TRABAJO FIN DE GRADO

GRADO EN INGENIERÍA MECÁNICA

UNIVERSIDAD DE ALMERÍA

ESCUELA SUPERIOR DE INGENIERÍA

DESARROLLO DE ESTIMADORES DE ESTADO EMPLEANDO LA LIBRERÍA DE SIMULACIÓN FÍSICA SIMBODY

Trabajo monográfico

Autor:

Raúl Aguilera López

Directores:

José Luis Blanco Claraco

Javier López Martínez

A mi familia.

A Bea y Edu.

Agradecimientos

Quiero agradecer a todas las personas que han estado conmigo, ayudándome y acompañándome a lo largo de estos años tan intensos.

A mi familia, sin cuyo apoyo no habría llegado a donde estoy ahora.

A mi amiga Bea, una de las primeras personas que conocí al comenzar la carrera, y con la que más tiempo he compartido desde entonces. Gracias por aguantarme, ayudarme y acompañarme todos estos años, tanto en los momentos buenos como en los malos.

A mi amigo Edu, al que ya conocía desde el instituto y con el que entré en Ingeniería, y con quien, aunque nos hayamos tenido que separar en tercero por ser de distinta especialidad, he seguido compartiendo buenos momentos.

A mis tutores y todos esos profesores que logran hacerte sentir motivación e interés por aquello que enseñan e investigan.

Índice

Índice de figuras

Índice de ecuaciones

Índice de tablas

Siglas

Siglas	Significado en español	Significado en inglés
EDI	Entorno de desarrollo integrado	Integrated Development Environment
ESS	Tamaño de muestra efectivo	Effective sample size
FDP	Función Distribución de Probabilidad	Probability Distribution Function
FK	Filtro de Kalman	Kalman Filter
FKE	Filtro de Kalman extendido	Extended Kalman Filter
FKU	Filtro de Kalman Unscented	Unscented Kalman Filter
FP	Filtro de partículas	Particle filter
GDL	Grado de libertad	Degree of freedom
IS	-	Importance Sampling
JIT	En tiempo de ejecución	Just in time
MCS	Montecarlo secuencial	Sequential Monte Carlo
SMC	Sistema multicuerpo	Multibody system
SO	Sistema operativo	Operative System
SIR	-	Sequential Importance Resampling
SIS	-	Sequential Importance Sampling
SISR	-	Sequential Importance Sampling with Resample

Resumen

La evolución de la computadora ha hecho posible la simulación de sistemas multicuerpo en tiempo real y el uso de estimadores de estado para inferir parámetros ocultos del estado del sistema. Entre estos estimadores de estado, se encuentran los filtros de Kalman, una opción eficiente y ampliamente usada, pero que no fueron formuladas para sistemas de segundo orden, no lineales y con restricciones, como suelen ser los sistemas multicuerpo. En dichos casos, se suelen utilizar distribuciones no paramétricas, como es el filtro de partículas. En este trabajo se pretende desarrollar una librería en C++ para la implementación de estimadores de estado, utilizando la técnica de filtro de partículas y la librería de simulación física Simbody.

Palabras clave: Sistemas multicuerpo, Estimadores de estado, Filtro de partículas, Simbody.

Abstract

The evolution of the computer has made possible to achieve real-time simulation of multibody systems and using state observers to infer hidden state variables of them. Kalman filters are an efficient and vastly used option among these states observers, originally not formulated for second-order, non-linear and restricted systems though, such as multibody systems. In those cases, non-parametric distributions are often used, particle filters as an example. The aim of this work is developing a C++ library to implement a state observer, using the particle filter technique and Simbody, a multibody physics library.

Keywords: Multibody systems, State observers, Particle filter, Simbody.

Capítulo 1:

Introducción

1. Introducción

1.1. Interés

La aparición de la computadora, y su evolución a lo largo del siglo XX, ha permitido la resolución numérica de ecuaciones que no eran posibles resolver de manera analítica en la mayoría de los casos. Entre ellas, se encuentran algunas como las ecuaciones de Navier-Stokes que rigen el comportamiento de los fluidos; o las ecuaciones de movimiento que describen la dinámica de los cuerpos. La resolución de éstas últimas ha impulsado el desarrollo de nuevos algoritmos de análisis de sistemas multicuerpo, y otras aplicaciones como los estimadores de estado.

Un estimador u observador de estado nos permite inferir parámetros (no medibles directamente) del estado de un sistema. Esto se consigue empleando un modelo más o menos simplificado del sistema y con varios sensores instalados en el mecanismo real, mediante un proceso iterativo en el que se resuelve el modelo y se aplica una corrección usando las medidas de los sensores [1] [2].

En la línea de estos estimadores recursivos, se encuentran los filtros de Kalman, una opción eficiente y ampliamente usada en multitud de campos, como visión artificial en Robótica [3], control y diagnóstico de motores diésel en Automoción [4] y también en estimación de sistemas multicuerpo [5].

Sin embargo, en su versión discreta, los filtros de Kalman fueron formulados para sistemas de primer orden, lineales y sin restricciones, cuando los sistemas multicuerpo son de segundo orden, no lineales y con restricciones [1]. Por esto, a pesar de que ha sido extendido a sistemas no lineales, en este trabajo se va a optar por una alternativa más flexible: el filtro de partículas o método Montecarlo secuencial.

Un filtro de partículas consiste en formular hipótesis ponderadas (llamadas partículas) del parámetro que se desea estimar (el objetivo, *target*), avanzando un modelo simplificado del sistema a lo largo del tiempo [2]. Se parte de un determinado estado inicial, y en cada iteración del filtro, se avanza un intervalo de tiempo determinado, se actualiza el peso de cada hipótesis y se elige un nuevo conjunto de partículas, aleatoriamente pero acorde a los nuevos pesos. Si se implementa correctamente, la distribución ponderada de partículas en todos los valores posibles del *target* tenderá con el tiempo a su función densidad de probabilidad [2].

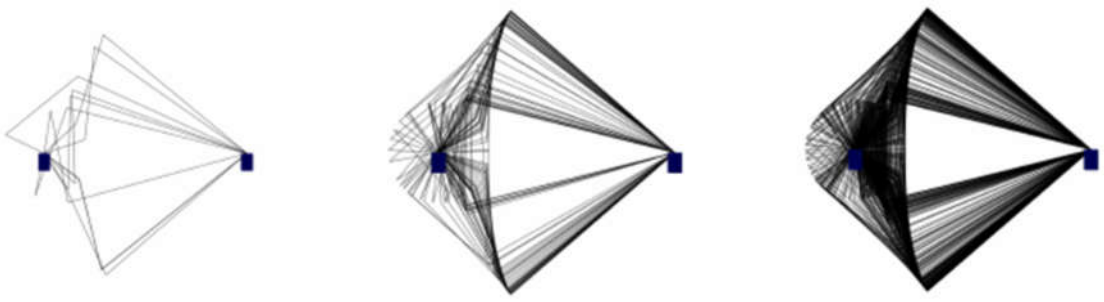

Figura 1. Hipótesis iniciales para distintos números de partículas, estimando la posición en un mecanismo de cuatro barras. De izquierda a derecha, el número de partículas es 10, 100 y 500. (Fuente: *[2]*)

Los filtros de partículas tienen también un gran campo de aplicación, destacando principalmente la Robótica, comúnmente en aplicaciones de seguimiento visual [6] [7]. Sin embargo, la creación de un modelo simplificado, pero suficientemente preciso de un sistema multicuerpo y su posterior resolución, no es una tarea sencilla. Por ello, se va a optar por utilizar una librería de simulación física. Entre las numerosas opciones disponibles [8] [9], se va a elegir Simbody, desarrollada principalmente por Michael Sherman [10].

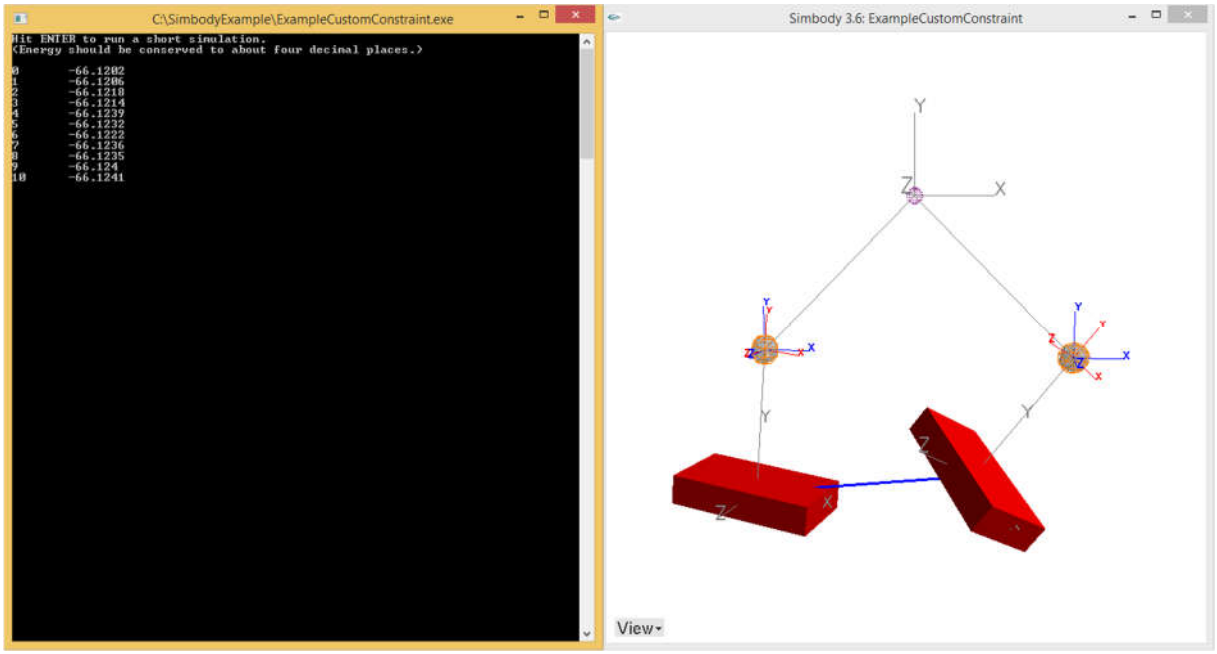

Figura 2. Ventana de comandos y visualizador de Simbody, ejecutando uno de los ejemplos de la librería.

Simbody es una librería de simulación física, eficiente y en C++, pensada para investigación biomédica, capaz de simular estructuras óseas de humanos y animales, pero aplicable también a biomoléculas, mecanismos, robots o vehículos. Aunque otros trabajos ya han usado Simbody (modelado de una columna vertebral [11] y de un robot humanoide [12]), nunca antes, que se sepa, se había desarrollado un estimador de estado empleando dicha librería, lo que haría de éste un trabajo novedoso.

1.2. Objetivos

El objetivo del trabajo es desarrollar una librería en C++ que permita implementar un estimador de estado mediante filtro de partículas para sistemas multicuerpo, basado en Simbody. El filtro de partículas se implementará en un mecanismo de cuatro barras cerrado, escogido por su sencillez.

Se valorará su viabilidad, robustez y precisión mediante simulaciones y ensayos posteriores basados en número de partículas, parámetros del filtro, costes computacionales, etc.

1.3. Fases de realización y cronograma

A continuación, se enumerarán y explicarán las distintas fases por las que ha pasado la realización de este trabajo:

- **(1) - Descarga e instalación de todo el software necesario**: este incluye la instalación de Visual Studio, Cmake y MATLAB, y todo el proceso de instalación y compilación de Simbody.
- **(2) - Profundización en el lenguaje de programación C++**: los conocimientos iniciales del lenguaje no eran suficientes, así que fue necesaria una etapa en la que aprender todas las características particulares de C++, especialmente las clases, herencia y polimorfismo y las plantillas.
- **(3) - Aprendizaje de la librería Simbody**: una vez conocido C++, el siguiente paso es aprender a realizar una simulación en Simbody. Para ello, fue de vital importancia leerse los manuales de Simbody y practicar con los ejemplos de la librería.
- **(4) - Implementación del algoritmo de filtro de partículas**: el siguiente paso una vez se conozca C++ y Simbody, es empezar la implementación del algoritmo recursivo de estimación de estado con filtro de partículas. La bibliografía sobre filtro de partículas ha resultado de mucha ayuda durante esta fase.
- **(5) - Análisis y optimización**: tras la primera implementación del filtro de partículas, comienza la larga etapa de optimización, solución de errores y análisis de su eficiencia y precisión.
- **(6) - Redacción de la memoria**: La redacción de este documento fue una etapa que comenzó paralela con la anterior, una vez se disponía de suficientes recursos y materia sobre la que redactar.

En la siguiente tabla se muestra un cronograma con las fechas de realización aproximadas y las horas totales estimadas de cada fase:

Fase	Horas	Fechas de realización
(1)	5	15/06/18 – 6/11/18
(2)	30	15/06/18 – 4/10/18
(3)	50	4/10/18 – 6/11/18
(4)	120	6/11/18 – 22/12/18
(5)	250	22/12/18 – 21/11/2019
(6)	150	21/06/19 – 21/11/2019

Tabla 1. Cronograma de este trabajo, con fases, fechas y horas totales dedicadas.

El número de horas dedicadas a este proyecto, por tanto, ha sido un total estimado de 605 horas.

1.4. Resumen de resultados

El objetivo de este trabajo era desarrollar una librería en C++ para la estimación con filtro de partículas, lo cual se ha conseguido. Su implementación para un mecanismo simulado sencillo como el mecanismo de cuatro barras para obtener un estimador de estado funcional también ha sido posible.

Sin embargo, su eficiencia no permite grandes números de partículas (por encima de 500 o 1000 partículas) ni pasos de simulación pequeños (del orden de 10 ms) si se desea realizar una estimación de estado en tiempo real, pues el tiempo de ejecución aumenta en gran medida.

Aunque, aun así, es posible realizar estimaciones de estado eficaces (por ejemplo, con 300 ms de paso y 200 partículas). Se considera que la librería tiene mucho potencial de mejora y puede optimizarse en gran medida.

Los esfuerzos de trabajos futuros deberán concentrarse en esta posible optimización de la librería, o en una posible implementación para la estimación de estado en un mecanismo real.

1.5. Competencias

La realización de este trabajo demuestra la adquisición y posesión de ciertas competencias. A continuación, se enumerarán dichas competencias conforme a su carácter, desde las más generales a aquellas más específicas.

Primero, se encuentran las competencias básicas aprobadas por el Real Decreto 1393/2007, de 29 de octubre, relacionadas con las enseñanzas universitarias oficiales en España:

- **RD1. Poseer y comprender conocimientos**: la propia elaboración de este trabajo demuestra posesión de conocimientos diversos pertenecientes, entre otras ramas, a la Automatización, Estadística, Informática o Mecánica.
- **RD2. Aplicación de conocimientos**: si bien la redacción este documento pueda no reflejarlo, el desarrollo de la librería en C++ para estimación con filtro de partículas requiere de la aplicación directa de los conocimientos poseídos, no solo de Programación, sino de Teoría de Mecanismos, entre otros.
- **RD3. Capacidad de emitir juicios**: para poder solucionar problemas relacionados con filtro de partículas, a menudo se genera un gran volumen de información, por lo que su correcta interpretación y juicio posterior sobre su solución es imprescindible.
- **RD4. Capacidad de comunicar y aptitud social**: la comunicación con los tutores, así como la misma defensa de este trabajo demuestra, no tanto la capacidad propia de comprensión sino la capacidad de transmitir a los demás conocimientos, problemas o conclusiones.
- **RD5. Habilidad para el aprendizaje**: una parte de los conocimientos necesarios han sido adquiridos durante la realización del trabajo y de forma autónoma. Un ejemplo sería el aprendizaje y uso del lenguaje de programación C++.

Después están las competencias transversales propias de la Universidad de Almería, aprobadas en Consejo de Gobierno de 17 de junio de 2008:

- **UAL1. Conocimientos básicos de la profesión**: especialmente relacionados con filtro de partículas, muy utilizado, por ejemplo, en visión artificial. Además, estos conocimientos facilitarán en el futuro la comprensión de otras técnicas relacionadas.
- **UAL2. Habilidad en el uso de las TIC**: dentro de estas tecnologías se encuentran el programa ofimático para redacción del documento, el entorno de desarrollo integrado (EDI) de programación en C++, repositorio online de código, páginas web para búsqueda de bibliografía, entre otras.
- **UAL3. Capacidad para resolver problemas**: mencionado en la RD3. Durante el desarrollo de la librería, la aparición de problemas ha sido constante e inevitable, y es necesario poder analizarlos y resolverlos adecuadamente para avanzar.
- **UAL4. Comunicación oral y escrita en la propia lengua**: un Trabajo de Fin de Grado es un documento formal, cuya elaboración requiere del uso correcto de la lengua propia. Está relacionada también con la RD4.
- **UAL5. Capacidad de crítica y autocrítica**: especialmente de autocrítica, fundamental para encontrar los defectos del trabajo propio para que puedan ser mejorados.
- **UAL7. Conocimiento de una segunda lengua**: la bibliografía utilizada en este trabajo, en casi su totalidad, se encuentra en inglés. Así mismo, los comentarios en archivos de código C++ y en su repositorio online se han escrito en inglés.
- **UAL9. Capacidad para aprender a trabajar de forma autónoma**: al igual que la RD5.

