan los ángulos de las articulaciones (`thetas`) con los valores iniciales proporcionados.

Iteraciones:

- Para cada iteración, se actualizan los parámetros D-H con los ángulos actuales de las articulaciones.
- Se calcula la matriz de transformación completa `T_completa`.
- `pos_actual` se extrae como la posición del extremo efector.
- `error` se calcula como la diferencia entre la posición deseada y la posición actual del efector final.
- Si el error es menor que la tolerancia establecida, el bucle se rompe.
- Se calcula el Jacobiano `J` y se usa la pseudo-inversa de `J` para obtener el cambio necesario en los ángulos (`d_thetas`).
- Se actualizan los ángulos de las articulaciones con `d_thetas`.

Se está considerando nulo el error de orientación, de ahí el uso de np.zeros(3) en la actualización de los ángulos de las articulaciones con `d_thetas`. Esto es así porque se ha considerado en esta investigación que lo más importante es la estimación de la posición del efector final y no su orientación. Solamente se estima relevante que el efector llegue a su posición final cuando lleve a cabo una acción generada por el modelo de difusión, y la necesidad de una orientación u otra para la realización de alguna tarea específica cuando el efecto llega a dicha posición no se ha considerado relevante, puesto que no se analiza qué ocurre en la interacción con el entorno cuando se llega a tal posición. Ignorar este error en la orientación puede ayudar a acelerar el proceso de convergencia y simplificar el cálculo.

Retorno de los ángulos calculados:

- La función retorna los ángulos de las articulaciones que posicionan el efector final en la posición deseada.

Tomamos como ejemplo la posición del efector final para el primer frame de la animación del salto y valores cero para todos los ángulos iniciales:

```
parametros_DH = [
    (theta1, d1, 0, np.pi/2),  # Articulación 1 (Base)
    (theta2, 0, a2, 0),        # Articulación 2 (Hombro)
```

```
    (theta3, 0, a3, 0)          # Articulación 3 (Codo)
]


pos_deseada = np.array([-2.29124621e-01, 8.78161132e-01, 1.36496544e-01])
thetas_iniciales = np.radians([0, 0, 0]) # Ángulos iniciales de las
articulaciones
thetas_solucion = cinemática_inversa(pos_deseada, parametros_DH,
thetas_iniciales)
```

Obteniendo los siguientes valores de theta:

```
thetas_finales = [   2129.37640921 -102357.93510246  100328.14428826]
```

Estos valores corresponden a la posición del efector final definida por el modelo de difusión, pero no tienen en cuenta las posiciones definidas para el codo y el hombro.

Esto se ilustra perfectamente en la comparativa entre el brazo robótico y la pose humanoide generada por *MotionDiffuse* de la Figura 8. El efector final se corresponde ya con la posición de la muñeca de la pose humanoide, pero no así la posición del codo y del hombro.

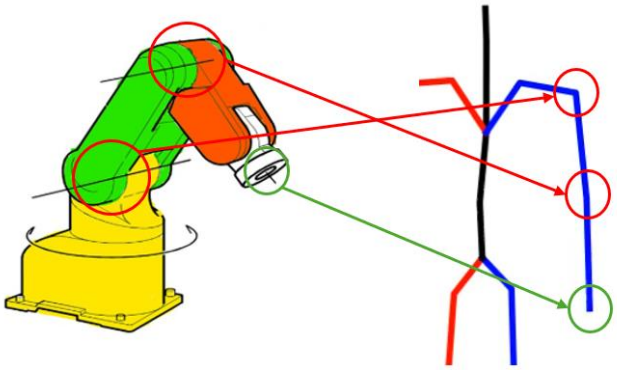

Figura 8. Correlación entre efector final y muñeca, existente tras el proceso de cinemática inversa, así como de codos y hombros, que requiere de un proceso múltiple.

Para ello, es necesario implementar una variación al método de Newton-Raphson que considere dichas posiciones y que permita aproximar los extremos de los eslabones del mecanismo minimizando una función de error mediante perturbaciones en los ángulos iniciales. Principalmente se introducirán variaciones en la función de cinemática inversa,

que pasará a ser múltiple, y se integrará una función de evaluación de las soluciones para asegurar la convergencia a una solución óptima.

La función `cinemática_inversa_múltiple` constituye la nueva función de cálculo de los ángulos articulares necesarios para que un brazo robótico alcance una posición deseada del efector final. Explora múltiples configuraciones iniciales para encontrar posibles soluciones a través de pequeñas perturbaciones de los ángulos iniciales.

1. Inicialización de parámetros y perturbaciones

   ```
   perturbaciones = np.linspace(-np.pi/8, np.pi/8, 5)
   ```

   Se generan pequeñas perturbaciones alrededor de los ángulos iniciales en el rango de $-\pi/8$ a $\pi/8$. Estas perturbaciones permiten explorar múltiples configuraciones iniciales, incrementando la probabilidad de encontrar una solución factible.

2. Iteración sobre perturbaciones

   ```
   for pert in perturbaciones:
       thetas = np.array(thetas_iniciales) + pert
   ```

   Para cada perturbación, se ajustan los ángulos iniciales añadiendo la perturbación actual.

3. Ciclo de iteraciones para ajuste de ángulos

   ```
   for _ in range(iteraciones):
       parametros_DH_actualizados = [(thetas[i], d, a, alpha) for i, (theta, d, a, alpha) in enumerate(parametros_DH)]
       T_completa = matriz_transformacion_completa(parametros_DH_actualizados)

       pos_actual = T_completa[:3, 3]
       error = pos_deseada - pos_actual

       if np.linalg.norm(error) < tolerancia:
           break

       J = calcular_jacobiano(thetas, parametros_DH)
       d_thetas = np.linalg.pinv(J) @ np.concatenate((error, np.zeros(3)))
   ```

```
    thetas += d_thetas
```

En cada iteración:

- Se actualizan los parámetros de Denavit-Hartenberg con los ángulos actuales.
- Se calcula la matriz de transformación completa para obtener la posición actual del efector final.
- Se calcula el error entre la posición actual y la deseada.
- Si el error es menor que la tolerancia, se termina la iteración.
- Se calcula la matriz Jacobiana y se obtiene la corrección de los ángulos (d_thetas) usando la pseudo-inversa.
- Se actualizan los ángulos con la corrección obtenida.

4. Almacenamiento de soluciones

```
soluciones.append(thetas)
```

Al finalizar las iteraciones para una perturbación, la solución encontrada (los ángulos articulares) se almacena en la lista de soluciones.

La función `evaluar_soluciones` selecciona la mejor solución de entre las múltiples generadas por `cinemática_inversa_múltiple` evaluando el error total respecto a posiciones conocidas de las articulaciones.

1. Inicialización de variables

```
mejor_solucion = None
menor_error = np.inf
```

Se inicializan `mejor_solucion` y `menor_error` para almacenar la mejor solución y el error mínimo encontrado.

2. Iteración sobre soluciones

```
for thetas in soluciones:
    parametros_DH_actualizados = [(thetas[i], d, a, alpha) for i, (theta, d, a, alpha) in enumerate(parametros_DH)]
    T_completa = matriz_transformacion_completa(parametros_DH_actualizados)
    pos_articulaciones = []
    T = np.eye(4)
    for param in parametros_DH_actualizados:
```

```
            T = T @ matriz_DH(*param)
            pos_articulaciones.append(T[:3, 3])
```

Para cada solución en la lista de soluciones:

- Se actualizan los parámetros D-H con los ángulos actuales.
- Se calcula la matriz de transformación completa.
- Se obtienen las posiciones de todas las articulaciones del brazo.