Por último, se encuentran las competencias específicas, relacionadas con el Grado. Entre ellas se encuentran:

- Competencias generales ordenadas por el BOE de Ingeniero Técnico Industrial.
    - **CT3. Conocimiento en materias básicas y tecnológicas, que les capacite para el aprendizaje de nuevos métodos y teorías, y les dote de versatilidad para adaptarse a nuevas situaciones**: relacionada con las RD1, RD5, UAL1 y UAL9.
    - **CT4. Capacidad de resolver problemas con iniciativa, toma de decisiones, creatividad, razonamiento crítico y de comunicar y transmitir conocimientos, habilidades y destrezas en el campo de la Ingeniería Industrial:** relacionada con las RD3, RD4, UAL3 y UAL5.
- Competencias de formación básica.
    - **CB1. Capacidad para la resolución de los problemas matemáticos que puedan plantearse en la ingeniería. Aptitud para aplicar los conocimientos sobre: álgebra lineal; geometría; geometría diferencial; cálculo diferencial e integral; ecuaciones diferenciales y en derivadas parciales; métodos numéricos; algorítmica numérica; estadística y optimización**: relacionada con las RD2 y UAL3, en especial, los problemas de estadística, geometría y métodos numéricos.
    - **CB2. Comprensión y dominio de los conceptos básicos sobre las leyes generales de la mecánica, termodinámica, campos y ondas y electromagnetismo y su aplicación para la resolución de problemas propios de la ingeniería**: en relación con las RD2 y UAL3, a destacar los conceptos de mecánica tales como coordenadas generalizadas, aceleraciones, fuerzas, momentos o inercias, para la resolución de problemas como la cinemática o dinámica de los cuerpos.

    - **CB3. Conocimientos básicos sobre el uso y programación de los ordenadores, sistemas operativos, bases de datos y programas informáticos con aplicación en ingeniería**: relacionada con la UAL2. Se puede destacar la programación en C++ y el uso de programas como MATLAB y otros EDIs.
- Competencias comunes de la rama Industrial.
    - **CRI6. Conocimientos sobre los fundamentos de automatismos y métodos de control**: conocimientos muy en relación con aquellos sobre estimadores de estado.
    - **CRI7. Conocimiento de los principios de teoría de máquinas y mecanismos**: fundamentales para la simulación de sistemas multicuerpo y mecanismos.

1.6. Estructura de la memoria

Este documento se divide, de mayor a menor nivel, en capítulos, secciones y apartados.

En el capítulo **2** se realizará una revisión bibliográfica, comentando las fuentes utilizadas y aportando algo de información sobre la historia y el estado de las distintas áreas de conocimiento. Concretamente, se dividirá en cuatro secciones, correspondientes a Librerías de simulación física (**2.1**), Lenguajes de programación (**2.2**), Teoría de mecanismos y sistemas multicuerpo (**2.3**) y Estimadores de estado (**2.4**).

El capítulo **3** se dedicará a la explicación de los materiales y métodos utilizados. Se ha dividido en tres secciones: Lenguaje C++ (**3.1**), donde serán enumeradas las características del lenguaje; Simbody (**3.2**), que dispondrá de varios apartados dedicados a la explicación exhaustiva de cada una de las características utilizadas de la librería de simulación; y Filtro de partículas (**3.3**), dividido en tres apartados en los cuales, respectivamente, se hablará de los fundamentos de estimadores de estado previos a la aplicación del filtro de partículas, de los modelos estadísticos que implementan las ecuaciones necesarias y del algoritmo recursivo que sigue un filtro de partículas para la estimación del estado de un sistema.

El capítulo **4** está dedicado a los resultados de este proyecto y su discusión. Se comenzará hablando sobre las dos librerías en las que se han trabajo, SimTKpf y Fourbar (**4.1** y **4.2**, respectivamente), comentando las clases y métodos que componen dichas librerías, así como los programas ejecutables principales donde se utilizan. Una sección adicional se dedicará al análisis de viabilidad, robustez y precisión, y la descripción de los experimentos realizados para lograr llevarlo a cabo.

En el capítulo **5** se plasmarán las Conclusiones obtenidas a partir de los resultados mostrados en el capítulo anterior, y posibles mejoras y correcciones con el propósito de optimizar el rendimiento de la librería y su estructura.

Por último, se presentará la bibliografía utilizada durante toda la redacción de este documento, además de un capítulo de anexos con todo el código de la librería.

Capítulo 2:

Revisión bibliográfica

2. Revisión bibliográfica

2.1. Librerías de simulación física

Una librería de simulación física es un conjunto de rutinas, fórmulas y objetos, cuyo objetivo es posibilitar la construcción de modelos de sistemas físicos, para poder llevar a cabo simulaciones con ellos. Las simulaciones permiten, sin necesidad de trabajar directamente con los sistemas físicos, extraer información de ellos, ya que los modelos empleados deberían reflejar el comportamiento del sistema que representan.

El uso de librerías de simulación física evita tener que implementar desde cero todas las leyes físicas necesarias para construir un modelo matemático de un sistema físico, disminuyendo el tiempo necesario antes de poder realizar una simulación. Su uso permite, de esta manera, concentrarse en mayor medida en la simulación que se desea llevar a cabo, ya que previamente solo se deberá instalar dicha librería y aprender su uso.

La primera librería de simulación que se va a mencionar es Box2D [13], una librería de código abierto y escrita en C++, que implementa una gran variedad de conceptos físicos como fricción, contacto, choques no elásticos, fuerzas, impulsos, momentos, articulaciones o motores. Box2D cuenta con manual de usuario y un foro activo, aunque su aplicación en ingeniería es reducida, ya que es una librería sencilla, que como su nombre indica, solo permite la construcción de modelos en dos dimensiones. Su uso principal es como motor físico para videojuegos. En la **Figura 3** se muestra un ejemplo en Box2D:

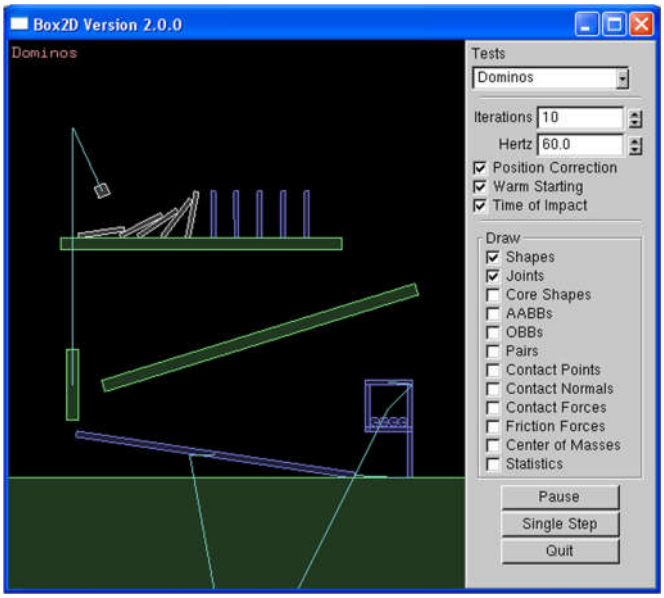

Figura 3. Uno de los ejemplos del banco de pruebas de Box2D. (Fuente: documentación de Box2D *[13]*)

En el ámbito de aplicación en ingeniería, se puede encontrar Gazebo [14], una librería de simulación robótica de código libre. Gazebo dispone de modelos de robots, gran variedad de sensores, simulaciones con gráficos en 3D y permite la utilización de otros motores físicos, de protocolos TCP/IP, simulaciones desde la nube y herramientas de líneas de comandos. Es una librería muy importante en el campo de la robótica.

Otra librería que destacar es Bullet [15], escrita en C++ y C, y también de código abierto. Sus aplicaciones varían desde realidad virtual, motor físico de videojuegos y efectos especiales en películas, hasta robótica e inteligencia artificial. Sus características principales son sus componentes de detección de colisiones y físicas de cuerpo rígido y cuerpo blando. Una librería muy potente que podría haber sido otra opción en este trabajo. En la siguiente figura puede verse su uso:

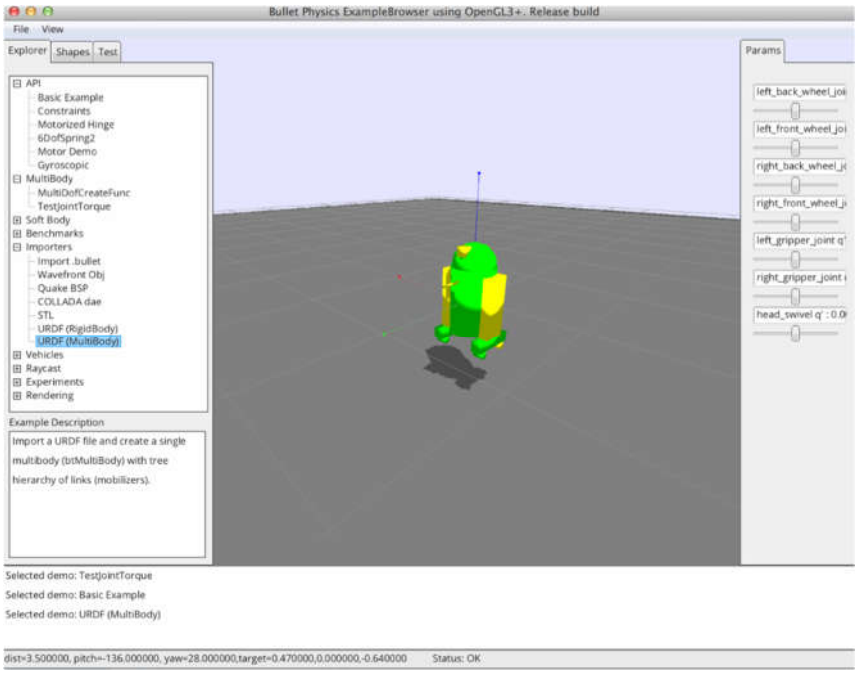

Figura 4. Ventana del navegador de ejemplos de Bullet. (Fuente: Documentación de Bullet *[15]*)

Por último, nos encontramos Simbody, una librería de simulación física de código abierto, eficiente y en C++, escrita principalmente por Michael Sherman, desarrollador en la Universidad de Stanford; y financiada gracias a la subvención U54 GM072970 de la Hoja de Ruta de los Institutos Nacionales de Salud (*NIH Roadmap*) de los Estados Unidos. Simbody fue pensada para investigación biomédica, capaz de simular estructuras óseas de humanos y animales, pero aplicable también a biomoléculas, mecanismos, robots o vehículos. Para este trabajo se ha escogido Simbody, por su carácter orientado a trabajos de investigación y desarrollo; y por su arquitectura, que como ya veremos en el apartado **3.2.1**, permitirá tener todas las variables del estado de un sistema en un solo objeto.

Simbody es uno de los muchos proyectos que se pueden encontrar en SimTK, una plataforma gratuita de alojamiento de proyectos de software sobre investigación biomédica [16] [17]. Hay otros proyectos de SimTK que valen la pena mencionar, siendo el primero Simmath [18], una librería orientada a objetos para la implementación eficiente de métodos matemáticos que incluyen optimización, diferenciación, integración o álgebra lineal, entre otros. También se encuentra SimTKcommon [19], una librería que proporciona definiciones de objetos esenciales para programación de más alto nivel, como pueden ser vectores y matrices. Por último, y como uno de los proyectos destacados en SimTK, está Opensim [20], una herramienta para el modelado, simulación, control y análisis de sistemas musculoesqueléticos en movimiento. Todos estos proyectos han sido desarrollados también por la universidad de Stanford, y mientras que Simbody emplea las librerías Simmath y SimTKcommon dentro de sus propios archivos, Opensim utiliza Simbody como motor de físicas para sus cálculos.

En el trabajo de Sherman et al. [10] se describen las características principales de Simbody. Existen además varios manuales para ayudar al usuario que utilice Simbody, que podemos encontrar en la sección de documentación en su repositorio de Github [21]. La guía de usuario de Simbody y Molmodel es el documento principal a la hora de aprender a usar Simbody. Molmodel es una API que usa Simbody para construir modelos de moléculas, que, aunque no resulta de utilidad en este trabajo, la guía ofrece información simplificada y sobre Simbody. En ella se presentan varios códigos como ejemplo y se explica línea a línea su funcionamiento. Una guía de programación avanzada también puede encontrarse en caso de que se busque crear clases derivadas de las que aporta Simbody. Por último, se encuentra una guía de Simmatrix, un paquete de software que permite operaciones con matrices y vectores de una manera eficiente de manera similar a MATLAB; y el manual de teoría, donde se relata con bastante detalle toda la estructura de Simbody y el proceso de cálculo que sigue.

2.2. Lenguajes de programación

Las computadoras son herramientas muy útiles capaces de realizar de forma rápida cálculos muy complicados para los humanos, procesos iterativos laboriosos y repetitivos, manejar volúmenes de información considerables y automatizar diversos aspectos de nuestra vida. Pero esto no las hace inteligentes. Las computadoras solo realizan las tareas que le son definidas, y hay que especificarlas de forma muy exacta y en un lenguaje que sean capaz de entender. Este lenguaje difiere por completo de las lenguas humanas, y es lo que se conoce como lenguaje de máquina o código máquina. Hoy en día, una computadora no tiene libre albedrío ni razona qué debe hacer en cada momento, solo ejecuta las instrucciones escritas en sus programas.

Sin embargo, un programa no es escrito directamente en código máquina, pues sería algo muy engorroso y propenso a fallos, y su corrección en caso de errores no sería nada fácil. En su lugar, existen numerosos lenguajes de programación, más similares a los utilizados por las personas que el código máquina. Estos se clasifican de forma diferente según cómo se realice su traducción a lenguaje de máquina:

- **Lenguajes compilados**, que se traducen a código máquina usando un programa conocido como compilador. La rapidez del código es dependiente de la efectividad del compilador durante la optimización, y puede darse la situación de que el código máquina resultante no se comporte bien en todos los sistemas operativos. El proceso de compilación de un programa se realiza antes de ser ejecutado y puede demorarse un tiempo. Un ejemplo es el propio C++.
- **Lenguajes interpretados** son leídos y ejecutados por un programa conocido como intérprete. Son muy versátiles y portables, pero el resultado es más lento en comparación con un programa equivalente compilado. Un ejemplo sería MATLAB.
- **Lenguajes compilados en tiempo de ejecución o "*just in time*" (JIT)**, son compilados rápidamente cuando deben ejecutarse (generalmente con muy poca optimización), y ofrecen un rendimiento no tan alto como los lenguajes compilados, pero sí mayor portabilidad. Un ejemplo podría ser Java.

Cuando se habla sobre lenguajes de programación, también se tiene en cuenta si es de bajo o alto nivel. El nivel de un lenguaje refleja cuán diferente es de aquel que utiliza una computadora, por lo que, a mayor nivel, menos similar es al código máquina.

Un lenguaje de bajo nivel es muy similar al código máquina y es más adecuado para programas como controladores de dispositivos o de muy alto rendimiento que necesitan acceso al hardware. Normalmente, se suele denominar lenguaje de bajo nivel al código máquina y al lenguaje ensamblador, aunque muchos lenguajes ofrecen elementos de bajo nivel. Como un lenguaje de bajo nivel está muy relacionado con hardware al que da instrucciones, suele ser poco portable. Los lenguajes de bajo nivel casi nunca se interpretan, ya que no suele ser necesario.

Un lenguaje de alto nivel se enfoca más en conceptos sencillos de entender para la mente humana, como los objetos o las funciones matemáticas. Por lo general, un lenguaje de alto nivel es más fácil de entender que un lenguaje de bajo nivel, y toma menos tiempo desarrollar un programa en un lenguaje de alto nivel que un lenguaje de bajo nivel. En cambio, se necesita sacrificar cierto grado de control sobre lo que realmente hace el programa resultante. Sin embargo, no es del todo imposible combinar funcionalidades de alto y bajo nivel en un idioma.

El lenguaje utilizado en este trabajo es C++, destacado por su eficiencia, en el cual está escrita la librería de simulación física Simbody. Durante todo este apartado se está usando como referencia “cplusplus.com” [22], una de las plataformas de recursos en C++ más grandes en Internet, la cual dispone de tutoriales, algoritmos e información relativa a los lenguajes de programación, y más concretamente, al propio C++, como sus características o su historia.

Los orígenes del lenguaje de programación C++ se remontan a 1979, cuando Bjarne Stroustrup estaba trabajando para su tesis doctoral en investigación. Stroustrup había trabajado con un lenguaje llamado Simula, un lenguaje diseñado principalmente para simulaciones. Simula 67 fue más concretamente la variante del lenguaje con la que trabajó Stroustrup, y es considerado el primer lenguaje de programación orientado a objetos. Stroustrup encontró el paradigma de orientación a objetos muy útil para el desarrollo de software, sin embargo, el lenguaje Simula era demasiado lento para tener un uso práctico.

Poco después, empezó a trabajar en "C con clases", que pretendía ser un superconjunto del lenguaje C, y cuyo objetivo era agregar programación orientada a objetos a dicho lenguaje, que era y sigue siendo un lenguaje muy respetado por su portabilidad sin sacrificar la velocidad o la funcionalidad de bajo nivel. Este idioma incluía así clases, herencia básica, funciones inline, funciones con argumentos predeterminados y comprobación de tipado fuerte, además de todas las características del lenguaje C.

El primer compilador de C con clases fue llamado Cfront, derivado de un compilador de C llamado CPre. Cfront fue diseñado para traducir el código de C con clases a simple C. Una característica bastante interesante que cabe destacar es que Cfront fue escrito mayoritariamente en C con clases, convirtiéndose en un compilador *self-hosting*, lo que significa que puede compilarse por sí mismo. Cfront se abandonaría más tarde en 1993 después de que se volviera difícil integrar nuevas características en él como, por ejemplo, las excepciones de C++. No obstante, Cfront tuvo una gran influencia en la implementación de futuros compiladores y en el sistema operativo (SO) Unix.

En 1983, el nombre del idioma se cambió a C++. El operador ++ se emplea en el lenguaje C para incrementar el valor de una variable, dando una idea del concepto de cómo Stroustrup consideró el lenguaje. Alrededor de ese tiempo se agregaron muchas características nuevas, siendo las más notables las funciones virtuales, la sobrecarga de funciones, las referencias con el símbolo &, la palabra clave const y los comentarios de una sola línea que usan dos barras diagonales, una característica tomada del lenguaje BCPL.

En 1985, fue publicada la primera obra de Stroustrup, "*The C++ Programming Language*" [23]. Ese mismo año, C++ fue implementado como producto comercial. El idioma aún no estaba estandarizado oficialmente, por lo que el libro era una referencia muy importante. C++ se actualizó de nuevo en 1989 para incluir miembros protegidos y estáticos, así como la herencia de varias clases.

Posteriormente, en 1990, Stroustrup publicó un manual de referencia [24] y se lanzó el compilador Turbo C ++ de Borland como un producto comercial. Turbo C ++ agregó una gran cantidad de bibliotecas adicionales que tendrían un impacto considerable en el desarrollo de C++. El compilador todavía se usa ampliamente hoy día, a pesar de que su última versión estable fue en 2006.

En 1998, el comité de estándares de C++ publicó el primer estándar internacional para C++ ISO/IEC 14882:1998, que sería conocido informalmente como C++98, utilizando el manual de referencia como base durante su redacción. La biblioteca de plantillas estándar (STL), cuyo concepto ya venía desarrollándose desde 1979, también fue incluida. Tras haber sido informados con múltiples problemas con este estándar, en 2003, el comité lo revisó y cambió en consecuencia, dando lugar al conocido como C++03.

En 2005, el comité de estándares de C++ publicó un informe técnico (denominado TR1) en el cual se detalla varias características para ser agregadas al último estándar de C ++. El nuevo estándar fue apodado informalmente C++0x, ya que se esperaba que fuera lanzado en algún momento antes del final de la primera década. Sin embargo, el nuevo estándar no fue lanzado hasta 2011. Se publicaron varios informes técnicos hasta entonces, y algunos compiladores comenzaron a agregar soporte experimental para las nuevas características.

Finalmente, a mediados de 2011, se terminó el nuevo estándar (denominado C++11). El proyecto de la biblioteca Boost tuvo un impacto considerable en el nuevo estándar, y algunos de los nuevos módulos se derivaron directamente de las bibliotecas correspondientes de Boost. Algunas de las características nuevas incluyen el soporte de expresiones regulares, una biblioteca completa de aleatorización, una nueva biblioteca de tiempo, el soporte de tipos atómicos, una biblioteca de subprocesos estándar (de la que, hasta 2011, carecían C y C++), una nueva sintaxis para bucles que proporciona una funcionalidad similar a los bucles foreach en otros idiomas, la palabra clave auto, las nuevas clases contenedores, un mejor soporte de uniones y listas de inicialización de matrices y plantillas variádicas.

En 2014, muchas de las características que introdujo C++11 fueron extendidas y mejoradas con el nuevo estándar (C++14); y este, a su vez, fue remplazado en 2017 (estándar C++17). C++17 es el estándar más reciente del lenguaje, pero está prevista una futura revisión, (conocida informalmente como C++20) que será una actualización importante, como lo fue C++11 en su día.

2.3. Teoría de mecanismos y sistemas multicuerpo

La principal referencia en español en cuanto a teoría de mecanismos usando métodos numéricos es el libro titulado “Teoría de Maquinas” [25] de Alejo Avello Iturriagagoitia, profesor de Tecnun, la Escuela Superior de Ingenieros de la Universidad de Navarra. De este libro, de gran importancia para la docencia en Ingeniería, serán de utilidad los capítulos 1 a 5, que hablan sobre terminología de los mecanismos, análisis estructural de mecanismos, cinemática y dinámica de solidos rígidos y análisis cinemático y dinámico por métodos numéricos; a pesar de que el libro trata otros temas como levas, trenes de engranajes, equilibrado de rotores o vibraciones en sistemas mecánicos.

Otro libro de utilidad será “Geometric Design of Linkages” [26], que aborda el tema de los acoplamientos y mecanismos desde un punto de vista más geométrico. Será de especial interés el apartado 1, que habla sobre algunos conceptos generales sobre acoplamientos, su movilidad y su clasificación; y el apartado 2.4, que trata sobre el rango de movimiento de uno de los mecanismos planares más importantes: el mecanismo de cuatro barras, el cual será tratado en este trabajo.

Por último, el propio manual de teoría de Simbody explica con bastante detalle conceptos básicos sobre sistemas multicuerpo (SMC) y también las ecuaciones de movimiento que describen estos sistemas. Estos conocimientos son fundamentales para poder comprender los SMC y como trabaja con ellos Simbody.

2.4. Estimadores de estado

Es habitual utilizar simulaciones de SMC en la industria para agilizar el proceso de desarrollo de nuevos productos. Al principio eran empleadas para predecir el comportamiento del producto, o bien porque la construcción de un prototipo resultaba complicada o costosa. Más tarde, con el desarrollo de nuevos procedimientos y el aumento de la potencia computacional, las simulaciones en tiempo real se volvieron posibles, permitiendo a personas y máquinas interactuar con los SMC a través de simulaciones y bancos de pruebas virtuales.

Hoy en día, con la llegada de computadores de bajo coste y consumo de energía, las simulaciones de SMC pueden ser ejecutadas en vehículos y robots como parte de sus algoritmos de control, proporcionando información sobre variables no medibles. El problema de este enfoque es que, en general, un SMC puede ser muy preciso a corto plazo si las fuerzas son conocidas con exactitud, pero con el tiempo, diverge [27].

Por esta razón, los SMC deben ser corregidos con información que provenga de los propios mecanismos para poder ser usados como estimadores de estado fiables. Una opción para conseguirlo es emplear técnicas de fusión de datos, para combinar la información proveniente de los SMC con aquella proporcionada por los sensores reales instalados en la máquina.

Un ejemplo destacado de estimador de estado empleando esta técnica de fusión de datos es el filtro de Kalman (FK), desarrollado por Rudolf Emil Kalman en 1960 [28]. El FK no es otra cosa que un conjunto de ecuaciones matemáticas aplicadas de forma recursiva para estimar el estado de un sistema. El algoritmo se basa en un bucle con realimentación en el que el filtro estima el estado en algún momento, y este es corregido con las mediciones, que constan de un cierto ruido. Las ecuaciones matemáticas entrarían entonces en dos grupos, las responsables de avanzar el estado a través del tiempo para obtener la estimación de la siguiente iteración (*a priori*); y aquellas que se encargan de la realimentación, combinando las mediciones con la estimación del estado, obteniendo una nueva estimación corregida (*a posteriori*) [29].

Sin embargo, el FK presenta algunos problemas. Originalmente, fue formulado para modelos lineales, sin restricciones y de primer orden, mientras que los SMC, por lo general, son modelos altamente no lineales, con restricciones y de segundo orden. Además, la implementación del algoritmo debe ser lo suficientemente eficiente como para poder ser ejecutado en tiempo real. Actualmente existe un campo de investigación abierto sobre este tema [27]. Múltiples variaciones del FK han sido desarrolladas desde su aparición, entre ellas, el filtro de Kalman extendido (FKE), que actúa linealizando el sistema; o el filtro de Kalman *Unscented* (FKU), aplicado cuando existen fuertes no-linealidades

La familia de filtros de Kalman supone distribuciones de probabilidad paramétricas, principalmente gaussianas multivariantes o suma de gaussianas, por lo que las integrales de las ecuaciones en las que aparecen las distribuciones resultan ser también son gaussianas. Esta elección conlleva almacenar pocos parámetros en cada iteración del algoritmo, y hace estos filtros muy eficientes computacionalmente. Por otro lado, no son capaces de distinguir entre distintas configuraciones de un mecanismo ni son aplicables cuando no se conoce el estado inicial [2].

En dichos casos, como el que se va a tratar en este trabajo, se emplean distribuciones no paramétricas. Una técnica conocida como *Importance Sampling* (IS), funciona de manera bastante intuitiva, asumiendo una función distribución de probabilidad (FDP) a la que se llama distribución de importancia, y distribuyendo un cierto número de muestras ponderadas en el espacio de estados posibles a partir de ella, de manera que habrá un mayor número de hipótesis y de mayor peso alrededor de la zona que se haya considerado de mayor importancia. Para poder usar este algoritmo como estimador de estado, esos pesos deben actualizarse de manera recursiva. Además, como no conocemos la FDP, se van a asumir pesos iguales para todas las muestras (todas son igual de significativas). Con estas variaciones, la técnica es conocida como *Sequential Importance Sampling* (SIS). Sin embargo, con SIS la varianza de los pesos aumenta a lo largo del tiempo: la mayoría tienden a cero y unos pocos se aproximan a uno (degeneración de la muestra), lo cual significa que la mayor parte de las muestras está siendo desperdiciada en hipótesis que no son probables, mientras que solo unas pocas son realmente significativas. La solución es desechar muestras con baja probabilidad y sustituirlas por otras que se encuentren en las regiones de mayor probabilidad. Este remuestreo se conoce como *resample*, y la nueva técnica SIS mejorada se conoce como *Sequential Importance Sampling with Resample* (SISR) o *Sequential Importance Resampling* (SIR). Cuando el SIR se aplica para realizar inferencia bayesiana, el método es conocido en la literatura principalmente como Montecarlo secuencial (MCS), *Condensation* o como filtro de partículas (FP). En este trabajo utilizaremos el último término, refiriéndonos a cada una de las muestras como “partículas”.

Capítulo 3:

Materiales y métodos

3. Materiales y métodos

3.1. Lenguaje C++

C++ tiene una serie de características que definen todo lo que puede ofrecer como lenguaje de programación: [22]

- **Es un lenguaje abierto estandarizado por ISO**: durante un tiempo, C++ no tuvo un estándar oficial y se mantuvo por un estándar de facto. Sin embargo, desde 1998, C++ está estandarizado por un comité de la ISO.
- **Es un lenguaje compilado**: C++ es compilado directamente en el código nativo de una máquina, lo que le permite ser uno de los idiomas más rápidos del mundo, si está optimizado.
- **Es un lenguaje inseguro fuertemente tipado**: C++ es un lenguaje que espera que el programador sepa lo que está haciendo, pero como resultado permite un control increíble.
- **Soporta tanto la tipificación explícita como la implícita**: desde su último estándar, C++ admite tanto tipificación explícita como implícita, aportando flexibilidad y un método para evitar verbosidad innecesaria.
- **Soporta tanto la comprobación de tipos estática como la dinámica**: C++ permite verificar las conversiones de tipos en tiempo de compilación o en tiempo de ejecución, lo que nuevamente ofrece otro grado de flexibilidad. Sin embargo, la mayoría de las comprobaciones de tipos C++ son estáticas.
- **Ofrece muchas opciones de paradigmas**: C++ ofrece un soporte extraordinario para los paradigmas de programación imperativa, genérica y orientada a objetos, y también son posibles muchos otros paradigmas.
- **Es portable**: como lenguaje abierto y de los más utilizados en el mundo, C++ posee una amplia gama de compiladores que se ejecutan en muchas plataformas diferentes con soporte. Todo código que utilice exclusivamente la biblioteca estándar de C++ se podrá ejecutar en muchas plataformas con pocos o casi ningún cambio.
- **Es retrocompatible con C**: al ser un lenguaje que se basa directamente en C, es compatible con casi todos los códigos C. Además, C++ puede usar bibliotecas de C con poca o ninguna modificación de su código.
- **Tiene un increíble soporte de bibliotecas**: con solo una simple búsqueda se puede acceder a miles de bibliotecas para C++.

El EDI elegido para la programación en C++ ha sido Visual Studio Community 2017. Además, se ha usado CMake 3.14.2 para la configuración del proyecto en dicho entorno. El compilador utilizado es Microsoft Visual C++ (conocido como MSVC++) y el generador de CMake es Visual Studio 15 2017.

3.2. Simbody

Esta sección ha sido redactada utilizando como referencias el artículo original sobre Simbody [10] y la documentación disponible en su repositorio de Github [21], siendo de especial importancia el manual de teoría y la guía de usuario de Simbody.

También se destacará que la versión de Simbody instalada para este trabajo ha sido la 3.6.1.

3.2.1. Estructura

La arquitectura de Simbody se centra en tres objetos principales, denominados Sistema, Estado y Estudio (*System*, *State* y *Study*).

El Sistema es una representación computacional del modelo matemático de un sistema físico (fuerzas, cuerpos, uniones, etc.), y está a su vez compuesto por varios subsistemas. El Sistema permanece, sin embargo, constante durante toda la simulación, siendo los valores de todos sus parámetros los que varían, afectando al comportamiento del Sistema. Este conjunto completo de valores de las variables de un Sistema es almacenado en un objeto aparte, llamado Estado. Los valores presentes en un Estado determinan completamente la respuesta de un Sistema. Un Sistema y uno o más Estados son agrupados en un Estudio, que representa un experimento llevado a cabo para revelar algo sobre el Sistema. El resultado de un Estudio es un Estado o conjunto de Estados que satisfacen unos criterios previamente definidos, así como los resultados que el Sistema puede calcular directamente con ellos. Este conjunto de Estados es denominado trayectoria (*trajectory*). Toda esta estructura queda representada en la siguiente figura:

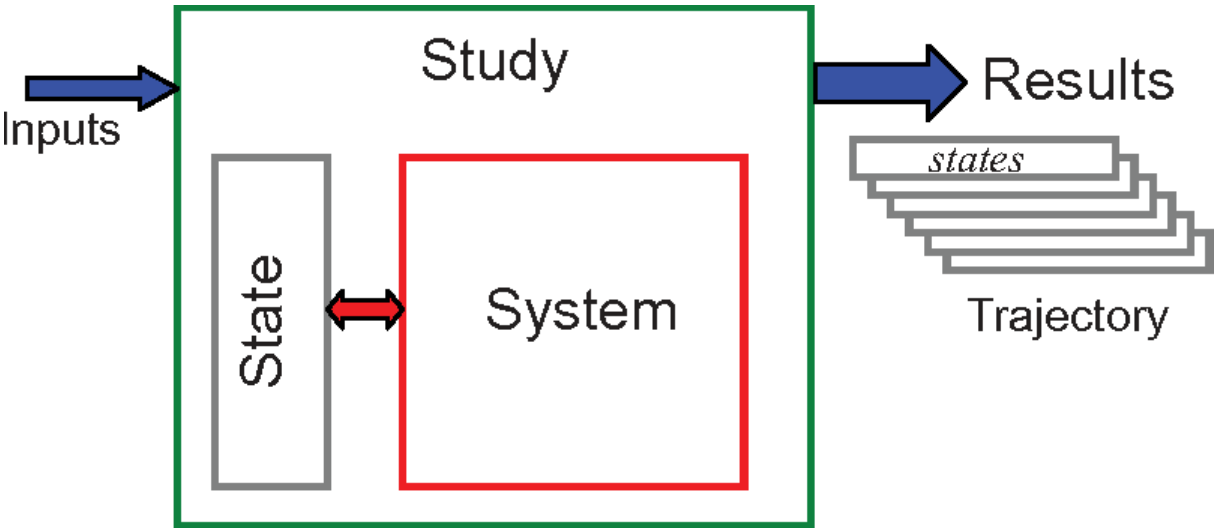

Figura 5. Arquitectura de Simbody. Un Sistema de solo lectura contiene todas las componentes de un modelo y define su parametrización. Los valores de dichos parámetros se almacenan en un Estado. Un Estudio genera un conjuntos de Estados como solución. (Fuente: *[10]*)

El concepto de estado en Simbody es bastante general, englobando cualquier valor variable en el Sistema, tanto variables continuas como tiempo, posiciones y velocidades como variables discretas, memoria de eventos pasados, opciones de modelado y algunos parámetros como longitudes y masas.

Los Sistemas, por otro lado, están formados por piezas interconectadas llamadas *Subsystems* (Subsistemas), tal y como ilustra la **Figura 6**. Tanto el Sistema como cada uno de estos subsistemas pueden tener sus propios parámetros, y el Sistema garantiza que las variables que necesitan cada uno de estos subsistemas, sean proporcionadas por el Estado general del Sistema. Cabe destacar que esta organización no es jerárquica, ya que a pesar de que los modelos jerárquicos son una manera muy potente de representar el mundo físico, los recursos informáticos son planos y sin jerarquía.