3. Cálculo del error

```
error = np.sum([np.linalg.norm(pos - pos_conocida) for pos, pos_conocida in zip(pos_articulaciones, pos_articulaciones_conocidas)])
```

Se calcula el error total como la suma de las distancias (normas) entre las posiciones calculadas y las posiciones conocidas de las articulaciones.

4. Selección de la mejor solución

```
if error < menor_error:
    menor_error = error
    mejor_solucion = thetas
```

Si el error actual es menor que el error mínimo encontrado hasta el momento, se actualizan `menor_error` y `mejor_solucion` con los valores actuales.

5. Retorno de la mejor solución

```
return mejor_solucion, menor_error
```

Finalmente, se retorna la mejor solución encontrada y el error asociado.

Si corremos el código para las posiciones del brazo en el primer frame obtenemos los siguientes resultados:

```
Mejor solución (en grados): [     6950.96253145 -133149.90512392 135275.40105831]
Error mínimo: 2.513432626024171
```

A continuación, se plantea el cálculo de la dinámica del brazo robótico. Para ello, será necesario el cálculo de las velocidades y aceleraciones de los ángulos *theta* de los eslabones en cada instante de tiempo.

Se deberá considerar igualmente el diferencial de tiempo entre cada frame de posiciones. En este caso la configuración de la inferencia de *MotionDiffuse* prepara la generación para 60 frames por segundo, por lo que dicho diferencial será 1/60.

En Python, el submódulo diff de numpy permite calcular las velocidades y aceleraciones angulares utilizando diferencias finitas:

```
theta_dot = np.diff(thetas, axis=0)/dt
```

```
theta_ddot = np.diff(theta_dot, axis=0)/dt
```

Además, se agregan filas de ceros al principio para igualar las dimensiones:

```
theta_dot = np.vstack([np.zeros((1, thetas.shape[1])), theta_dot])
```

```
theta_ddot = np.vstack([np.zeros((2, thetas.shape[1])), theta_ddot])
```

Debemos además asignar masas e inercias a los diferentes eslabones:

```
# Asignar valores aleatorios para las masas e inercia
np.random.seed(42)  # Para reproducibilidad
masas = np.random.uniform(1, 10, 3)  # Masas aleatorias entre 1 y 10 kg
inercia = [np.random.uniform(0.01, 0.1, 3) for _ in range(3)]
```

Tomamos un ejemplo de velocidades y aceleraciones angulares. Por ejemplo, las obtenidas en el instante 1:

```
theta_dot = [1.88502242e+05, -1.53984732e+01, -9.42441375e+04]
theta_ddot = [0., 0., 0.]
```

Definimos las velocidades y aceleraciones iniciales:

```
v = [np.zeros(3)] # Velocidades lineales
omega = [np.zeros(3)] # Velocidades angulares
v_dot = [np.zeros(3)] # Aceleraciones lineales
omega_dot = [np.zeros(3)] # Aceleraciones angulares
```

Para cada eslabón, calculamos las velocidades y aceleraciones basándonos en las transformaciones obtenidas y las velocidades/aceleraciones del eslabón anterior.

1. Rotación (`R`): La matriz de rotación del eslabón actual.
2. Traslación (`P`): El vector de traslación del eslabón actual.

```
R = T_matrices[i][:3, :3] # Matriz de rotación del eslabón i
P = T_matrices[i][:3, 3] # Vector de traslación del eslabón i
```

La velocidad angular del eslabón actual se calcula como la suma de la velocidad angular del eslabón anterior (transformada a través de la matriz de rotación `R`) y la velocidad angular debida a la rotación del eslabón actual.

```
omega_i = R.T @ omega[-1] + np.array([0, 0, theta_dot[i]])
```

La velocidad lineal del eslabón actual se calcula utilizando la velocidad lineal del eslabón anterior, la velocidad angular del eslabón anterior, y el vector de traslación `P`. Esto incluye el efecto de la rotación (`np.cross(omega[-1], P)`).

```
v_i = R.T @ (v[-1] + np.cross(omega[-1], P))
```

La aceleración angular del eslabón actual se calcula como la suma de la aceleración angular del eslabón anterior (transformada a través de `R`), el efecto de la velocidad angular en la rotación, y la aceleración angular debido a la rotación del eslabón actual.

```
omega_dot_i = R.T @ (omega_dot[-1] + np.cross(omega[-1], np.array([0, 0,
theta_dot[i]]))) + np.array([0, 0, theta_ddot[i]])
```

La aceleración lineal del eslabón actual se calcula utilizando la aceleración lineal del eslabón anterior, la aceleración angular del eslabón anterior, la velocidad angular del eslabón anterior, y el vector de traslación `P`. Esto incluye los efectos de la rotación y las aceleraciones angulares.

```
v_dot_i = R.T @ (v_dot[-1] + np.cross(omega_dot[-1], P) + np.cross(omega[-
1], np.cross(omega[-1], P)))
```

Como último paso, se calculan las fuerzas y torques. Este paso utiliza las ecuaciones de Newton-Euler para calcular las fuerzas y torques que actúan sobre cada eslabón del robot, empezando desde el extremo (último eslabón) y trabajando hacia la base (primer eslabón).

1. Fuerza (`F`)**:**
    - Se calcula como la masa del eslabón multiplicada por su aceleración lineal menos la gravedad.
2. Torque (`N`)**:**
    - Se calcula usando el tensor de inercia del eslabón y sus aceleraciones/velocidades angulares.

3. Transmisión de fuerzas y torques**:**
    - Si no estamos en el último eslabón, sumamos las contribuciones de las fuerzas y torques transmitidos a través del siguiente eslabón.

Lo integramos dentro de un mismo código:

```
fuerzas = []
torques = []
for i in reversed(range(len(parametros_DH))):
      # Calculamos la fuerza en el eslabón actual
      F = masas[i] * (v_dot[i + 1] - np.array([0, 0, 9.81])) # Gravedad
      hacia abajo

      # Calculamos el torque en el eslabón actual
      N = inercia[i] * omega_dot[i + 1] + np.cross(omega[i + 1], inercia[i]
      * omega[i + 1])

      # Si no es el último eslabón, sumamos las contribuciones de los
      eslabones siguientes
      if i < len(parametros_DH) - 1:
            R = T_matrices[i + 1][:3, :3]
            P = T_matrices[i + 1][:3, 3]

            # Sumar la fuerza transmitida desde el siguiente eslabón
            F += R @ fuerzas[-1]
            # Sumar el torque transmitido desde el siguiente eslabón
            N += R @ (torques[-1] + np.cross(P, fuerzas[-1]))
      # Añadir la fuerza y el torque calculados para el eslabón actual a
      las listas correspondientes
      fuerzas.append(F)
      torques.append(N)
# Invertir las listas para tener las fuerzas y torques en el orden correcto
fuerzas.reverse()
torques.reverse()
```

Para las velocidades y aceleraciones angulares definidas, obtenemos las siguientes fuerzas y torques en cada eslabón:

```
Eslabón 1:
Fuerza: [-9.63439616 -1.99814857 -1.80228318]
Torque: [ 9.80661756 -4.58383383 -1.68027604]
Eslabón 2:
Fuerza: [ 9.50070075  2.30361808 -2.20627669]
Torque: [-1.49413953  1.11796533  1.23252262]
Eslabón 3:
Fuerza: [-1.41204564  1.70029364  2.33424099]
Torque: [-5.33125552 -5.47989754  7.09446282]
```

Las fuerzas y torques vienen en unidades del S.I., en este caso N y Nm.