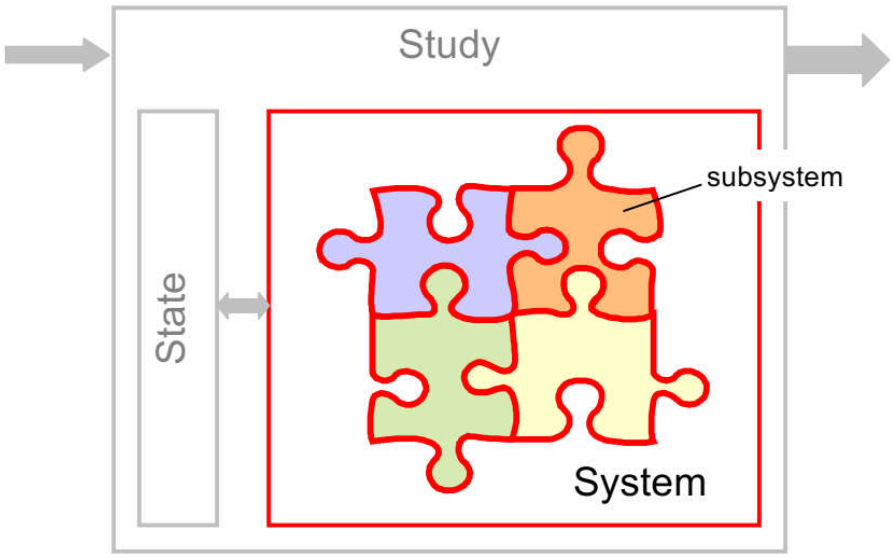

Figura 6. Analogía de un rompecabezas para ilustrar las conexiones entre un Sistema y sus correspondientes subsistemas. Los subsistemas son como piezas que se encuentran conectadas entre sí, y en sus extremos se conectan con el propio Sistema, formando un todo. (Fuente: Manual de teoría de Simbody *[21]*)

Los cálculos realizados por los distintos subsistemas son interdependientes entre sí ya que poseen dependencias entrelazadas. Estas dependencias pueden, sin embargo, desenlazarse realizando el proceso de cálculo por "etapas", siendo responsabilidad del Sistema secuenciar los subsistemas de forma apropiada a través de dichas etapas.

Simbody proporciona los componentes necesarios para representar computacionalmente las mecánicas de un SMC completo. El más importante es el objeto *SimbodyMatterSubsystem*, el subsistema de materia, encargado de representar cuerpos con masa interconectados entre sí. Por otro lado, se encuentran los subsistemas de fuerzas, que pueden ser definidos por el usuario, provenir de otro programa o usarse los que proporciona Simbody. De esta manera, Simbody puede llevar a cabo una gran variedad de Estudios, debido a que entiende el concepto de fuerza y sabe cómo aplicarlas. Sin embargo, no es consciente ni de donde provienen ni qué las genera.

Ambos subsistemas requieren de variables de estado para parametrizar el Sistema, pero como se ha explicado anteriormente, el Sistema y, por tanto, sus subsistemas, son objetos que almacenan los valores de sus parámetros en un objeto Estado aparte. Ambos, además, están interconectados de manera que, por ejemplo, una determinada fuerza aplicada en un cuerpo necesitará los valores de posición y velocidad que devuelva el subsistema de materia, mientras que las aceleraciones que definen dichos valores dependen de los propios valores de las fuerzas. El Sistema que describe un SMC en Simbody estaría estructurado como muestra la **Figura 7**.

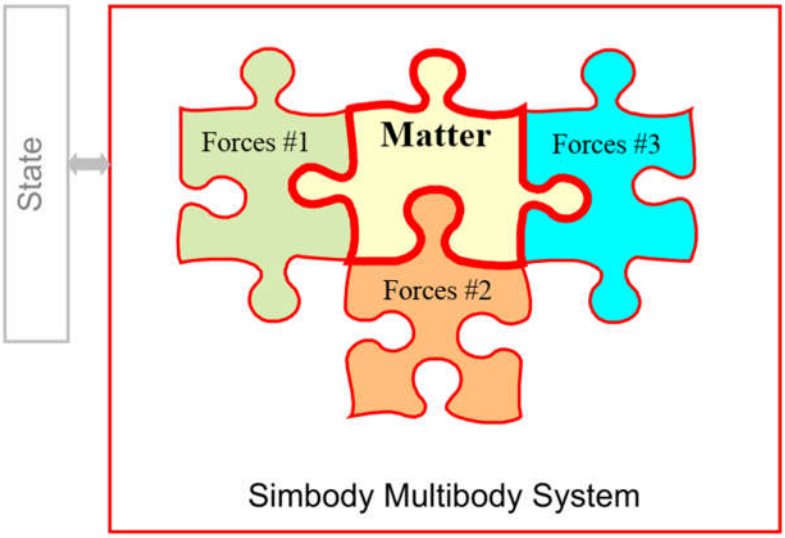

Figura 7. Esquema del Sistema de un SMC en Simbody, y la interconexión de sus subsistemas. (Fuente: Manual de teoría de Simbody *[21]*)

Se requiere manejar cuidadosamente un Estado para obtener una simulación correcta y eficiente, ya que puede haber multitud de valores que aporten información valiosa, como posiciones, fuerzas o aceleraciones. Los cálculos necesarios para obtenerlos pueden ser costosos computacionalmente, así que es de vital importancia que se calculen una única vez para un determinado estado y guardar el resultado obtenido. Cuando el Estado cambia durante la simulación, estos valores deben ser recalculados para no hacer referencia a valores desactualizados. Esto se debe realizar de forma automática para evitar errores, pero no es sencillo detectar cuándo hacerlo.

Simbody aborda el problema incluyendo un espacio dentro del objeto Estado destinado a almacenar cálculos dependientes del propio Estado. Este espacio de almacenamiento recibe el nombre de *realization cache*, mientras que el proceso por el que se calcularán los valores que se almacenan en él se denomina *realization*. Se debe entender este proceso como presentar un Estado a un Sistema para que este último calcule las consecuencias físicas de los valores almacenados en el Estado. En este trabajo se va a interpretar como una "actualización" de los valores almacenados en dicha caché, la "caché de actualización".

En la práctica, el proceso de evaluación de un SMC sigue unos pasos claramente ordenados: se deben conocer las posiciones de los cuerpos antes de calcular sus velocidades, dichas velocidades son necesarias para realizar el cálculo de las fuerzas y se deben conocer las fuerzas para poder calcular las aceleraciones. Así mismo, un cambio en una de las variables invalida los cálculos realizados a posteriori, pero no aquellos calculados con anterioridad (la variación de una velocidad generalizada invalida los cálculos de velocidades, fuerzas y aceleraciones, pero no los de posiciones). La arquitectura de Simbody integra esta estructura y divide la caché de actualización y las variables de un Estado conforme a unas etapas (*Stages)*, como muestra la **Figura 8**.

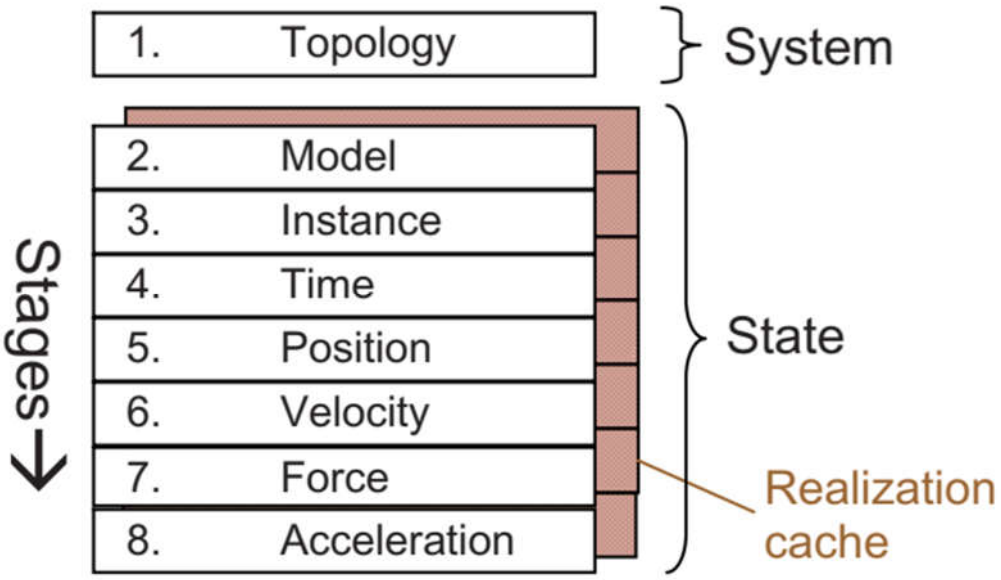

Figura 8. Estructura por etapas de Simbody. La construcción del Sistema supone la primera etapa del proceso de cálculo durante una simulación. (Fuente: Guia de usuario de Simbody *[21]*)

Aunque se almacene dentro de un Estado, la caché de actualización no supone ninguna parte lógica del estado de un Sistema. Son solamente cálculos intermedios derivados del Estado, que pueden ser fácilmente descartados y recalculados cuando sea preciso. Su necesidad es debida exclusivamente a motivos de eficiencia a la hora de realizar cálculos, dada la estructura de Estado, Sistema y Estudio. Además, las entradas de la caché de actualización relativas a cada etapa se invalidan automáticamente cuando se efectúa un cambio en una variable del Estado de dicha etapa o previas, quedando "desactualizadas". Cualquier intento de acceso a una entrada no válida provocará que se recalculen dichas entradas o que se genere un mensaje de error oportuno.

3.2.2. Resumen matemático

Un SMC puede verse como un sistema físico compuesto de cuerpos más o menos rígidos, de cierta masa, que se desplazan de forma relativa unos respecto de los otros. Como tales, una simulación no puede trabajar con ellos, solo con objetos virtuales. La solución es definirlos como un conjunto de ecuaciones con las que las simulaciones sí pueden trabajar, y si dichas ecuaciones están bien diseñadas, deberían reflejar de alguna manera el comportamiento del sistema que pretenden emular. Más concretamente, y como ya se ha comentado en el apartado anterior, el estado de un sistema es descrito como un conjunto de variables de estado. Este vector de variables de estado es referido como $\mathbf{y}$, y el objetivo de la simulación será integrar las ecuaciones de movimiento, que tienen la siguiente forma:

$$\frac{d\mathbf{y}}{dt} = \mathbf{f}(\mathbf{y}, t) \tag{1}$$

La intención es obtener $\mathbf{y}(t)$ en función del tiempo, que representa el conjunto completo de estados que anteriormente había sido denominado trayectoria. La función $\mathbf{f}(\mathbf{y},t)$ refleja las leyes físicas y las fuerzas que actúan sobre los cuerpos.

El vector de estado $\mathbf{y}$ puede dividirse en coordenadas generalizadas, referidas como $\mathbf{q}$, velocidades generalizadas, referidas como $\mathbf{u}$, y variables auxiliares, referidas como $\mathbf{z}$, de manera que el vector de estado es la concatenación de estos tres vectores: $\mathbf{y} = [\mathbf{q}, \boldsymbol{u}, \boldsymbol{z}]$. El uso de coordenadas generalizadas evita la imposición de restricciones por su propia definición, aunque a veces son necesarias restricciones adicionales. Una restricción, matemáticamente, es una ecuación algebraica que debe cumplirse en todo momento durante la simulación:

$$\mathbf{c}(t, \mathbf{q}, \mathbf{u}) = \mathbf{0} \tag{2}$$

Existe una ecuación por cada restricción impuesta en el sistema, y todas juntas, representan una variedad (*manifold*) sobre la que debe encontrarse siempre el vector de estado.

Las ecuaciones (1) y (2) suponen en todo momento que el sistema está modelado de manera que evolucione de forma continua a lo largo del tiempo, de acuerdo con un conjunto de ecuaciones diferenciales y algebraicas. A veces no es suficiente, y el sistema debe cambiar discontinuamente a tiempos discretos, como en algunos modelos de contacto. Matemáticamente, esto se consigue con funciones activadoras de eventos (*event trigger functions*), funciones arbitrarias de tiempo y estado, monitoreadas constantemente durante la simulación y que activan eventos cuando cruzan el valor cero:

$$\mathbf{e}(t, \mathbf{y}) = 0 \tag{3}$$

Cuando un evento ocurre, se llama al *handler* de evento correspondiente, que se encarga de modificar el estado de forma discontinua y arbitraria. El concepto de estado queda extendido para incluir *variables discretas*, las cuales quedarán referidas como $\mathbf{d}$. Estas variables pueden contener valores de cualquier tipo, y solo pueden ser modificadas por *handlers* de evento, a tiempos discretos. Las ecuaciones (1)-(3) pueden depender del valor actual de las variables discretas:

$$\frac{d\mathbf{y}}{dt} = \mathbf{f}(\mathbf{d}; \mathbf{y}, t) \tag{4}$$

$$\mathbf{0} = \mathbf{c}(\mathbf{d}; t, \mathbf{q}, \mathbf{u}) \tag{5}$$

$$\mathbf{0} = \mathbf{e}(\mathbf{d}; t, \mathbf{y}) \tag{6}$$

Se muestra $\mathbf{d}$ con punto y coma en las funciones como recordatorio de que su valor se mantiene constante durante intervalos de tiempo continuos, mientras que el valor de t e $\mathbf{y}$ varía constantemente.

3.2.3. Etapas de computación

Un objeto Estado, con el objetivo de describir completamente el estado de un sistema, almacena los valores de tiempo t, variables discretas $\mathbf{d}$ y variables continuas $\mathbf{y}$. Pero como ya se comentó en el apartado **3.2.1**, también dispone de un espacio en el que almacenar ciertos valores de interés, calculados a partir de los anteriores, la caché de actualización. El motivo es calcularlos una vez, poder acceder a ellos cuando se quiera y evitar recalcularlos innecesariamente. La caché está dividida en etapas, como se puede apreciar en la **Figura 8**, y cuando se actualiza, especificándose hasta qué etapa, se calculan los datos almacenados en ella pertenecientes a dicha etapa, y todas las anteriores.

Las etapas hasta la número 3 son necesarias para la creación e inicialización del sistema, pero aquellas que contienen información accesible de interés para la mayoría de las simulaciones son las siguientes:

- **Etapa 4, Tiempo (*Time*)**: todavía no se ha calculado información derivada. Se puede consultar el valor de t, $\mathbf{d}$ e $\mathbf{y}$ pero nada más.
- **Etapa 5, Posición (*Position*)**: las posiciones de todos los cuerpos en coordenadas cartesianas son conocidas.
- **Etapa 6, Velocidad (*Velocity*)**: las velocidades de todos los cuerpos en coordenadas cartesianas son conocidas.
- **Etapa 7, Fuerza (*Force*)**: las fuerzas que actúan sobre cada uno de los cuerpos son conocidas, así como las energías cinética y potencial totales del sistema.
- **Etapa 8, Aceleración (*Aceleration*)**: las derivadas respecto del tiempo de todas las variables de estado y todos los valores de las funciones activadoras de eventos son conocidos.

El Estado se ocupará de que la información almacenada en la caché sea consistente con los valores actuales de las variables de estado, retrocediendo a una etapa anterior si alguno de ellos es modificado, invalidando entradas de la caché posteriores:

- Si se modifica t, se devuelve el Estado a la etapa de Instancia (*Instance*).
- Si se modifica $\mathbf{q}$, se devuelve el Estado a la etapa de Tiempo.
- Si se modifica $\mathbf{u}$, se devuelve el Estado a la etapa de Posición.
- Si se modifica $\mathbf{z}$, se devuelve el Estado a la etapa de Velocidad.
- Cuando se define una variable discreta $\mathbf{d}$, se especifica a qué etapa se debe devolver el estado si fuera modificada, para invalidar entradas de caché que dependan de ella.

3.2.4. Eventos

Como se ha descrito en el apartado **3.2.2**, se señala que debe suceder un evento cuando una función activadora de evento (**Ecuación 6**) cruza el valor cero. Esto implica que su valor debe calcularse en cada iteración, y cuando se detecta que se ha producido un cambio de signo de un paso al siguiente. Cuando esto ocurre, se sabe que un evento ha ocurrido en algún punto intermedio entre ambos pasos, y a continuación, se interpolan Estados intermedios, y en cada uno de ellos se calcula el valor de la función activadora.

El resultado es conseguir un intervalo de tiempo $[t_{low}, t_{high}]$ dentro del cual no se sabe cuando ha ocurrido un evento, pero se sabe que ha ocurrido uno. La amplitud de este intervalo es configurable, y una menor amplitud ocasionará una localización del evento más precisa, pero también más lenta. Una vez conocido dicho intervalo, se llama a su *handler* de evento, y se le pasa el Estado que el sistema hubiera tenido en el instante t_{high} sin tener el evento en consideración. El *handler* se ocupa entonces de modificar el Estado como sea conveniente, y posteriormente se devuelve, partiendo del Estado modificado como punto de partida para la siguiente iteración.

Puede darse el caso de que existan eventos programados para ocurrir en un determinado instante de tiempo, conocido de antemano. El mecanismo anterior puede usarse, definiendo la función activadora como

$$\mathbf{e}(\mathbf{d}; t, \mathbf{y}) = \mathbf{e}(t) = t - t_{event} \tag{7}$$

pero eso es innecesariamente ineficiente, pues se tienen que interpolar varios Estados y acotar un intervalo temporal dentro del cual se sepa que ha ocurrido el evento, a pesar de que el instante en el que va a ocurrir es conocido. Simbody dispone de un mecanismo especial para este tipo de eventos, los cuales denomina eventos programados (*scheduled events*), a diferencia de los anteriormente vistos, los eventos activados (*triggered events*). Existe otro caso especial en el que un evento no modifica el Estado, sino que solo lo examina o monitorea. Si se conoce que el Estado no va a ser modificado, se puede ahorrar recursos y tiempo en la simulación, implementando otro mecanismo especifico: *handlers* especiales conocidos como *event reporters*.