Como la búsqueda es de las fuerzas y torques para toda la animación 3D del movimiento del brazo, integramos las funciones de generación de velocidades y aceleraciones angulares para todos los instantes de tiempo junto con las del cálculo de fuerzas y torques.

El siguiente código muestra dicha fusión:

```
# Asignar valores aleatorios para las masas e inercia

np.random.seed(42)  # Para reproducibilidad

masas = np.random.uniform(1, 10, 3)  # Masas aleatorias entre 1 y 10 kg

inercia = [np.random.uniform(0.01, 0.1, 3) for _ in range(3)]  # Momentos
de inercia aleatorios

# Inicializar listas para almacenar fuerzas y torques en cada instante

todas_fuerzas = []

todos_torques = []

# Iterar sobre todos los instantes de tiempo

for i in range(len(thetas)):

    thetas_inst = thetas[i]

    theta_dot_inst = theta_dot[i]

    theta_ddot_inst = theta_ddot[i]
```

```
    parametros_DH = [(thetas_inst[j], d, a, alpha) for j, (theta, d, a,
alpha) in enumerate(parametros_DH_base)]

    # Calcular las matrices de transformación completa para cada eslabón

    T_matrices = matriz_transformacion_completa(parametros_DH)

    # Inicialización de las velocidades y aceleraciones

    v = [np.zeros(3)]  # Velocidades lineales

    omega = [np.zeros(3)]  # Velocidades angulares

    v_dot = [np.zeros(3)]  # Aceleraciones lineales

    omega_dot = [np.zeros(3)]  # Aceleraciones angulares

    # Calcular las velocidades y aceleraciones de cada eslabón

    for j in range(len(parametros_DH)):

        R = T_matrices[j][:3, :3]

        P = T_matrices[j][:3, 3]

        omega_j = R.T @ omega[-1] + np.array([0, 0, theta_dot_inst[j]])

        v_j = R.T @ (v[-1] + np.cross(omega[-1], P))



        omega_dot_j = R.T @ (omega_dot[-1] + np.cross(omega[-1],
np.array([0, 0, theta_dot_inst[j]]))) + np.array([0, 0,
theta_ddot_inst[j]])

        v_dot_j = R.T @ (v_dot[-1] + np.cross(omega_dot[-1], P) +
np.cross(omega[-1], np.cross(omega[-1], P)))

        omega.append(omega_j)

        v.append(v_j)

        omega_dot.append(omega_dot_j)
```

```
        v_dot.append(v_dot_j)

    # Calcular las fuerzas y torques usando las ecuaciones de Newton-Euler

    fuerzas = []

    torques = []

    for j in reversed(range(len(parametros_DH))):

        F = masas[j] * (v_dot[j + 1] - np.array([0, 0, 9.81]))  # Gravedad
hacia abajo

        N = inercia[j] * omega_dot[j + 1] + np.cross(omega[j + 1],
inercia[j] * omega[j + 1])

        if j < len(parametros_DH) - 1:

            R = T_matrices[j + 1][:3, :3]

            P = T_matrices[j + 1][:3, 3]

            F += R @ fuerzas[-1]

            N += R @ (torques[-1] + np.cross(P, fuerzas[-1]))

        fuerzas.append(F)

        torques.append(N)

    fuerzas.reverse()

    torques.reverse()

    todas_fuerzas.append(fuerzas)

    todos_torques.append(torques)
```

Los resultados de este código son análogos a los anteriores para cada frame de movimiento 3D del brazo robótico.

3.3. Diseño del robot

Conocidos los valores de cinemática y dinámica a partir del movimiento generado del prompt de texto es posible realizar el diseño motor del robot, es decir, la selección de aquellos actuadores que permitan abordar las características físicas asociadas a cada eslabón del brazo.

Los motores se seleccionan en función de los siguientes parámetros:

· Par de Torsión (torque): La capacidad del motor para generar el torque requerido en cada articulación.

· Velocidad Angular: La velocidad a la que el motor puede operar de manera efectiva.

· Potencia: La capacidad del motor para generar potencia suficiente para mover la carga.

· Precisión y Control: Algunos motores tienen mejor precisión y capacidad de control que otros.

El principal parámetro obtenido del algoritmo en el que se basará entonces la selección del actuador es el torque. Es posible vincular al sistema un catálogo comercial, del que seleccionará para cada eslabón un motor en función de que posibilite un torque mayor del torque máximo calculado para cualquier instante del movimiento, de forma que se maneje un margen de seguridad ante el fallo.

Un ejemplo de catálogo comercial integrable en el código es:

```
motores_catalogo = [ {'modelo': 'Motor A', 'torque_max': 5.0, 'velocidad_max': 3000, 'potencia': 100}, {'modelo': 'Motor B', 'torque_max': 10.0, 'velocidad_max': 2500, 'potencia': 200}, {'modelo': 'Motor C', 'torque_max': 15.0, 'velocidad_max': 2000, 'potencia': 300}, {'modelo': 'Motor D', 'torque_max': 20.0, 'velocidad_max': 1500, 'potencia': 400},

# ... agregar más motores según sea necesario

]
```

Se desarrolla una función que evalúe los torques máximos calculados en la sección dinámica para realizar la selección en el catálogo. Como ejemplos de catálogos comerciales, Maxon Motors, Kollmorgen, Oriental Motor, Faulhaber o ABB son proveedores de catálogos

detallados y herramientas en línea para el apoyo en la selección de los actuadores en función de los parámetros definidos.

El citado margen de seguridad es relevante para garantizar el funcionamiento fiable y duradero del brazo robótico. El cálculo o estimación del mismo depende de diversas variables, la acción a realizar o variabilidad de la misma, en el caso de aplicaciones como la propuesta en el presente documento, la frecuencia en que esta se va a realizar, sobre qué objetos, con qué propiedades, qué masas y momentos de inercia se van a seleccionar en los eslabones, los ciclos de fatiga del mecanismo, etc.

Esta propuesta se centrará únicamente en la consideración del torque, para conectar directamente con la aplicación dinámica que proviene del movimiento 3D generado, pero, además, es importante considerar factores clave dependientes de la aplicación específica de estos sistemas, como el tamaño, el peso o la eficiencia del motor, así como el controlador que se plantea utilizar.