Los eventos, como muchas otras características de un Sistema, son definidos por Subsistemas, pero no es necesario escribir un nuevo Subsistema desde cero para crear un evento. Un Subsistema especial: *DefaultSystemSubsystem*, proporciona un mecanismo para definir nuevos eventos: se crea una clase nueva derivada de unas clases base de *handlers* y *reporters*, y se añaden al Sistema mediante métodos de clase. En la siguiente tabla se muestran dichas clases y funciones:

	Event handler	**Event reporter**
Base classes	TriggeredEventHandler ScheduledEventHandler	TriggeredEventReporter ScheduledEventReporter
Function	addEventHandler()	addEventReporter()

Tabla 2. Clases y funciones para añadir eventos en Simbody. Para definir un evento, se crea una clase derivada de una de las que se muestran en la tabla y se añade al Sistema con la función correspondiente.

Las clases *PeriodicEventHandler* y *PeriodicEventReporter* permiten definir eventos programados que además son ejecutados en intervalos de tiempo regulares, y derivan de las clases de la tabla anterior, que a su vez derivan de las clases abstractas *EventHandler* y *EventReporter*.

3.2.5. Time Stepping

Una simulación normalmente implica resolver la trayectoria de un sistema a lo largo del tiempo. Para ello se parte de un punto conocido de la trayectoria (que en Simbody se corresponde con un Estado). A este punto se le llama la “condición inicial y a partir de ella se desarrolla la trayectoria de acuerdo con un modelo matemático. Este desarrollo es calculado elaborando una secuencia de pasos, en la cual el Estado obtenido al final de cada paso es utilizado como condición inicial al inicio del siguiente paso. Este proceso es conocido como *time stepping*, y el intervalo temporal usado es conocido normalmente como paso o *time step*.

La integración numérica es crucial dentro del *time stepping* si se desea que la trayectoria final sea lo más continua y "suave" posible. Sin embargo, los modelos matemáticos pueden contar tanto con ecuaciones continuas como discretas. Cuando disponen de ambas, los sistemas que modelan suelen ser denominados "sistemas híbridos". Cuando se tratan este tipo de sistemas, el *time stepper* se encarga de hacer avanzar el sistema a través del tiempo de acuerdo con su modelo matemático; y el integrador numérico se encarga tanto de hacer avanzar la trayectoria en intervalos donde solo cambie la componente continua del modelo, como de detectar eventos que indiquen cambios en la componente discreta. El integrador devuelve el control al *time stepper* cuando se detecta un evento o cuando se alcanzan instantes de tiempo específicos, conocidos como *report times*.

El integrador debe conseguir un nivel de precisión especificado por el usuario durante cada intervalo continuo, escogiendo un tamaño de paso interno apropiado. En un intervalo, el integrador puede tomar muchos pasos internos o puede tomar solo uno que incluso supere el próximo *report time*. En el segundo caso, se devuelve un Estado interpolado suficientemente preciso al *time stepper*, siendo posible incluso que un solo paso interno cruce varios *report times*.

Cuando un evento es detectado, el integrador determina en que instante sucede, completa el intervalo de tiempo actual y devuelve el control al *time stepper*. A continuación, el *time stepper* llama al *handler* de evento correspondiente, que en general, efectuará cambios tanto en variables discretas y continuas. Estas modificaciones debidas a los eventos producen un conjunto de condiciones iniciales para el siguiente paso diferentes a las resultantes en el paso anterior. Normalmente los intervalos continuos son largos y los eventos relativamente raros, pero es posible definir un sistema que avance en el tiempo a base de eventos discretos.

Un ejemplo de todo el proceso de *time stepping* e integración numérica puede verse en la siguiente figura, que será discutido a continuación.

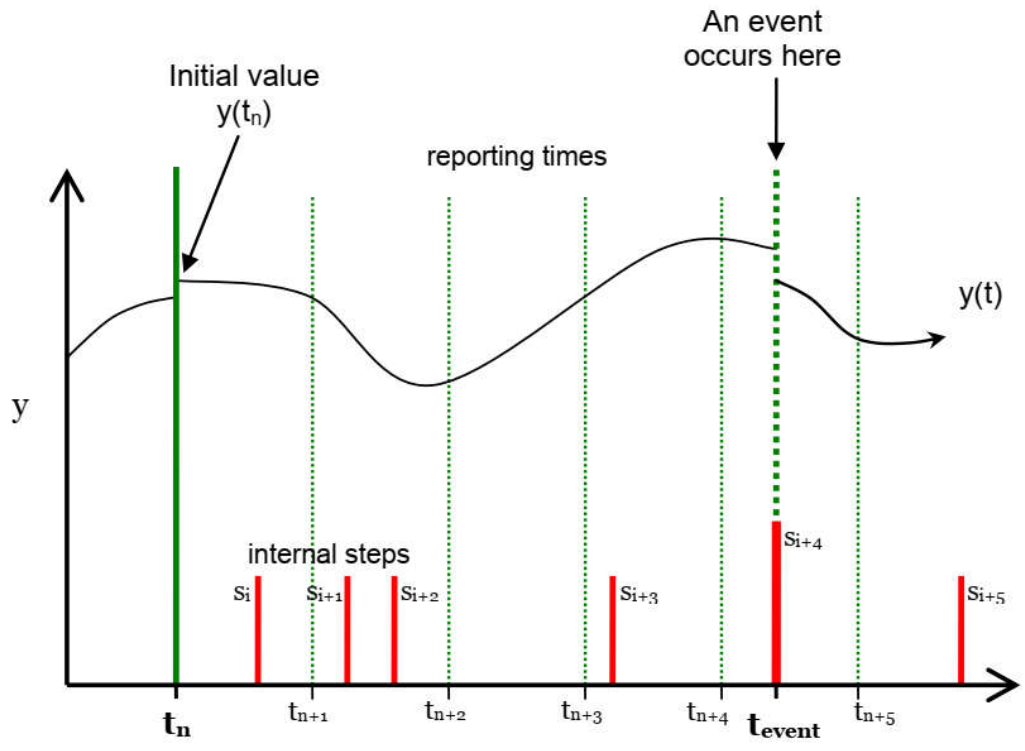

Figura 9. Integración numérica sobre un segmento continuo de la trayectoria. (Fuente: Guía de usuario de Simmath *[21]*)

En el eje horizontal se representa el tiempo (t), y en el eje vertical, los hipotéticos valores de la trayectoria ($\mathbf{y}$).

- La línea curva de color negro representa la trayectoria del sistema en cada instante ($\mathbf{y}(t)$).
- Las líneas altas de color verde representan retornos del control al time stepper.
    - La línea gruesa continua representa el inicio del segmento continuo (t_n).
    - La línea gruesa discontinua representa retorno del control no solicitado por el usuario debido a un evento, y el fin del segmento continuo (t_{event}).
    - Las líneas delgadas discontinuas representan *report times* solicitados por el usuario (t_{n+1}, t_{n+2}, t_{n+3}, t_{n+4}, t_{n+5}).
- Las líneas cortas de color rojo representan los pasos internos realizados por el integrador numérico (s_i, s_{i+1}, s_{i+2}, s_{i+3}, s_{i+4}, s_{i+5}).

El proceso comenzaría partiendo de la condición inicial $\mathbf{y}(t_n)$, donde el *time stepper* inicia la integración y establece un *report time* t_{n+1}. Para alcanzar la precisión requerida, el integrador realiza dos pasos internos, s_i y s_{i+1}, y devuelve control al *time stepper* en el instante t_{n+1} con un Estado interpolado, ya que el último paso sobrepasó t_{n+1}. El integrador no pierde su estado interno entre llamadas, por lo que retoma el proceso donde lo dejó y realiza otros dos pasos internos, s_{i+2} y s_{i+3}, devolviendo el control con un Estado interpolado para t_{n+2}. Cuando recibe de nuevo el control, lo devuelve inmediatamente sin tomar más pasos internos, ya que el último también sobrepasó t_{n+3}.

Cuando el *time stepper* establece un *report time* t_{n+4}, el integrador realiza un paso interno s_{i+4}, pero se detecta que ocurrirá un evento en el instante t_{event}. A partir de aquí, se devuelve control al *time stepper* con un Estado interpolado para t_{n+4}, que encuentra antes que el evento. El *time stepper* establece un *report time* t_{n+5}, pero el integrador le devuelve el control prematuramente en el instante t_{event}, informándole que un evento está a punto de ocurrir (pero todavía no ha sucedido). Entonces el *time stepper* hace una llamada al *handler* de evento correspondiente para producir un Estado modificado, rompiendo la continuidad de la trayectoria en el instante t_{event}. El integrador toma el nuevo estado, se reinicia y toma un nuevo paso interno s_{i+5} para alcanzar el *report time* t_{n+5}, recordando que ya había sido establecido antes de que sucediera el evento.

Se puede considerar que los *report times* no producen impacto en la evolución del sistema. Un integrador tomará la misma secuencia de pasos sin importar el número de *report times*. En la práctica, esto no es del todo cierto, ya que el integrador puede "desplazar" ligeramente sus pasos internos cuando un paso se encuentra cerca de un *report time*, para no tener que interpolar para instantes muy próximos entre sí. No obstante, el cálculo requerido para cada paso interno es relativamente costoso en comparación con las interpolaciones necesarias para satisfacer un *report time* concreto, por lo que se puede decir que la evolución del sistema estará determinada principalmente por los requerimientos de precisión y por la ocurrencia de eventos.

Un paso interno constituye un avance irreversible en la solución continua del problema. Por ello, el integrador tomará antes pasos internos de prueba que serán rechazados si no cumplen con la precisión requerida. Una consecuencia es que, desde el punto de vista del objeto Sistema (que es llamado en cada paso, incluso en los de prueba) el tiempo avanza y retrocede hasta que avanza de manera definitiva. Como consecuencia, el Sistema no depende de que el tiempo aumente de manera monótonamente creciente. Por otra parte, el integrador solo recuerda un pequeño margen de la trayectoria pasada, suficiente para interpolar Estados necesarios en *reporting times*, o para predecir la trayectoria futura a corto plazo. Una copia de la trayectoria puede guardarse en un archivo o reflejarse en una variable auxiliar del Estado. En cualquier caso, el resto de la trayectoria es olvidada por el integrador, al no ser necesaria.

3.2.6. Sistema de coordenadas

Un sistema de coordenadas, sistema coordenado o sistema de referencia es un conjunto compuesto de un punto llamado origen, y de tres direcciones ortogonales entre sí, llamadas ejes. El sistema de coordenadas se denota como F, el origen se denota como F_O y los ejes son denotados como F_x, F_y y F_z y siguen la regla de la mano derecha, por lo que $F_z = F_y \times F_x$.

La función de los sistemas de coordenadas es ubicar y medir cosas en ellos. Por ejemplo, la localización de un punto P se puede expresar midiendo un vector de posición $\boldsymbol{r}$, desde F_O hasta P, siendo $\boldsymbol{r} = P - F_O$, y descomponiendo $\boldsymbol{r}$ en cada una de las direcciones de los ejes. Al hacer esto, se obtienen unos valores numéricos llamados *measure numbers*, que comúnmente son denominados "coordenadas". Las coordenadas de $\boldsymbol{r}$ en el sistema de referencia F se denotan de la siguiente manera:

$$^{F}\boldsymbol{r} = (r_x, r_y, r_z) = (\boldsymbol{r} \cdot F_x, \boldsymbol{r} \cdot F_y, \boldsymbol{r} \cdot F_z) \tag{8}$$

Como puede verse en la ecuación anterior, las coordenadas de un vector pueden obtenerse realizando el producto escalar del vector con cada uno de los ejes del sistema de coordenadas. La siguiente figura ilustra el ejemplo anterior y muestra la notación usada:

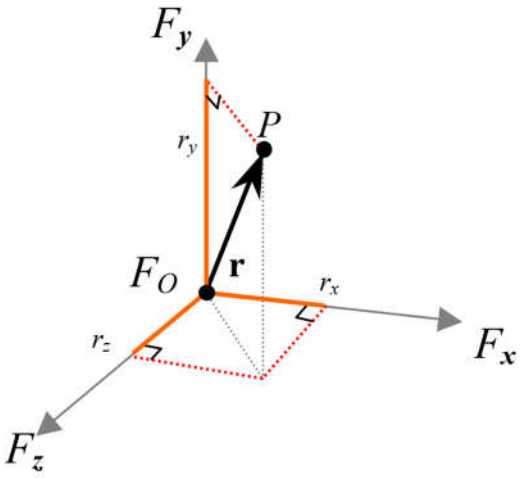

Figura 10. Sistema de coordenadas *F* y localización de un punto *P*. (Fuente: Manual de teoría de Simbody *[21]*)

Cabe destacar que, mientras $\boldsymbol{r}$ es una magnitud única e inconfundible (el vector que va desde F_O hasta P), sus coordenadas pueden ser diferentes dependiendo del sistema de coordenadas en el que sean expresadas. Por este motivo, se emplea la notación $^{F}[Q]$ para indicar que una magnitud Q es expresada en un sistema de coordenadas F cuando el sistema usado pueda no ser obvio.

Por otra parte, el concepto de sistema de coordenadas tiene sentido incluso si no se tiene referencia de donde está situado o a dónde apuntan sus ejes. Una vez definido F, se puede comenzar a medir magnitudes en él, incluso el propio F: sus ejes (**ecuaciones 9-11**) y su origen (**ecuación 12**). En ese sentido, F constituye su propio "universo" autoconsistente sin necesidad de ninguna otra referencia. Este universo se extiende infinitamente en toda dirección, y en mecánica de SMC y robótica, recibe el nombre de cuerpo o eslabón (*body*, *link*).

$$^{F}[F_x] = (1,0,0) \quad (9)$$

$$^{F}[F_y] = (0,1,0) \quad (10)$$

$$^{F}[F_z] = (0,0,1) \quad (11)$$

$$^{F}[F_O] = (0,0,0) \quad (12)$$

3.2.7. Cuerpos

Un cuerpo es, en esencia, un sistema de coordenadas en movimiento. Como consecuencia, un cuerpo se extiende infinitamente en todas direcciones, y aunque resulte contraintuitivo, es una afirmación muy conveniente cuando se comience a conectar cuerpos entre sí.

Para un SMC denotado B, el cuerpo i-ésimo que lo compone será denotado como $B[i] \in B$. Normalmente se trabaja en una simulación con pocos cuerpos, por lo que se puede evitar esta notación utilizando diferentes letras, y en especial, se denota como G al cuerpo *Ground*, que en ocasiones es denominado "tierra". Este *Ground* proporciona un sistema de referencia inercial (sin aceleración lineal ni rotaciones) de origen G_O (que simplemente se denota como O) y ejes $\boldsymbol{x} \triangleq G_x$, $\boldsymbol{y} \triangleq G_y$, $\boldsymbol{z} \triangleq G_z$; y por convención: $B[0] = G$.

Los cuerpos tienen asociadas magnitudes que pueden ser medidas y expresadas en su propio sistema de referencia, como otros sistemas de referencia, vectores, puntos o escalares. Todos los cuerpos cuentan con ciertas magnitudes especificas: su propio sistema de coordenadas, como ya evidencia la definición de cuerpo, con ejes y origen que puede expresar respecto a si mismo, como se ve en las **ecuaciones 9-12**; y cuenta con lo que se denominan "propiedades másicas" (representada en Simbody con el objeto *MassProperties*), que incluye la masa total del sistema (escalar), el centro de masas del sistema (punto, representado por un vector) y su tensor de inercia (matriz 3×3 simétrica), la inercia rotacional del sistema, que se denomina además "central" cuando se expresa en torno a su centro de masas.

En los cuerpos rígidos las propiedades másicas son constantes, y en los cuerpos deformables (actualmente no soportados por Simbody) el centro de masas y el tensor de inercia pueden variar si son medidos en el sistema de coordenadas del cuerpo. Por motivos prácticos, cada cuerpo tiene asignada una propiedad llamada "estructura másica" (que en Simbody engloba la clase *Body*), cuyo tipo puede ser uno de los siguientes:

- *Ground*: Representación de un objeto inmóvil, de masa e inercia infinita a efectos prácticos. Su clase en Simbody no puede editarse para ser otra cosa.
- *Massless*: Objeto sin masa o inercia (valor cero), cuya clase tampoco puede editarse para tener una masa o inercia.
- *Particle*: Define un cuerpo de masa puntual, sin inercia, con centro de masas en el origen de su sistema de referencia local. Su clase puede ser editada para tener cierta masa o ninguna, pero su centro de masas no puede ser movido ni puede tener inercia.
- *Linear*: Cuerpo en forma de línea que no tiene inercia a lo largo de su eje. Las propiedades másicas de su clase pueden ser editadas, pero solo de forma que la inercia a lo largo de su eje siga siendo nula.
- *Rigid*: Un cuerpo rígido general. Su clase es totalmente editable.
- *Deformable*: Representa cuerpos con coordenadas de deformación interna, pero no es una clase definida por Simbody, por lo que el usuario debe determinar la manera de definir cuerpos deformables.