Así pues, se presenta el código de puesta en práctica de la selección del motor en función de catálogo, o código de diseño de actuadores del brazo robótico:

```
# Ejemplo de catálogo de motores

motores_catalogo = [

    {'modelo': 'Motor A', 'torque_max': 5.0, 'velocidad_max': 3000, 'potencia': 100},

    {'modelo': 'Motor B', 'torque_max': 10.0, 'velocidad_max': 2500, 'potencia': 200},

    {'modelo': 'Motor C', 'torque_max': 15.0, 'velocidad_max': 2000, 'potencia': 300},

    {'modelo': 'Motor D', 'torque_max': 20.0, 'velocidad_max': 1500, 'potencia': 400},

    # ... agregar más motores según sea necesario

]

# Requisitos de torque máximos calculados para cada articulación
```

```
torques_maximos = [8.0, 12.0, 5.0] # Ejemplo de torques máximos en Nm

# Función para seleccionar el motor adecuado

def seleccionar_motor(torque_requerido, motores_catalogo):

    for motor in motores_catalogo:

        if motor['torque_max'] >= torque_requerido:

            return motor

    return None

# Seleccionar motores para cada articulación

motores_seleccionados = []

for torque in torques_maximos:

    motor = seleccionar_motor(torque, motores_catalogo)

    motores_seleccionados.append(motor)

# Imprimir los motores seleccionados

for i, motor in enumerate(motores_seleccionados):

    if motor:

        print(f"Articulación {i+1}: {motor['modelo']} (Torque Máximo: {motor['torque_max']} Nm, Velocidad Máxima: {motor['velocidad_max']} RPM, Potencia: {motor['potencia']} W)")

    else:

        print(f"Articulación {i+1}: No se encontró un motor adecuado")
```

4. PROPUESTA DE DISEÑO Y CONTROL COMPLETA

En los anexos de este documento se presenta el código completo para la generación end-to-end a partir de texto de diseños dinámicos de brazo robótico de tres grados de libertad.

En el siguiente enlace a Google Colaboratory es posible correrlo con seguridad, siguiendo todas las indicaciones: https://colab.research.google.com/drive/1paYC279iyIRhyFHjooyxAatuqbSTAOFf?usp=sharing.

En el presente apartado se reflejan tres de los experimentos realizados a partir de texto para validar el movimiento 3D generado por el modelo de difusión *MotionDiffuse*, así como las propiedades de los eslabones y el diseño de sus actuadores en función del análisis cinemático y dinámico.

Experimento 1: Queremos diseñar un brazo robótico que sea capaz de realizar un movimiento para coger un vaso.

El prompt elegido para tal acción es: “A person takes the glass”.

En este caso, el modelo de difusión genera la animación de coger con la mano izquierda, no con la derecha. Se ha testeado especificando el tipo de mano con el que se desea que realice la acción, pero en muchos casos el modelo no está preparado para recibir muchas especificaciones, por lo que, si se comprueba que es necesario realizar la simulación con la mano izquierda, simplemente en el código es necesario cambiar los índices correspondientes a las articulaciones de muñeca, codo y hombro, que en tal caso serían 16, 18 y 20.

Se simula el movimiento del brazo y se obtienen los siguientes frames representativos:

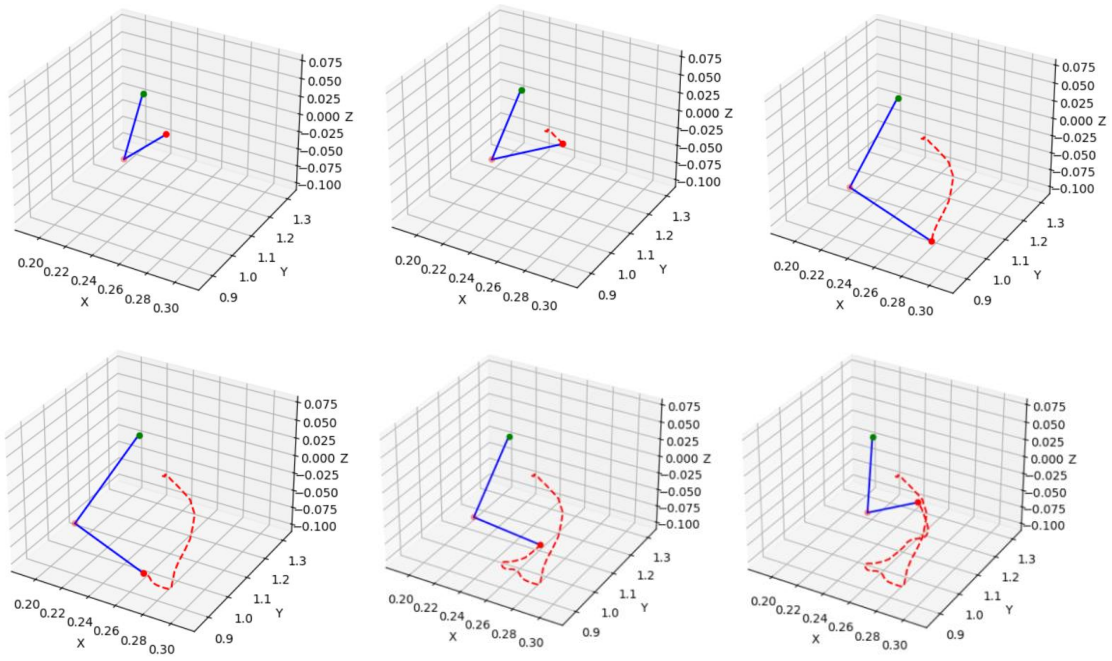

Figura 9. Secuencia de movimiento del brazo robótico generada por *MotionDiffuse* con 3 articulaciones principales para la acción "una persona coge el vaso".

En ellos es posible verificar cómo la articulación de la muñeca, cercana al efector final que agarraría el objeto, se extiende y se retrae en un movimiento que permite alcanzar una zona frente a él y recoger el objeto. El vaso quedaría en el último frame pegado al efector final, al extremo rojo del brazo.

Nótese que en este punto podría añadirse información del entorno para converger la planificación de movimiento que provee el modelo de difusión y la posición exacta del elemento en cuestión. Esta aproximación podría llevarse a cabo de forma paralela, es decir, integrar en el sistema un modelo de visión artificial capaz de identificar el objeto vaso y su posición el espacio y que esta información permitiera modificar en tiempo real los resultados del modelo de difusión, o podría realizarse desde el mismo entrenamiento del modelo. Para este segundo caso sí sería necesario reentrenar *MotionDiffuse* con propiedades de los elementos en interacción del texto o realizar un desarrollo de cero de arquitectura de difusión que contemple esta información además del dataset de movimiento 3D.

Si se ejecuta el resto del sistema se obtiene el siguiente diseño de actuadores:

```
Articulación 1: Motor F (Torque Máximo: 37.0 Nm, Velocidad Máxima: 1000
RPM, Potencia: 600 W)
Articulación 2: Motor F (Torque Máximo: 37.0 Nm, Velocidad Máxima: 1000
RPM, Potencia: 600 W)
```

```
Articulación 3: Motor E (Torque Máximo: 30.0 Nm, Velocidad Máxima: 1300
RPM, Potencia: 500 W)
```

Es posible comprobar que las primeras articulaciones requieren el mayor torque y potencia para el movimiento del resto del mecanismo según el movimiento trazado. A su vez, no requieren de una velocidad tan grande como la del efector final y las articulaciones más cercanas a este.

Experimento 2: Queremos diseñar un brazo robótico que vierta el contenido del vaso a otro recipiente.

El prompt elegido para tal acción es: “A person pours the contents of the glass”.

En este caso la animación vuelve a desarrollarse con la mano izquierda, por lo que se respetan los índices establecidos para el primer experimento.

Se simula el movimiento del brazo y se obtienen los siguientes frames representativos:

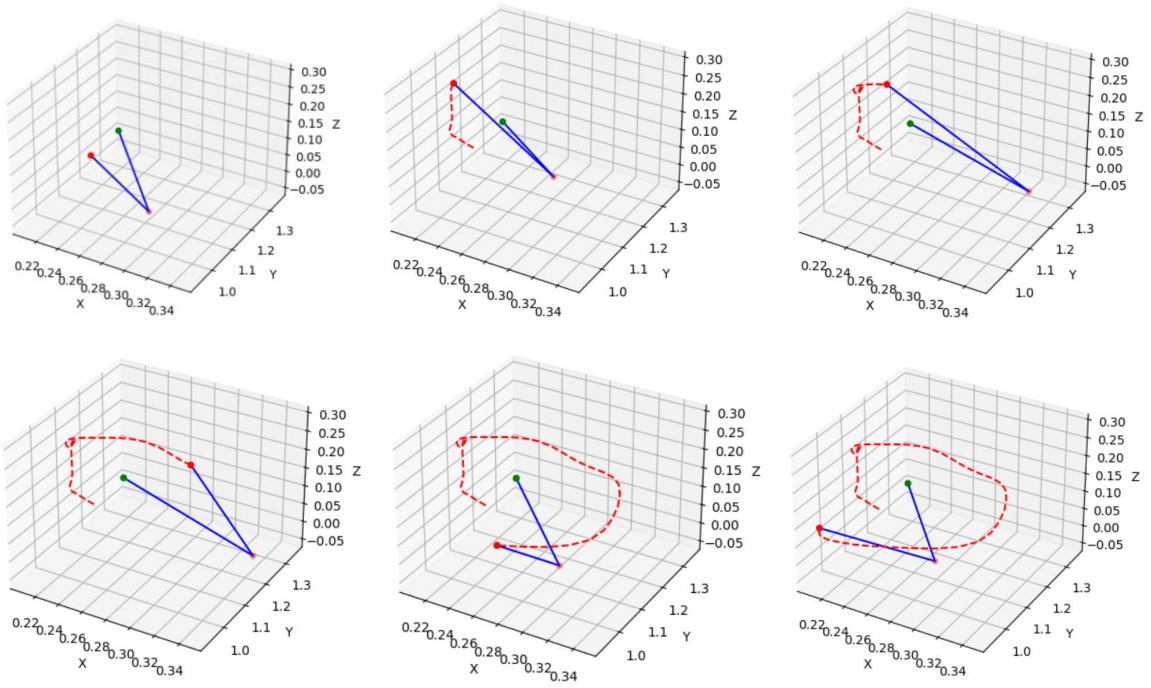

Figura 10. Secuencia de movimiento del brazo robótico generada por *MotionDiffuse* con 3 articulaciones principales para la acción “una persona vierte el contenido del vaso”.

Es perceptible la pequeña curva que dibuja el extremo en el aire y se respresenta en la tercera captura, cómo el codo se aleja de la muñeca, ocupando un espacio a la izquierda de la escena y cómo esa muñeca baja de nuevo para dejar el vaso en el lugar donde lo ha tomado.

Si se ejecuta el resto del sistema se obtiene el siguiente diseño de actuadores:

```
Articulación 1: Motor E (Torque Máximo: 30.0 Nm, Velocidad Máxima: 1300
RPM, Potencia: 500 W)
Articulación 2: Motor E (Torque Máximo: 30.0 Nm, Velocidad Máxima: 1300
RPM, Potencia: 500 W)
Articulación 3: Motor E (Torque Máximo: 30.0 Nm, Velocidad Máxima: 1300
RPM, Potencia: 500 W)
```

Vemos cómo de nuevo hay una disposición descendente de las necesidades de potencia desde las articulaciones primeras a las últimas, y que el vertido no requiere tanta energía como la toma del vaso.

Experimento 3: Queremos diseñar un brazo robótico que lleve el recipiente a otro lugar.

El prompt elegido para tal acción es: “A person transports the container.”.

En este caso, la acción se desarrolla con ambas manos en el enfoque de salida del modelo de difusión, pero es posible considerar una única y diseñar una configuración determinada en el efector final que permita mantener el equilibrio durante el agarre y el transporte.

También es posible plantear este sistema para un par de brazos robóticos que puedan trabajar de forma colaborativa en la consecución de tareas que puedan requerirlo, como el caso de esta acción.

Se simula el movimiento del brazo y se obtienen los siguientes frames representativos:

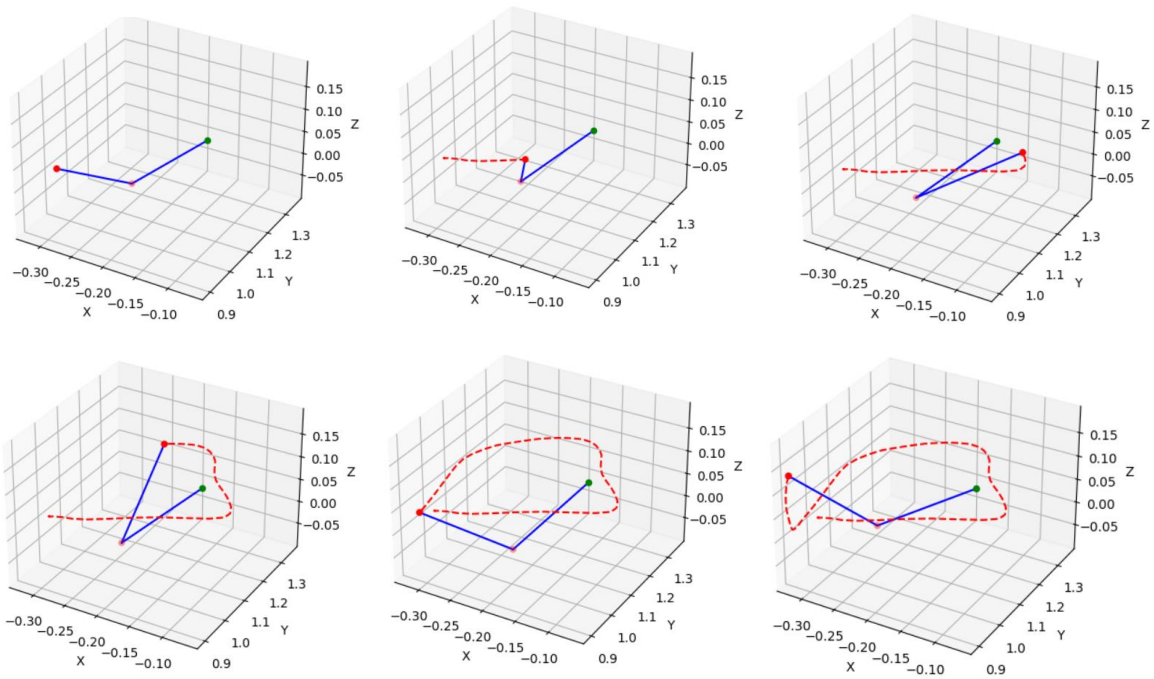

Figura 11. Secuencia de movimiento del brazo robótico generada por *MotionDiffuse* con 3 articulaciones principales para la acción "una persona transporta el recipiente".

Efectivamente el brazo desplegado toma desde su extremo derecho una curva con el codo para para llevar el objeto hacia su posición de destino, especialmente a partir de la segunda representación. En la tercera captura llega al punto final donde deja el objeto y es en las siguientes donde se visualiza el regreso del efector final a la posición original.