3.2.8. Movilizadores

Los movilizadores conectan un cuerpo a su cuerpo padre y proporcionan movilidad entre esos dos cuerpos, que debe entenderse como grados de libertad (GDLs). Esta movilidad expresa los movimientos permitidos entre el sistema de referencia de un cuerpo con el de otro, y no debe confundirse con la idea de movimientos restringidos: en Simbody los cuerpos inicialmente no tienen movilidad, estando ambos sistemas de referencia bloqueados entre sí permanentemente. No hay movimiento que restringir, y los movilizadores son quienes otorgan ese movimiento relativo entre un cuerpo y su padre (translaciones o rotaciones) y permiten parametrizar dichos movimientos. El cuerpo *Ground*, que tiene todo SMC en Simbody, es el único que no tiene cuerpo padre ni tampoco movilidad.

El concepto de articulación o unión es más general y abstracto que el de movilizador. Es usado para referirse al concepto físico de articulación del mundo real, que combina movilizadores y restricciones e incluye elementos como fricción y amortiguación. Por esto, es posible crear un bucle cerrado con articulaciones, pero no con movilizadores, pues los movilizadores solo pueden añadir GDLs entre un cuerpo y su padre, mientras que las articulaciones pueden añadirlos o eliminarlos entre todos los cuerpos que formen dicha articulación. Un ejemplo puede verse en la siguiente figura:

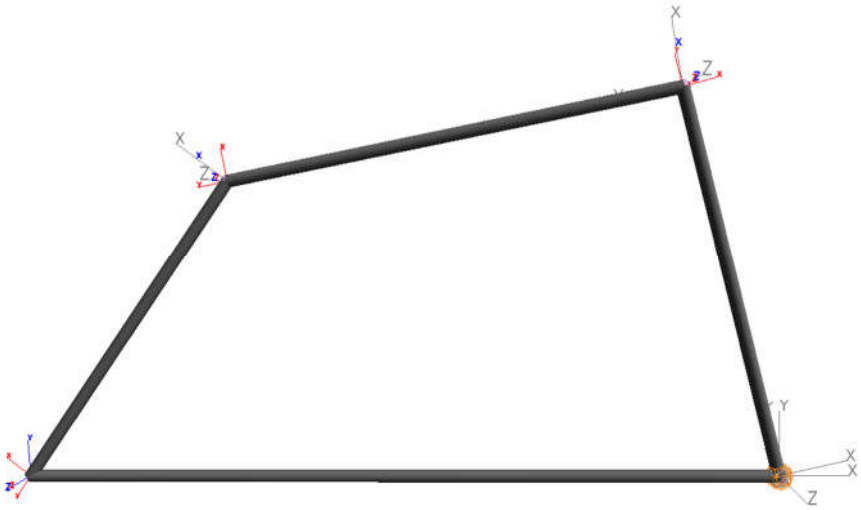

Figura 11. Ejemplo de mecanismo de cuatro barras cerrado en Simbody. Está formado además por cuatro articulaciones, de las cuales solo tres pueden ser movilizadores sin seccionar una de las barras.

Un movilizador puede ser un tipo de articulación, pero no todas las articulaciones son movilizadores. Cuando una articulación forma un bucle cerrado como en el mecanismo de la **Figura 11**, la movilidad total del sistema se reduce en lugar de incrementar, lo que requeriría de una restricción en lugar de un movilizador. Mientras un mecanismo real es descrito inequívocamente en función de sus cuerpos y articulaciones, en Simbody hay varias maneras de describir el mismo sistema. Volviendo a la **Figura 11**, se podría seccionar uno de los cuerpos, construir las dos ramas del mecanismo y unir las dos mitades del cuerpo seccionado con una restricción de soldadura; o bien, separar una de las articulaciones, construir el mecanismo completo abierto y cerrar el extremo libre con una restricción de punto fijo. La primera opción requeriría dividir proporcionalmente las propiedades másicas del cuerpo seccionado, pero las articulaciones estarían definidas con movilizadores, mientras que, en el segundo caso, el movimiento de una de las articulaciones no puede parametrizarse, al no estar modelada con un movilizador.

Los movilizadores más comunes están basados en tres tipos de movimientos: deslizamiento, torsión y orientación:

- Deslizamiento: permite translación a lo largo de un eje (un GDL). Añade una coordenada generalizada con unidades de longitud.
- Torsión: permite rotación a lo largo de un eje (un GDL). Añade una coordenada generalizada con unidades angulares.
- Orientación: permite cambio de orientación entre los dos cuerpos (tres GDLs). Añade tres coordenadas generalizadas (que requieren un cuaternión para la dinámica).

La mayor parte de los movilizadores son una combinación de estos tipos de movimientos: un movilizador cartesiano es una composición de deslizamientos para los tres ejes (tres GDLs), mientras que un movilizador libre es la composición de esos tres deslizamientos y de orientación (seis GDLs).

Uniones más complejas pueden construirse empleando movilizadores y restricciones (apartado **3.2.9**), como una unión roscada, que puede modelarse con un deslizamiento y torsión coaxiales, más una restricción que relacione ambos movimientos de manera que simule el paso de la rosca. Esta unión añadiría dos coordenadas generalizadas (dos GDLs) y una restricción. Sin embargo, algunas uniones aparentemente complejas pueden modelarse sin el uso de restricciones, aprovechando los conceptos de movilizador y de cuerpo que se entiende infinitamente en todas direcciones. De hecho, Simbody dispone de un movilizador de unión roscada con una sola coordenada generalizada y cero restricciones.

Un cuerpo puede disponer de entre cero y seis movilidades (es decir, GDLs) respecto a su cuerpo padre. Todas las movilidades de todos los cuerpos en un SMC definen su espacio móvil. Este espacio se debe parametrizar para expresar cualquier configuración posible del sistema, y no existe una única forma para conseguirlo. Por suerte, las movilidades de cada cuerpo son mutuamente independientes y cada cuerpo se puede parametrizar por separado. De esta manera, el conjunto de los parámetros de todos los cuerpos conforma la parametrización completa del sistema.

Esta independencia de movilidades delimita el problema a la parametrización de cada cuerpo. Como consecuencia, cada movilizador define dos valores escalares por cada movilidad: uno para expresar su posición relativa y otro para expresar su velocidad relativa. Los primeros son conocidos como coordenadas generalizadas, y los segundos como velocidades generalizadas. Como ya fue comentado en el apartado **3.2.2**, el vector de coordenadas generalizadas es denotado como $\mathbf{q}$, y el vector de velocidades generalizadas, como $\boldsymbol{u}$.

En Simbody, el número de velocidades generalizadas de un cuerpo coincide siempre con el número de movilidades del propio cuerpo. Cada velocidad generalizada tiene, por tanto, independencia de las demás y significado físico que se puede interpretar. Las ecuaciones de movimiento están expresadas en función de derivadas de $\boldsymbol{u}$, denotadas como $\dot{\boldsymbol{u}}$, y son referidas como aceleraciones generalizadas. Por otra parte, esto no se cumple con las coordenadas generalizadas, que a veces son elegidas por su conveniencia computacional, por lo que, en general: $\dot{\mathbf{q}} \neq \boldsymbol{u}$. Un ejemplo son los movilizadores que permiten orientación completa, que usan cuatro coordenadas generalizadas para los componentes de un cuaternión mientras que solo usan tres velocidades generalizadas para cada una de las componentes del vector de velocidad angular relativa. Dicho cuaternión debe cumplir una restricción de modulo unitario, lo que deja 3 GDLs finales.

3.2.9. Restricciones

El espacio móvil de un SMC representa una pequeña parte de todas las posibles disposiciones de los cuerpos que lo conforman en el espacio. Este espacio reducido será, en la mayoría de los casos, sustancialmente más grande que el espacio de soluciones del sistema. En el caso de la **Figura 11**, el espacio móvil incluirá configuraciones en las que el mecanismo ni siquiera forme un bucle cerrado. Este espacio de soluciones tendrá como máximo, el mismo número de GDLs que el espacio móvil, aunque frecuentemente su número será menor.

Es posible redefinir la movilidad de los cuerpos para que el espacio móvil coincida con el espacio de soluciones, pero de esa manera, se pierden parámetros que pueden ser relevantes. En su lugar, resulta más apropiado construir un sistema con un número de movilidades sencillo de definir y posteriormente añadir un conjunto de restricciones cuyo cumplimiento implícitamente defina el espacio de soluciones.

Una restricción genera una o más ecuaciones de restricción, cada una de las cuales resta un GDL al sistema. En este sentido, las restricciones son complementarias a los movilizadores, cuyas velocidades generalizadas suman cada una un GDL al sistema. De hecho, cualquier movilizador de *n* GDLs puede representarse como un movilizador libre con *6-n* ecuaciones de restricción.

Las restricciones actúan sobre un sistema introduciendo fuerzas internas. Estas fuerzas actúan exactamente igual que aquellas aplicadas externamente, con la salvedad de que resultan desconocidas, y deben ser calculadas a la par que las aceleraciones generalizadas.

3.2.10. Fuerzas

En Simbody, el concepto de fuerza es más similar al concepto de carga, ya que incluye tanto fuerzas como momentos. Estas fuerzas pueden actuar sobre los cuerpos del sistema o bien sobre las mismas velocidades generalizadas. Cuando esto último ocurre, las fuerzas son escalares en lugar de vectoriales y son llamadas fuerzas generalizadas o fuerzas de movilidad. Todas las fuerzas y momentos en un sistema pueden ser reducidas a fuerzas generalizadas, y Simbody aporta un operador para realizar esta conversión de manera eficiente.

Las fuerzas pueden ser funciones del tiempo, posición, velocidad o de ellas mismas. También pueden ser puntuales, ser el resultado de campos de fuerzas distribuidos o constantes, o bien actuar por pares en dos puntos distintos del sistema. Aquellas que no dependen del tiempo ni de la velocidad son llamadas fuerzas conservativas, y son el gradiente de alguna función de energía potencial. Las fuerzas no conservativas, por otro lado, dependen del tiempo y de la velocidad y pueden no contribuir a la energía potencial del sistema.

3.2.11. Cinemática

La cinemática puede definirse como el estudio del movimiento de un sistema sin tener en cuenta masas ni fuerzas. En la práctica, consiste en hallar la correlación entre las coordenadas y velocidades generalizadas con las posiciones, velocidades y aceleraciones cartesianas de cada cuerpo. Las magnitudes generalizadas determinan de manera inequívoca las magnitudes cartesianas, por lo que la obtención de las segundas a partir de las primeras es rápida e inmediata, denominándose cinemática directa.

- Conocido $\mathbf{q}$, se saben inmediatamente las posiciones de los cuerpos.
- Conocidos $\mathbf{q}$ y $\boldsymbol{u}$, se conoce cómo se están moviendo los cuerpos.
- Conocidos $\mathbf{q}$, $\boldsymbol{u}$ y $\dot{\boldsymbol{u}}$, se conoce cómo están reaccionando (acelerando) los cuerpos.

El proceso inverso, cuando se intenta determinar las magnitudes generalizadas a partir de las cartesianas, es conocido como cinemática inversa y es más complicado si todos los cuerpos no disponen de 6 GDLs (movilidad no restringida). El objetivo de la cinemática inversa es, dado un conjunto de valores observados para las magnitudes cartesianas, encontrar los valores de las magnitudes generalizadas que mejor se ajusten a las observaciones y ecuaciones de restricción del sistema. Simbody dispone de la clase *Assembler* para solucionar muchos de los problemas comunes de cinemática inversa, que deben ser realizados antes de estudiar la dinámica.

3.2.12. Dinámica

En contraste con la cinemática, la dinámica estudia las relaciones entre fuerzas y aceleraciones para unos valores fijos de $\mathbf{q}$ y $\boldsymbol{u}$. Estas relaciones están determinadas por la segunda ley de Newton:

$$f = ma \tag{13}$$

El objetivo de la dinámica directa es calcular aceleraciones y fuerzas internas de restricción a partir de un conjunto dado de fuerzas generalizadas f, mientras que el objetivo de la dinámica inversa es obtener el conjunto de fuerzas generalizadas que mejor explique un conjunto conocido de aceleraciones generalizadas.

Avanzar un Estado a través del tiempo para producir una trayectoria o utilizar variables de estado para obtener información particular son Estudios de alto nivel que son posibles gracias a estas definiciones de dinámica y a las capacidades de Simbody. Las clases básicas para realizar dichos estudios son *Integrator* y *TimeStepper*, ya mencionadas en el apartado **3.2.5**.

3.2.13. Tipos de datos numéricos

Simbody cuenta con definiciones para varios entes matemáticos cuya notación y clases en Simbody serán explicadas en este apartado.

Los primeros y más comunes son los vectores y las matrices, definidas en el paquete Simmatrix, que a su vez forma parte de la librería SimTKcommon, y esta última parte de Simbody. Simmatrix decide llamar *Vector* y *Matrix* a objetos grandes de tamaño variable, mientras que, por ejemplo, *Vec3* y *Mat33* hacen referencia a objetos pequeños de tamaño fijo (un vector de tres elementos y una matriz 3×3, respectivamente). Estos objetos serán las clases base a partir de las cuales se definirán todos los siguientes.

A nivel geométrico, Simbody dispone de varios objetos que, si bien no pretenden conformar un paquete de recursos genérico, serán de utilidad teniendo en cuenta que principalmente se trabajará con geometría en tres dimensiones.

La primera de las clases hace referencia a puntos, los cuales Simbody denota como *stations*, ya que permanecen estacionarios dentro de un sistema de referencia concreto. Por este motivo, los puntos serán determinados por el vector posición dentro del sistema de referencia empleado, utilizando un*Vec3* (no existe una clase *Station*).

Por otro lado, también están los vectores unitarios o direcciones (*directions*), que son definidos utilizando la clase *UnitVec3*, idéntica a *Vec3* pero que asegura que sus elementos formen un vector de longitud uno. De esa forma, se evita revisar que los vectores empleados en los cálculos estén normalizados y se evitan normalizaciones innecesarias, que son operaciones costosas.

Las rotaciones en tres dimensiones se pueden expresar de muchas maneras distintas, con sus propias peculiaridades, pero todas son equivalentes a una matriz 3×3 denominada matriz de rotación, matriz de orientación o matriz de cosenos directores. Esta matriz tiene la característica de que sus columnas son direcciones perpendiculares entre sí, que representan los ejes de un sistema de coordenadas expresadas respecto a los ejes de otro. Está definida en el objeto *Rotation*, idéntica a un *Mat33* pero que dispone de métodos para construir la propia matriz siguiendo diferentes esquemas y asegurando ciertas propiedades:

- Cada columna y fila forma un vector unitario.
- Todas las columnas son perpendiculares entre sí, y todas las filas son perpendiculares entre sí.
- Las filas y columnas respetan la regla de la mano derecha, por lo que la tercera fila es el producto escalar positivo de las dos primeras y lo mismo ocurre con las columnas.
- Como consecuencia, es una matriz ortogonal, su transpuesta es igual a su inversa y su determinante es +1.

Se denota como ${}^{from}R^{to}$ la matriz rotación que representa la orientación del sistema de coordenadas "*to"* medido desde el sistema de coordenadas "*from"*, como en el siguiente ejemplo:

$$ {}^{G}R^{B} \equiv \left({}^{G}[B_x] \ {}^{G}\left[B_y\right] \ {}^{G}[B_z] \right) \tag{14} $$

Donde hay que recordar que B_x, B_y y B_z son las coordenadas de los ejes $\mathbf{x}$, $\mathbf{y}$ y $\mathbf{z}$ del sistema de referencia B expresadas en el sistema de referencia G y, por tanto, cada uno de ellos es un vector de tres elementos, que conforman una matriz 3×3.

Dicha notación debe usarse siempre escribiendo los superíndices para evitar confusiones. Como la matriz de rotación es ortogonal, su traspuesta (representada en Simmatrix con el operador "~") será igual a su inversa, y como consecuencia, se invierten los superíndices, como representa la siguiente ecuación:

$$\sim {}^{G}R^{B} = {}^{B}R^{G} \tag{15}$$

Estas matrices se pueden usar para obtener las coordenadas de un vector expresadas en un sistema de referencia en el otro, de la siguiente manera:

$${}^{G}v = {}^{G}R^{B}\ {}^{B}v \tag{16}$$

$${}^{B}v = {}^{B}R^{G}\ {}^{G}v \tag{17}$$

En Simmatrix, la transposición de una matriz es entendido como un cambio en el punto de vista, y no involucra cálculos ni duplicado de datos. Como consecuencia, ambos términos de la **ecuación 15** podrían utilizarse en la **ecuación 17** con la misma eficiencia.

Otra clase de Simbody es *Transform*, usada para definir matrices de transformación lineal, que combinan rotaciones con además traslaciones. Un ejemplo de matriz de transformación lineal está constituida de la siguiente manera:

$${}^{G}X^{B} \triangleq \begin{pmatrix} {}^{G}[B_x] & {}^{G}[B_y] & {}^{G}[B_z] & {}^{G}p^{B} \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{18}$$

El término ${}^{G}p^{B}$ sería el vector que va desde el origen del sistema de referencia G hasta el origen del sistema de referencia B, un vector de traslación. La notación es similar a las de matrices de rotación, incluso puede notarse que una matriz de transformación lineal simplemente se trata de una matriz de rotación, seguida de un vector de posición que señala el origen del sistema y una fila extra, que siempre tiene los valores que se muestran:

$${}^{G}X^{B} \equiv \begin{pmatrix} \begin{pmatrix} & {}^{G}R^{B} & \end{pmatrix} & \begin{pmatrix} {}^{G}p^{B} \end{pmatrix} \\ (0 \quad 0 \quad 0 & 1\ ) \end{pmatrix} \tag{19}$$

Un objeto *Transform* se puede utilizar como si fuera una matriz 4×4, pero Simbody trabaja en memoria con él como si fuera una matriz 3×4, es decir, almacena la matriz de rotación y el vector de traslación, y desecha la última fila, al ser siempre igual. También se puede trabajar con su matriz de rotación y su vector de traslación individualmente, sin necesidad de realizar copias.

La matriz de transformación lineal, por la manera en que ha sido definida en la **ecuación 18**, no es ortogonal. Sin embargo, su inversa, como ocurría con la matriz de rotación, es fácilmente aplicable, sin necesidad de cálculos adicionales. Simmatrix tiene sobrecargado el operador de trasposición de matrices ("~") para interpretar un objeto *Transform* como su matriz inversa, sin coste computacional adicional. Un ejemplo de su uso aparece en la siguiente ecuación:

$${}^{B}X^{C} = \sim {}^{G}X^{B}\,{}^{G}X^{C} \tag{20}$$

Simbody cuenta no solo con clases geométricas sino también con clases destinadas al cálculo de la mecánica de un sistema.

Entre las propiedades másicas de los cuerpos (apartado **3.2.7**), se encontraban la masa total, definida con un simple escalar; el centro de masas, definido con un vector de tres elementos; y la matriz de inercia, que se define con un tensor, una matriz 3×3 que exhibe ciertas propiedades características, entre ellas, que es simétrica. Simbody dispone de la clase *Inertia*, que se almacena en memoria exactamente igual que la clase *SymMat33* de Simmatrix: conteniendo solo seis elementos distintos. Un objeto *Inertia* se comporta como una matriz ordinaria en operaciones de lectura, pero se construye y asigna atendiendo a conceptos con significado físico (por ejemplo, dadas las inercias respecto a cada eje) y dispone de ciertas funciones especiales, como expresar la matriz de inercia en otros sistemas de referencia. Por comodidad, este objeto *Inertia* es almacenado junto con un escalar y un *Vec3* en el objeto *MassProperties*, que implícitamente dispone de un sistema de referencia en el que están expresados y medidos el centro de masas y la distribución de inercia.

Otras clases útiles en mecánica son ciertos vectores y matrices ampliados que combinan magnitudes lineales y angulares en un solo objeto. Esta notación es llamada en Simbody "notación espacial" ("*spatial notation*"), y dichos objetos son definidos por Simmatrix como *SpatialVec* y *SpatialMat*, y son, respectivamente, dos vectores de tres elementos apilados y una matriz 2×2 de matrices 3×3.

Respecto a los vectores espaciales, el primer subvector es siempre el de componentes angulares y el segundo es siempre el de componentes lineales. Unos cuantos ejemplos de estos vectores espaciales serían la velocidad espacial V, la aceleración espacial A y la fuerza espacial F:

$$V \triangleq \begin{pmatrix} \omega \\ v \end{pmatrix} \tag{21}$$

$$A \triangleq \begin{pmatrix} \beta \\ a \end{pmatrix} \tag{22}$$

$$F \triangleq \begin{pmatrix} \mu \\ f \end{pmatrix} \tag{23}$$

Donde ω es una velocidad angular, v un vector de velocidad lineal, β un vector de aceleración angular, a un vector de aceleración lineal, μ un momento y f una fuerza, todos vectores de tres elementos y medidos respecto del sistema de referencia de tierra G salvo que se indique lo contrario. Nótese que no existe un vector de "posición espacial", debido a que la orientación no es una magnitud vectorial. En su lugar, se usa la clase *Transform* para posición espacial.

Para cualquier magnitud vectorial v, se emplea la notación $\mathrm{v}_\times$ para denotar la matriz tal que, dado cualquier otro vector w, se cumpla que $\mathrm{v}_\times\,\mathrm{w} = \mathrm{v} \times \mathrm{w}$:

$$\mathrm{v}_\times = \begin{pmatrix} 0 & -\mathrm{v}_z & \mathrm{v}_y \\ \mathrm{v}_z & 0 & -\mathrm{v}_x \\ -\mathrm{v}_y & \mathrm{v}_x & 0 \end{pmatrix} \tag{24}$$

Nótese que dicha matriz es antisimétrica y, por tanto, $v_\times^T = -\mathrm{v}_\times$. Usando esta notación, pueden encontrarse varias identidades usadas a menudo en Simbody, reflejadas en las **ecuaciones 25-27**.

$$(\mathrm{v}+\mathrm{w})_\times = \mathrm{v}_\times + \mathrm{w}_\times \tag{25}$$

$$\mathrm{v}_\times \mathrm{w}_\times = \mathrm{w}\mathrm{v}^\mathrm{T} - \mathrm{w}^\mathrm{T}\mathrm{v}1_3 \tag{26}$$

$$\mathrm{v}_\times^2 \triangleq \mathrm{v}_\times^\mathrm{T}\,\mathrm{v}_\times = -\mathrm{v}_\times\mathrm{v}_\times = \mathrm{v}^\mathrm{T}\mathrm{v}\mathbf{1}_3 - \mathrm{v}\mathrm{v}^\mathrm{T} \tag{27}$$

$$\mathrm{U}\mathrm{v}_\times\mathrm{U}^\mathrm{T} = (\mathrm{U}\mathrm{v})_\times \tag{28}$$

$$\dot{\mathrm{v}}_\times = (\dot{\mathrm{v}})_\times \tag{29}$$

En la **ecuación 26**, $\mathbf{1}_3$ representa una matriz identidad 3×3, y en la **ecuación 27**, $v_\times^T\,\mathrm{v}_\times$ es una matriz simétrica con diagonales de elementos no negativos con la siguiente estructura:

$$v_\times^2 \triangleq v_\times^T\,\mathrm{v}_\times = \begin{pmatrix} v_y^2 + v_z^2 & \mathrm{T} & \mathrm{T} \\ -\mathrm{v_x v_y} & v_x^2 + v_z^2 & \mathrm{T} \\ -\mathrm{v_x v_z} & -\mathrm{v_y v_z} & v_x^2 + v_y^2 \end{pmatrix} \tag{30}$$

Donde T indica el elemento en la misma oposición que su traspuesto. Esta expresión también puede verse como la matriz de inercia de una partícula de masa unitaria localizada en el vector v, medido desde el origen.

En la **ecuación 28**, U sería cualquier matriz ortogonal, y dado que las matrices de rotación son ortogonales, dicha ecuación será muy útil para expresar magnitudes espaciales en otros sistemas de referencia. Por último, en la **ecuación 29**, el punto superior representa la derivada respecto del tiempo en un sistema de referencia sobreentendido por el contexto, y el propósito de la expresión mostrada es simplemente permitir escribir $\dot{\mathrm{v}}_\times$ sin ningún tipo de ambigüedad.

Respecto a las matrices espaciales, las propiedades másicas comentadas anteriormente también pueden expresarse como una matriz espacial de inercia, de la siguiente forma:

$$M \triangleq m\begin{pmatrix} \boldsymbol{\mathcal{G}} & p_\times \\ -p_\times & \mathbf{1}_3 \end{pmatrix} \tag{31}$$

Donde m es la masa total, p es el vector posición del centro de masas y $\boldsymbol{\mathcal{G}}$ es el tensor de inercia de masa unitaria (también llamado matriz de giro), es decir, un tensor que contiene los momentos de inercia del área y cumple que $\boldsymbol{\mathcal{I}} = m\boldsymbol{\mathcal{G}}$, siendo $\boldsymbol{\mathcal{I}}$ el tensor de inercia.

La matriz espacial de inercia M_B de un cuerpo concreto B entorno a su origen B_O, sería:

$$M_B = m_B\begin{pmatrix} \boldsymbol{\mathcal{G}_B} & -{}^{B_O}p_\times^{B_C} \\ -{}^{B_O}p_\times^{B_C} & \mathbf{1}_3 \end{pmatrix} \tag{32}$$

Cabe destacar que, cuando las propiedades másicas son dadas respecto a su centro de masas B_C, resulta que $p = \vec{0}$ (vector nulo) y, por tanto:

$$M_B^{B_C} = m_B\begin{pmatrix} \boldsymbol{\mathcal{G}_B^{B_C}} & 0 \\ 0 & \mathbf{1}_3 \end{pmatrix} \tag{33}$$

Donde la matriz de giro central $\boldsymbol{\mathcal{G}_B^{B_C}}$, aplicando el teorema de Steiner y la identidad de la **ecuación 27**, sería:

$$\boldsymbol{\mathcal{G}_B^{B_C}} \triangleq \boldsymbol{\mathcal{G}_B} - \left(p^\mathrm{T}p\mathbf{1}_3 - pp^\mathrm{T}\right) = \boldsymbol{\mathcal{G}_B} - p_\times^\mathrm{T}p_\times = \boldsymbol{\mathcal{G}_B} - p_\times^2 \tag{34}$$

Si además la velocidad espacial V^C está referida al centro de masas, se puede definir otra magnitud espacial: el momento espacial de un cuerpo, también referido a su centro de masas:

$$P^C \equiv M^C V^C = m\begin{pmatrix} \boldsymbol{\mathcal{G}}^{\boldsymbol{C}} & 0 \\ 0 & \mathbf{1}_3 \end{pmatrix}\begin{pmatrix} \omega \\ v^C \end{pmatrix} = m\begin{pmatrix} \boldsymbol{\mathcal{G}}^{\boldsymbol{C}}\omega \\ v^C \end{pmatrix} \tag{35}$$

Donde puede apreciarse que el primer subvector contiene el momento angular del cuerpo y el segundo subvector contiene el momento lineal del cuerpo. En el caso más general y típico en el que el origen del cuerpo no coincida con su centro de masas, el momento espacial respecto de B_O será calculado como:

$$P \equiv MV = m\begin{pmatrix} \boldsymbol{\mathcal{G}} & p_\times \\ -p_\times & \mathbf{1}_3 \end{pmatrix}\begin{pmatrix} \omega \\ v^C \end{pmatrix} = m\begin{pmatrix} \boldsymbol{\mathcal{G}}\omega + p_\times v \\ v - p_\times \omega \end{pmatrix} = P^C + m\begin{pmatrix} p_\times v^C \\ 0 \end{pmatrix} \tag{36}$$

El momento espacial respecto a un punto genérico no coincide con aquel respecto al centro de masas, puesto que hay que sumar la componente del momento angular asociada al movimiento del propio centro de masas.

A partir del momento espacial, se puede calcular la energía cinética de un cuerpo:

$$KE = \frac{1}{2}V^{\mathrm{T}}MV = \frac{1}{2}V^{\mathrm{T}}P = \frac{1}{2}m\left(\omega^{\mathrm{T}}\boldsymbol{\mathcal{G}}\omega + v^2 + \rho\right) \tag{37}$$

$$\rho \equiv= (\omega^T p_\times v - v^T p_\times \omega) = 2\omega^T p_\times v \tag{38}$$

Se puede destacar que la energía cinética calculada es un escalar y que es independiente del punto al que esté referido el momento angular:

$$KE = \frac{1}{2}{V^C}^T M^C V^C = \frac{1}{2}{V^C}^T P^C = \frac{1}{2}m\left(\omega^{\mathrm{T}}\boldsymbol{\mathcal{G}}^{\boldsymbol{C}}\omega + {v^C}^2\right) \tag{39}$$

Equivalente a la **ecuación 37** sustituyendo $\boldsymbol{\mathcal{G}}^{\boldsymbol{C}} = \boldsymbol{\mathcal{G}} - p_\times^{\mathrm{T}} p_\times$ y $v^C = v - p_\times \omega$.

Por último, los objetos *Rotation* y *Transform* también tienen sus equivalentes espaciales, las matrices espaciales de rotación ${}^A\boldsymbol{R}^B$ (*spatial rotation*), desplazamiento ${}^Q\boldsymbol{S}^P$ (*spatial shift*) y transformación ${}^A\boldsymbol{X}^B$ (*spatial transform*):

$${}^A\boldsymbol{R}^B = \begin{pmatrix} {}^A R^B & 0 \\ 0 & {}^A R^B \end{pmatrix} \tag{40}$$

$${}^P\boldsymbol{S}^Q = \begin{pmatrix} \mathbf{1}_3 & 0 \\ -{}^P p_\times^Q & \mathbf{1}_3 \end{pmatrix} \tag{41}$$

$${}^A\boldsymbol{X}^B = {}^{A_O}\boldsymbol{S}^{B_O}\,{}^A\boldsymbol{R}^B = \begin{pmatrix} {}^A R^B & 0 \\ {}^{A_O}p_\times^{B_O}\,{}^A R^B & {}^A R^B \end{pmatrix} \tag{42}$$

Para rotar, desplazar o transformar vectores y matrices espaciales, se usarán estas matrices, sus inversas y sus duales (definidas más adelante). Intercambiar los superíndices producirá la inversa de cada una de las matrices, como puede verse en las **ecuaciones 43-45**.

$$ {}^{B}\boldsymbol{R}^{A} = \begin{pmatrix} {}^{B}R^{A} & 0 \\ 0 & {}^{B}R^{A} \end{pmatrix} = ({}^{A}\boldsymbol{R}^{B})^{\mathrm{T}} = ({}^{A}\boldsymbol{R}^{B})^{-1} \tag{43} $$

$$ {}^{Q}\boldsymbol{S}^{P} = \begin{pmatrix} \mathbf{1}_3 & 0 \\ {}^{Q}p_{\times}^{P} & \mathbf{1}_3 \end{pmatrix} = \begin{pmatrix} \mathbf{1}_3 & 0 \\ {}^{P}p_{\times}^{Q} & \mathbf{1}_3 \end{pmatrix} = ({}^{P}\boldsymbol{S}^{Q})^{-1} \tag{44} $$

$$ {}^{B}\boldsymbol{X}^{A} = \begin{pmatrix} {}^{B}R^{A} & 0 \\ {}^{Bo}p_{\times}^{Ao}\ {}^{B}R^{A} & {}^{B}R^{A} \end{pmatrix} = \begin{pmatrix} {}^{B}R^{A} & 0 \\ -{}^{B}R^{A}\ {}^{Ao}p_{\times}^{Bo} & {}^{B}R^{A} \end{pmatrix} = ({}^{A}\boldsymbol{X}^{B})^{-1} \tag{45} $$

Se definirá el operador “dual” (*) como la traspuesta de la inversa de una matriz, por lo que:

$$ {}^{A}\boldsymbol{R}^{*B} \triangleq ({}^{A}\boldsymbol{R}^{B})^{-T} = ({}^{B}\boldsymbol{R}^{A})^{T} = {}^{A}\boldsymbol{R}^{B} \tag{46} $$

$$ {}^{P}\boldsymbol{S}^{*Q} \triangleq ({}^{P}\boldsymbol{S}^{Q})^{-\mathrm{T}} = ({}^{Q}\boldsymbol{S}^{P})^{T} = \begin{pmatrix} \mathbf{1}_3 & {}^{P}p_{\times}^{Q} \\ 0 & \mathbf{1}_3 \end{pmatrix} \tag{47} $$

$$ {}^{A}\boldsymbol{X}^{*B} \triangleq ({}^{A}\boldsymbol{X}^{B})^{-\mathrm{T}} = ({}^{B}\boldsymbol{X}^{A})^{\mathrm{T}} = \begin{pmatrix} {}^{A}R^{B} & {}^{Ao}p_{\times}^{Bo}\ {}^{A}R^{B} \\ 0 & {}^{A}R^{B} \end{pmatrix} \tag{48} $$

Las matrices $\boldsymbol{R}$, $\boldsymbol{S}$ y $\boldsymbol{X}$ se usarán para magnitudes espaciales derivadas de la posición, como velocidad y aceleración, mientras que las matrices $\boldsymbol{R}^{*}$, $\boldsymbol{S}^{*}$ y $\boldsymbol{X}^{*}$ se usarán magnitudes espaciales derivadas de la fuerza, como fuerzas e impulsos.

Por ejemplo, para rotar, desplazar o transformar una matriz espacial de inercia:

$$ {}^{A}\boldsymbol{R}^{*B} M_B\, {}^{B}\boldsymbol{R}^{A} = ({}^{B}\boldsymbol{R}^{A})^{T} M_B\, {}^{B}\boldsymbol{R}^{A} = {}^{A}\boldsymbol{R}^{B} M_B\, {}^{B}\boldsymbol{R}^{A} \tag{49} $$

$$ {}^{A}\boldsymbol{S}^{*B} M_B\, {}^{B}\boldsymbol{S}^{A} = ({}^{A}\boldsymbol{S}^{B})^{T} M_B\, {}^{B}\boldsymbol{S}^{A} \tag{50} $$

$$ {}^{A}\boldsymbol{X}^{*B} M_B\, {}^{B}\boldsymbol{X}^{A} = ({}^{B}\boldsymbol{X}^{A})^{\mathrm{T}} M_B\, {}^{B}\boldsymbol{X}^{A} \tag{51} $$

3.2.14. Construcción de un sistema multicuerpo

Simbody realiza cálculos con un modelo definido por el usuario, pero no puede tomar las decisiones de modelado por él. Las principales decisiones que tomar serán:

- Cómo descomponer el sistema físico en un conjunto particular de cuerpos rígidos.
- Qué movilizadores utilizar para interconectar los cuerpos unos con los otros, estableciendo una estructura en forma de árbol.
- Qué restricciones, si fueran necesarias, utilizar para restringir la movilidad de los cuerpos.