Si se ejecuta el resto del sistema se obtiene el siguiente diseño de actuadores:

```
Articulación 1: Motor F (Torque Máximo: 37.0 Nm, Velocidad Máxima: 1000
RPM, Potencia: 600 W)
Articulación 2: Motor E (Torque Máximo: 30.0 Nm, Velocidad Máxima: 1300
RPM, Potencia: 500 W)
Articulación 3: Motor E (Torque Máximo: 30.0 Nm, Velocidad Máxima: 1300
RPM, Potencia: 500 W)
```

La potencia necesaria vuelve a alcanzar cotas como durante el proceso de captación del vaso.

5. CONCLUSIONES Y PRÓXIMOS PASOS

Ante los objetivos planteados en esta investigación, se han elaborado las siguientes conclusiones:

Objetivo Principal

Integrar un modelo de difusión para la generación de poses 3D de un brazo robótico con 6 grados de libertad a partir de prompts de texto.

Se ha validado exitosamente que es posible llevar a cabo una implementación para la generación de movimiento 3D a partir de texto. Este logro se ha extendido al cálculo cinemático-dinámico, lo que ha permitido el diseño de los actuadores necesarios para la tarea del robot. La integración de modelos de difusión ha demostrado ser efectiva en traducir descripciones textuales en secuencias de poses precisas para un brazo robótico, abriendo nuevas posibilidades para la programación y control de robots mediante comandos de texto.

Objetivos Secundarios

1. Adaptar un modelo de difusión especializado en la generación de poses 3D humanas, enfocándose exclusivamente en las poses de un brazo y su traducción al control de un brazo robótico.

El modelo de difusión inicialmente diseñado para la generación de poses humanas ha sido exitosamente adaptado para enfocarse exclusivamente en las poses de un brazo robótico. Este proceso ha involucrado la personalización del modelo para capturar las especificidades del movimiento del brazo, lo que ha permitido una traducción precisa y eficiente de las poses generadas al control del brazo robótico. Esta adaptación ha demostrado la flexibilidad del modelo de difusión para aplicaciones más especializadas en robótica.

2. Desarrollar el cálculo preciso de la cinemática del brazo robótico, abordando tanto la cinemática directa como inversa.

Se ha desarrollado y validado un método preciso para el cálculo de la cinemática inversa del brazo robótico. Este método ha permitido determinar los ángulos de las articulaciones a partir de las posiciones y orientaciones del brazo en el espacio tridimensional para alcanzar una posición deseada. La precisión del cálculo cinemático es fundamental para asegurar el correcto funcionamiento y control del brazo robótico en tareas complejas.

3. Realizar un análisis dinámico del brazo robótico para simular las fuerzas y torques necesarios en la ejecución de tareas, incluyendo la selección de motores adecuados.

El análisis dinámico realizado ha permitido simular las fuerzas y torques que el brazo robótico necesita para ejecutar diferentes tareas. Este análisis ha sido crucial para seleccionar los motores adecuados que puedan manejar las cargas y movimientos requeridos. La simulación dinámica ha proporcionado una comprensión detallada de los requisitos de actuación, lo que ha facilitado la selección de componentes y la optimización del diseño del brazo robótico.

4. Implementar una serie de simulaciones y pruebas prácticas para validar la efectividad y precisión del sistema desarrollado en la generación de poses y su impacto en la operatividad del brazo robótico.

Se han llevado a cabo tres simulaciones que han validado la efectividad y precisión del sistema desarrollado. Estas pruebas han confirmado que el brazo robótico puede reproducir fielmente las poses generadas por el modelo de difusión, y que las estrategias de control basadas en el cálculo cinemático-dinámico son efectivas en escenarios reales. La validación práctica ha demostrado que el sistema es robusto y puede ser aplicado en diversas tareas robóticas.

5. Proporcionar una herramienta versátil y adaptativa para la programación y control de robots mediante comandos de texto, facilitando su uso en diversas aplicaciones industriales y de investigación.

El desarrollo de una herramienta que permite la programación y control de robots mediante comandos de texto ha sido un logro significativo de esta investigación. Esta herramienta se ha mostrado versátil y adaptativa, facilitando su uso en una amplia gama de aplicaciones industriales y de investigación. La simplicidad y efectividad de los comandos textuales han reducido la complejidad de la programación robótica, haciendo esta tecnología más accesible.

Futuras Líneas de Investigación:

Como próximos pasos, es esencial avanzar hacia la integración de sistemas de realimentación en tiempo real para mejorar la adaptabilidad y precisión del brazo robótico en su entorno de trabajo. Se necesitaría:

- Realimentación de imagen: Implementar sistemas de visión que permitan al robot interpretar y reaccionar ante cambios en el entorno.

- Realimentación de peso y contacto: Incorporar sensores de fuerza y tacto para que el robot pueda manejar objetos con mayor precisión y detectar obstáculos imprevistos.
- Sistemas de orientación del movimiento: Desarrollar algoritmos que permitan ajustar la trayectoria del brazo robótico en función de las características detectadas en su entorno, optimizando así la ejecución de tareas en tiempo real.

Esta investigación proporciona una base sólida para la planificación inicial del movimiento de robots, la cual se puede mejorar con sistemas de realimentación y orientación para adaptarse a las condiciones dinámicas del entorno.

6. BIBLIOGRAFÍA

- Austin, J., Johnson, D. D., Ho, J., Tarlow, D., & van den Berg, R. (2021). *Structured Denoising Diffusion Models in Discrete State-Spaces*. NeurIPS 2021 Poster.
- Craig, J. J. (2005). *Introduction to Robotics: Mechanics and Control*. Pearson Prentice Hall.
- Murray, R. M., Li, Z., & Sastry, S. S. (1994). *A Mathematical Introduction to Robotic Manipulation*. CRC Press.
- Schneuing, A., Du, Y., Harris, C., Jamasb, A. R., Igashov, I., Du, W., Blundell, T. L., Lio, P., Gomes, C. P., Welling, M., Bronstein, M. M., & Correia, B. (2023). *Structure-based Drug Design with Equivariant Diffusion Models*. Submitted to ICLR 2023.
- Siciliano, B., Sciavicco, L., Villani, L., & Oriolo, G. (2010). *Robotics: Modelling, Planning and Control*. Springer.
- Song, Y., & Ermon, S. (2021). *Score-Based Generative Modeling through Stochastic Differential Equations*. Proceedings of the International Conference on Learning Representations.

7. WEBGRAFÍA

- Ajay, A., et al. (2022). *Is Conditional Generative Modeling all you need for Decision-Making?*. Disponible en: https://arxiv.org/abs/2211.15657 [Consulta: 20 de enero de 2024].
- Chi, A., et al. (2023). *Diffusion Policy: Visuomotor Policy Learning via Action Diffusion*. Disponible en: https://arxiv.org/abs/2303.04137 [Consulta: 20 de enero de 2024].
- He, H., Zhu, Z., Zhao, H., Zhong, Y., Zhang, S., Guo, H., Chen, T., & Zhang, W. (2024). *Diffusion Models for Reinforcement Learning: A Survey*. Disponible en: https://arxiv.org/html/2311.01223v4 [Consulta: 20 de enero de 2024].
- Ho, J., Jain, A., & Abbeel, P. (2020). *Denoising Diffusion Probabilistic Models*. Disponible en: https://arxiv.org/abs/2006.11239 [Consulta: 20 de enero de 2024].
- Janner, M., et al. (2022). *Planning with diffusion: Flexible trajectory generation via iterative denoising*. Disponible en: https://arxiv.org/abs/2205.09991 [Consulta: 20 de enero de 2024].
- Kong, Z., et al. (2020). *DiffWave: A Versatile Diffusion Model for Audio Synthesis*. Disponible en: https://arxiv.org/abs/2009.09761 [Consulta: 20 de enero de 2024].
- Liang, J., et al. (2023). *SkillDiffuser: Interpretable Hierarchical Planning via Skill Abstractions in Diffusion-Based Task Execution*. Disponible en: https://arxiv.org/abs/2312.11598 [Consulta: 20 de enero de 2024].
- Schneuing, A., et al. (2023). *Structure-based Drug Design with Equivariant Diffusion Models*. Disponible en: https://arxiv.org/abs/2210.13695 [Consulta: 20 de enero de 2024].
- Wang, Z., et al. (2023). *Diffusion Policies as an Expressive Policy Class for Offline Reinforcement Learning*. Disponible en: https://arxiv.org/abs/2208.06193 [Consulta: 20 de enero de 2024].
- Xiao, C., et al. (2019). *Learning to Combat Compounding-Error in Model-Based Reinforcement Learning*. Disponible en: https://arxiv.org/abs/1912.11206 [Consulta: 20 de enero de 2024].

8. ÍNDICE DE ILUSTRACIONES

9. ÍNDICE DE TABLAS

10. ANEXO

```
import os
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from IPython.display import HTML

## Si ya hemos cargado todo o no:
ya_preparado = True

if not ya_preparado:
  # Instalación de librerías

  # Installs
  os.system('apt install ffmpeg')
  os.system('pip install git+https://github.com/openai/CLIP.git')
  os.system('pip install mmcv==1.7.1')
  os.system('pip install matplotlib==3.3.1')

  # Clonación del proyecto
  os.system('gdown
https://drive.google.com/uc?id=1vzBZ2rNCQWBQpYvC6hpyJfR3iK1O_FEG')
  os.system('unzip MotionDiffuse.zip')
  os.system('rm MotionDiffuse.zip')
  os.chdir("MotionDiffuse")

  # Nos colocamos en la carpeta raíz
  os.chdir("/content/MotionDiffuse")

# Ejecutamos el comando principal para la generación de animación 3D
humanoide a partir de texto
os.environ["PYTHONPATH"] = ".:$PYTHONPATH"
os.system('python -u tools/visualization.py --opt_path
checkpoints/t2m/t2m_motiondiffuse/opt.txt --text "One person
transports the container." --motion_length 60 --result_path
"test_sample.mp4"')

# Importamos la librería necesaria para cargar el array de poses
generado
array_cargado = np.load('/content/element.npy')
poses = array_cargado

# Extraemos los puntos del brazo derecho
brazo_derecho_indices = [17, 19, 21]
brazo_derecho_poses = poses[:, brazo_derecho_indices, :]
```

```
# Tomamos la posición del hombro en el primer frame como la posición
fija
hombro_pos_fija = brazo_derecho_poses[0, 0, :]

# Ajustamos todas las posiciones de los puntos del brazo para que el
hombro sea estático
for i in range(brazo_derecho_poses.shape[0]):
    desplazamiento = brazo_derecho_poses[i, 0, :] - hombro_pos_fija
    brazo_derecho_poses[i, :, :] -= desplazamiento

# Función para actualizar los frames de la animación
def actualizar(num, brazo_derecho_poses, plot):
    ax.clear()
    puntos = brazo_derecho_poses[num]

    # Graficamos los puntos
    ax.scatter(puntos[0, 0], puntos[0, 1], puntos[0, 2], c='g',
marker='o')
    ax.scatter(puntos[1:, 0], puntos[1:, 1], puntos[1:, 2], c='r',
marker='o')

    # Conectamos los puntos con líneas
    for i in range(len(puntos) - 1):
        ax.plot([puntos[i, 0], puntos[i+1, 0]],
                [puntos[i, 1], puntos[i+1, 1]],
                [puntos[i, 2], puntos[i+1, 2]], 'b-')

    ax.set_xlabel('X Label')
    ax.set_ylabel('Y Label')
    ax.set_zlabel('Z Label')

# Función para animar los puntos del brazo derecho
def animar_brazo_derecho(brazo_derecho_poses):
    fig = plt.figure()
    global ax
    ax = fig.add_subplot(111, projection='3d')

    ani = animation.FuncAnimation(fig, actualizar,
frames=brazo_derecho_poses.shape[0],
                                  fargs=(brazo_derecho_poses, None),
interval=100)
    return ani

# Animamos los puntos del brazo derecho
ani = animar_brazo_derecho(brazo_derecho_poses)
HTML(ani.to_jshtml())

### Cálculo de cinemática inversa y dinámica
```

```
def matriz_DH(theta, d, a, alpha):
    return np.array([
        [np.cos(theta), -np.sin(theta) * np.cos(alpha), np.sin(theta)
* np.sin(alpha), a * np.cos(theta)],
        [np.sin(theta), np.cos(theta) * np.cos(alpha), -np.cos(theta)
* np.sin(alpha), a * np.sin(theta)],
        [0, np.sin(alpha), np.cos(alpha), d],
        [0, 0, 0, 1]
    ])

def matriz_transformacion_completa(parametros_DH):
    T = np.eye(4)
    for param in parametros_DH:
        T = T @ matriz_DH(*param)
    return T

def matriz_transformacion_completa2(parametros_DH):
    T = np.eye(4)
    matrices = []
    for param in parametros_DH:
        T = T @ matriz_DH(*param)
        matrices.append(T)
    return matrices

def calcular_jacobiano(thetas, parametros_DH):
    n = len(thetas)
    J = np.zeros((6, n))
    T = np.eye(4)

    z = [np.array([0, 0, 1])]
    p = [np.array([0, 0, 0])]

    for i, (theta, d, a, alpha) in enumerate(parametros_DH):
        T_i = matriz_DH(theta, d, a, alpha)
        T = T @ T_i
        z.append(T[:3, 2])
        p.append(T[:3, 3])

    p_end = p[-1]

    for i in range(n):
        J[:3, i] = np.cross(z[i], (p_end - p[i]))
        J[3:, i] = z[i]

    return J

def cinemática_inversa_múltiple(pos_deseada, parametros_DH,
thetas_iniciales, iteraciones=1000, tolerancia=1e-6):
```

```
    soluciones = []

    perturbaciones = np.linspace(-np.pi/8, np.pi/8, 5)  # Pequeñas
perturbaciones alrededor de los ángulos iniciales

    for pert in perturbaciones:
        thetas = np.array(thetas_iniciales) + pert

        for _ in range(iteraciones):
            parametros_DH_actualizados = [(thetas[i], d, a, alpha)
for i, (theta, d, a, alpha) in enumerate(parametros_DH)]
            T_completa =
matriz_transformacion_completa(parametros_DH_actualizados)

            pos_actual = T_completa[:3, 3]
            error = pos_deseada - pos_actual

            if np.linalg.norm(error) < tolerancia:
                break

            J = calcular_jacobiano(thetas, parametros_DH)
            d_thetas = np.linalg.pinv(J) @ np.concatenate((error,
np.zeros(3)))
            thetas += d_thetas

        soluciones.append(thetas)

    return soluciones

def evaluar_soluciones(soluciones, parametros_DH,
pos_articulaciones_conocidas):
    mejor_solucion = None
    menor_error = np.inf

    for thetas in soluciones:
        parametros_DH_actualizados = [(thetas[i], d, a, alpha) for i,
(theta, d, a, alpha) in enumerate(parametros_DH)]
        T_completa =
matriz_transformacion_completa(parametros_DH_actualizados)

        pos_articulaciones = []
        T = np.eye(4)
        for param in parametros_DH_actualizados:
            T = T @ matriz_DH(*param)
            pos_articulaciones.append(T[:3, 3])
```

```
        error = np.sum([np.linalg.norm(pos - pos_conocida) for pos,
pos_conocida in zip(pos_articulaciones,
pos_articulaciones_conocidas)])

        if error < menor_error:
            menor_error = error
            mejor_solucion = thetas

    return mejor_solucion, menor_error

# Definir parámetros de Denavit-Hartenberg
a2 = 0.261  # Longitud del brazo superior (puedes ajustar este valor
según tu robot)
a3 = 0.266  # Longitud del antebrazo (puedes ajustar este valor según
tu robot)
d1 = 0.4  # Altura de la base

# Parámetros D-H para cada eslabón
parametros_DH = [
    (theta1, d1, 0, np.pi/2),  # Articulación 1 (Base)
    (theta2, 0, a2, 0),        # Articulación 2 (Hombro)
    (theta3, 0, a3, 0)         # Articulación 3 (Codo)
]

# Cálculo de thetas para todos los instantes de tiempo
thetas = []
thetas_iniciales = np.radians([0, 0, 0])  # Ángulos iniciales de las
articulaciones

for i in brazo_derecho_poses:
    pos_deseada = np.array(i[2])
    pos_articulaciones_conocidas = i[:2]

    soluciones = cinemática_inversa_múltiple(pos_deseada,
parametros_DH_base, thetas_iniciales)
    mejor_solucion, menor_error = evaluar_soluciones(soluciones,
parametros_DH_base, pos_articulaciones_conocidas)
    thetas_iniciales = mejor_solucion
    thetas.append(mejor_solucion)

thetas = np.array(thetas)

# Calcular velocidades y aceleraciones
dt = 1/60  # Intervalo de tiempo entre cada muestra (60 fps)
theta_dot = np.diff(thetas, axis=0) / dt
theta_ddot = np.diff(theta_dot, axis=0) / dt

# Agregar filas de ceros al principio para igualar las dimensiones
```

```
theta_dot = np.vstack([np.zeros((1, thetas.shape[1])), theta_dot])
theta_ddot = np.vstack([np.zeros((2, thetas.shape[1])), theta_ddot])

# Cálculo de la dinámica para todos los instantes de tiempo
np.random.seed(42)
masas = np.random.uniform(1, 10, 3)
inercia = [np.random.uniform(0.01, 0.1, 3) for _ in range(3)]

todas_fuerzas = []
todos_torques = []

for i in range(len(thetas)):
    thetas_inst = thetas[i]
    theta_dot_inst = theta_dot[i]
    theta_ddot_inst = theta_ddot[i]

    parametros_DH = [(thetas_inst[j], d, a, alpha) for j, (theta, d,
a, alpha) in enumerate(parametros_DH_base)]
    T_matrices = matriz_transformacion_completa2(parametros_DH)

    v = [np.zeros(3)]
    omega = [np.zeros(3)]
    v_dot = [np.zeros(3)]
    omega_dot = [np.zeros(3)]

    for j in range(len(parametros_DH)):
        R = T_matrices[j][:3, :3]
        P = T_matrices[j][:3, 3]

        omega_j = R.T @ omega[-1] + np.array([0, 0,
theta_dot_inst[j]])
        v_j = R.T @ (v[-1] + np.cross(omega[-1], P))

        omega_dot_j = R.T @ (omega_dot[-1] + np.cross(omega[-1],
np.array([0, 0, theta_dot_inst[j]]))) + np.array([0, 0,
theta_ddot_inst[j]])
        v_dot_j = R.T @ (v_dot[-1] + np.cross(omega_dot[-1], P) +
np.cross(omega[-1], np.cross(omega[-1], P)))

        omega.append(omega_j)
        v.append(v_j)
        omega_dot.append(omega_dot_j)
        v_dot.append(v_dot_j)

    fuerzas = []
    torques = []

    for j in reversed(range(len(parametros_DH))):
```

```
        F = masas[j] * (v_dot[j + 1] - np.array([0, 0, 9.81]))
        N = inercia[j] * omega_dot[j + 1] + np.cross(omega[j + 1],
inercia[j] * omega[j + 1])

        if j < len(parametros_DH) - 1:
            R = T_matrices[j + 1][:3, :3]
            P = T_matrices[j + 1][:3, 3]

            F += R @ fuerzas[-1]
            N += R @ (torques[-1] + np.cross(P, fuerzas[-1]))

        fuerzas.append(F)
        torques.append(N)

    fuerzas.reverse()
    torques.reverse()

    todas_fuerzas.append(fuerzas)
    todos_torques.append(torques)

# Imprimir resultados para cada instante de tiempo
for i in range(len(todas_fuerzas)):
    print(f"Instante {i}:")
    for j in range(len(todas_fuerzas[i])):
        print(f"Eslabón {j+1}:")
        print(f"Fuerza: {todas_fuerzas[i][j]}")
        print(f"Torque: {todos_torques[i][j]}")

# Calcular el torque máximo para cada eslabón
torques_maximos = []
num_eslabones = len(todos_torques[0])

for i in range(num_eslabones):
    torque_max = 0
    for frame in todos_torques:
        torque_magnitude = np.linalg.norm(frame[i]/10000000000)
        if torque_magnitude > torque_max:
            torque_max = torque_magnitude
    torques_maximos.append(torque_max)

print("Torques máximos calculados para cada eslabón:",
torques_maximos)

# Ejemplo de catálogo de motores
motores_catalogo = [
    {'modelo': 'Motor A', 'torque_max': 5.0, 'velocidad_max': 3000,
'potencia': 100},
```

```
    {'modelo': 'Motor B', 'torque_max': 10.0, 'velocidad_max': 2500,
'potencia': 200},
    {'modelo': 'Motor C', 'torque_max': 15.0, 'velocidad_max': 2000,
'potencia': 300},
    {'modelo': 'Motor D', 'torque_max': 20.0, 'velocidad_max': 1500,
'potencia': 400},
    # ... agregar más motores según sea necesario
]

# Función para seleccionar el motor adecuado
def seleccionar_motor(torque_requerido, motores_catalogo):
    for motor in motores_catalogo:
        if motor['torque_max'] >= torque_requerido:
            return motor
    return None

# Seleccionar motores para cada articulación
motores_seleccionados = []
for torque in torques_maximos:
    motor = seleccionar_motor(torque, motores_catalogo)
    motores_seleccionados.append(motor)

# Imprimir los motores seleccionados
for i, motor in enumerate(motores_seleccionados):
    if motor:
        print(f"Articulación {i+1}: {motor['modelo']} (Torque Máximo:
{motor['torque_max']} Nm, Velocidad Máxima: {motor['velocidad_max']}
RPM, Potencia: {motor['potencia']} W)")
    else:
        print(f"Articulación {i+1}: No se encontró un motor
adecuado")

if __name__ == "__main__":
    # Aquí puedes agregar cualquier código adicional para ejecutar
todo el proceso
    pass
```