Estas decisiones serán tomadas durante la primera etapa de construcción de un SMC: la definición de la topología del sistema. La topología en Simbody engloba:

- El conjunto de cuerpos (y sus sistemas de referencia), que contará siempre con un cuerpo *Ground*.
- La estructura másica de cada cuerpo, comentada en el apartado **3.2.7**, que condicionará la manera de definir las propiedades másicas del cuerpo.
- El “padre” de cada cuerpo (excepto del *Ground*), respecto al cual se define su movilidad.
- Un conjunto de restricciones topológicas, es decir, aquellas restricciones que están siempre presentes y activas, como las encargadas de formar un mecanismo cerrado.

El conjunto de cuerpos interconectados entre sí formará una estructura en forma de árbol, cuya raíz será el *Ground*, que será el único cuerpo existente al principio. Todos los cuerpos estarán contenidos en el Subsistema *SimbodyMatterSubsystem*, y para añadir un cuerpo B a dicho Subsistema, será necesario especificar:

- El cuerpo padre P, con sistema de referencia P, que ya debe encontrarse en el sistema.
- Un sistema de referencia B, del cuerpo que se desea agregar.
- Las propiedades másicas del cuerpo B, con centro de masas B_O e inercia unitaria $\mathcal{G}_B$, ambas magnitudes expresadas en B.
- El sistema de referencia móvil M del movilizador, ligado a B. Debe expresarse como una transformación ${}^{B}X^{M}$ de M medida desde B.
- El sistema de referencia fijo F del movilizador, ligado a P y conectado a M por el movilizador. Debe expresarse como una transformación ${}^{P}X^{F}$ de F medida desde P.
- El tipo de movilizador utilizado para conectar B con P, y si se debe invertir la manera en la que se interpreten las coordenadas generalizadas.

Tras proporcionar toda la información anterior a la función apropiada de Simbody, el cuerpo B queda añadido al sistema, y se le asigna un número entero para hacer referencia al cuerpo posteriormente (al cuerpo *Ground* se le asigna el valor cero).

La siguiente figura ilustra la conexión de dos cuerpos con un movilizador y todos los sistemas de referencia mencionados:

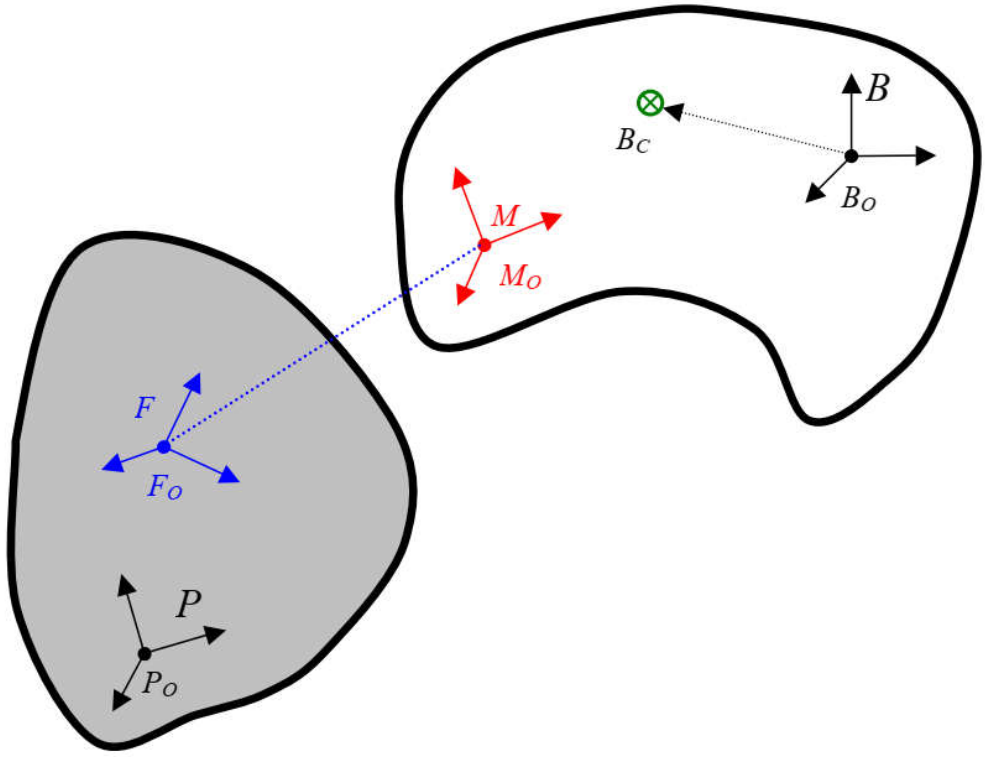

Figura 12. Adición de un cuerpo en Simbody. (Fuente: Manual de teoría de Simbody *[21]*)

Un cuerpo puede tener varios sistemas de referencia fijos de movilizadores, a los que se conecten otros cuerpos, pero solo puede tener un sistema de referencia móvil, que lo conecte a un cuerpo más cercano al cuerpo *Ground*. El término "fijo" es arbitrario y no debe interpretarse físicamente, ya que los únicos sistemas de referencia de un movilizador realmente fijos son aquellos en los que el cuerpo padre es el propio *Ground*.

Para la mayoría de movilizadores, la configuración en la que los sistemas de referencia M y F están superpuestos corresponde con coordenadas generalizadas de valor cero. Para todos ellos, el valor de las coordenadas generalizadas se expresará como una transformación ${}^{F}X^{M}$ que indica la localización y orientación de M, expresada en F. El tipo de movilizador definirá el significado de cada coordenada generalizada y los tipos de transformación posibles.

Mientras que los movilizadores son locales y aportan movilidad a un solo cuerpo, las restricciones son globales y reducen la movilidad de todo el sistema. Por ejemplo, una simple restricción de distancia constante entre un punto fijo de un cuerpo respecto a un punto fijo de otro, alejados entre sí en el árbol de cuerpos, expresará una relación compleja entre muchos cuerpos que debe cumplirse siempre. Por ese motivo, resulta más eficiente definir menos movilidad desde un principio que otorgar mayor movilidad y tener que restringirla después. Sin embargo, esto no siempre es posible, y por eso se unan las restricciones.

A diferencia de los movilizadores, que son independientes entre sí, las restricciones son interdependientes, lo que puede generar que, en ciertas situaciones, algunas se vuelvan ineficaces, redundantes o inconsistentes. Añadir la misma restricción dos veces sería redundante: la segunda restricción no cambia nada, ya que las coordenadas generalizadas que satisfacen la primera también satisfacen la segunda (por ser la misma). Un ejemplo de restricción ineficaz sería fijar la distancia de un punto de una rueda con su centro, para que sea igual a su radio. Dicha restricción se cumple siempre, y el sistema tendría la misma movilidad con o sin ella. Fijar la distancia para que sea distinta al radio de la rueda resultaría en una restricción inconsistente, que no puede ser cumplida nunca.

Simbody cuenta con varias clases de restricciones incorporadas. Entre ellas, se van a comentar las restricciones de distancia (*Rod*), de puntos coincidentes o bola (*Ball*) y de sistemas de coordenadas coincidentes o soldadura (*Weld*). Una restricción de distancia genera una ecuación de restricción para mantener dos puntos entre cuerpos a una distancia constante no nula. Cada restricción de distancia no redundante sustrae un GDL de la movilidad del sistema. Una restricción de puntos coincidentes genera tres ecuaciones de restricción para mantener dos puntos entre cuerpos en la misma posición, como si formasen una unión de bola. Cada restricción de puntos coincidentes no redundante sustrae 3 GDLs de la movilidad del sistema. Una restricción de soldadura mantiene dos sistemas de referencia entre cuerpos coincidentes, restringiendo traslación y orientación. Se añaden seis ecuaciones de restricción y, si no son redundantes, se sustraen seis GDLs de la movilidad del sistema.

Para añadir una restricción a un SMC en Simbody hay que especificar la siguiente información:

- Dos cuerpos distintos, de los cuales uno (pero no los dos) puede ser el *Ground*. Ambos cuerpos deben ser ya parte del sistema y deben identificarse por el número que les fue asignado cuando fueron añadidos.
- Tanto para restricciones de distancia o puntos coincidentes, un punto fijo P_A perteneciente al cuerpo A y un punto fijo P_B perteneciente al cuerpo B. Cada punto es medido y expresado en el sistema de referencia del cuerpo al que pertenecen, de manera que las coordenadas de los vectores posición serán constantes durante la simulación.
- Para la restricción de distancia constante, además, un escalar que represente la distancia $\mathrm{d} = |\mathrm{P_B} - \mathrm{P_A}|$ que no debe variar durante toda la simulación. Esta distancia debe ser significativamente mayor que cero (en dicho caso, se usa la restricción de puntos coincidentes).
- Para la restricción de soldadura, un sistema de referencia F_A fijo en A, y otro sistema de referencia F_B fijo en B. Ambos están expresados en los sistemas de referencia del cuerpo al que perteneces: F_A es definido por ${}^{A}X^{F_A}$ y F_B es definido por ${}^{B}X^{F_B}$.

Las restricciones no redundantes no serán satisfechas para valores arbitrarios de las coordenadas y velocidades generalizadas. Encontrar el conjunto de valores que las cumplan será, por lo general, un problema no lineal complejo que debe resolverse antes de comenzar la simulación (proceso que se llama análisis de ensamblado para $\mathbf{q}$ y análisis de velocidad para $\mathbf{u}$). Dado un conjunto de valores de $\mathbf{q}$ y $\mathbf{u}$, Simbody puede calcular eficientemente la diferencia en los términos de las ecuaciones de restricción, y ofrece métodos para reducir dichas diferencias por debajo de una tolerancia deseada, hasta que las restricciones se consideren satisfechas.

Simbody, además, permite añadir fuerzas a un SMC. Como se comentó en el apartado **3.2.10**, las fuerzas, que incluyen fuerzas lineales o rotacionales, pueden ser aplicadas a cuerpos o directamente a la movilidad del sistema, concretamente a las velocidades generalizadas $\mathbf{u}$. Aquellas aplicadas a cuerpos reciben el nombre de fuerzas espaciales o fuerzas de cuerpo, mientras que aquellas aplicadas a movilidades son llamadas fuerzas generalizadas o de movilidad. Para un conjunto de fuerzas espaciales, hay siempre un conjunto único de fuerzas generalizadas equivalentes, que producirán las mismas aceleraciones. Simbody puede calcular este conjunto equivalente, y deberá hacerlo, ya que las ecuaciones de movimiento están escritas en términos de movilidades.

Simbody proporciona multitud de subsistemas para el cálculo de fuerzas básicas como la gravedad, fuerzas elásticas o fuerzas atómicas; un conjunto que, aunque no es exhaustivo, puede ser fácilmente ampliado. Sin embargo, el cálculo de las fuerzas aplicadas no se limita a los tipos de fuerza proporcionados por Simbody, y es un problema específico de modelado. El trabajo de Simbody es proporcionar la información necesaria al modelador para que calcule las fuerzas, y actuar conforme a las leyes de Newton.

3.2.15. Teoría sobre movilizadores

Como ya ha sido comentado en el apartado **3.2.8**, un movilizador define la movilidad permitida entre un cuerpo B respecto a otro cuerpo P, más cercano al *Ground*, denominado padre. Los movilizadores proporcionan *n* movilidades o GDLs al cuerpo B con respecto al cuerpo P, donde $0 \leq n \leq 6$.

Todos los cuerpos tienen un único padre, por lo que existe correspondencia uno a uno entre el número de movilizadores y cuerpos. La combinación de un cuerpo y su movilizador correspondiente recibe el nombre de *MobilizedBody* (abreviado *Mobod*). Su movilidad permitida se describirá mediante *n* velocidades generalizadas u y $n_q \geq n$ coordenadas generalizadas q. Además, están las derivadas de las velocidades generalizadas con respecto del tiempo, denominadas aceleraciones generalizadas $\dot{\mathrm{u}}$.

El significado de estas magnitudes se define a partir de las **ecuaciones 52-56**, que expresan el movimiento relativo entre un cuerpo B con respecto de su padre P, en función de q y u. Dicho movimiento se define empleando los sistemas de referencia móvil M y fijo F (mencionados en el apartado **3.2.14**), ligados a sendos cuerpos B y P mediante sus respectivas transformaciones ${}^{B}X^{M}$ y ${}^{P}X^{F}$. Un esquema puede verse en la **Figura 13**.

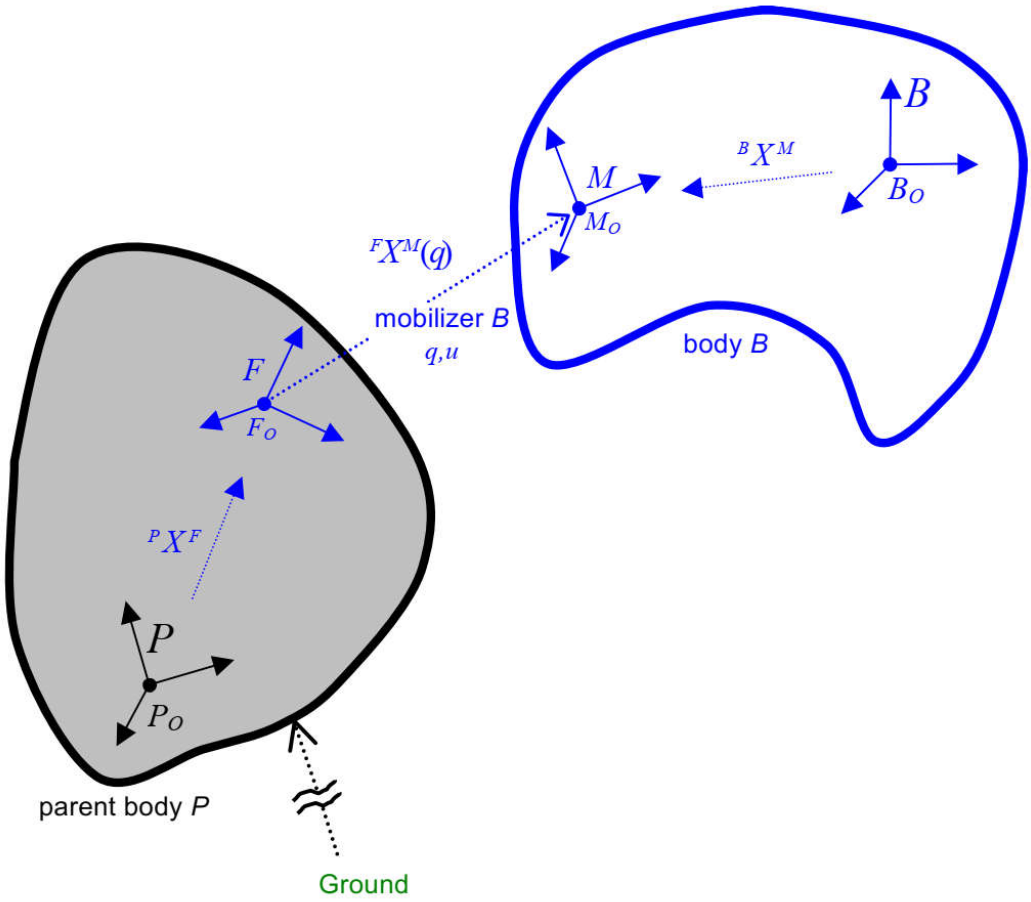

Figura 13. Sistemas de referencia utilizados para describir la movilidad de un *MobilizedBody* B. Todos los elementos en azul están asociados a un objeto *MobilizedBody*. (Fuente: Manual de teoría de Simbody *[21]*)

$$^{F}X^{M}(q) \triangleq \left(^{F}R^{M}(q)\middle|^{F}p^{M}(q)\right) \tag{52}$$

$$^{F}V^{M}(q,u) \triangleq \begin{pmatrix} ^{F}V_{\omega}^{M} \\ ^{F}V_{v}^{M} \end{pmatrix} = \begin{pmatrix} ^{F}\omega^{M} \\ ^{F}v^{M} \end{pmatrix} = {}^{F}\mathbf{H}^{M}\left(^{F}X^{M}(q)\right)u \tag{53}$$

$$^{F}A^{M}(q,u,\dot{u}) \triangleq {}^{F}\dot{V}^{M}(q,u) = \begin{pmatrix} ^{F}\beta^{M} \\ ^{F}a^{M} \end{pmatrix} = {}^{F}\mathbf{H}^{M}\dot{u} + {}^{F}\dot{\mathbf{H}}^{M}u \tag{54}$$

$$\dot{q} = {}^{F}\mathbf{N}^{M}(q)u \tag{55}$$

$$n(q) = 0 \tag{56}$$

Donde las derivadas respecto del tiempo están expresadas en F. La **ecuación 56** declara restricciones adicionales que debe cumplir q en caso de que la **ecuación 55** no pudiera especificar q de forma inequívoca. Esto ocurre cuando hay más coordenadas generalizadas que velocidades generalizadas, normalmente cuando q es un cuaternión. En ese caso, $n(q)$ es la condición de normalización del cuaternión.

La derivada de X respecto del tiempo está relacionada con V:

$$^{F}\dot{X}^{M} \triangleq \left(^{F}\dot{R}^{M}\middle|^{F}\dot{p}^{M}\right) = \left(^{F}\omega_{\times}^{M}\ {}^{F}R^{M}\middle|^{F}v^{M}\right) = \left(\left(^{F}V_{\omega}^{M}\right)_{\times}\ {}^{F}R^{M}\middle|\ {}^{F}V_{v}^{M}\right) \tag{57}$$

Lo que significa que, entre X, $\mathbf{H}$ y $\mathbf{N}$ existen relaciones que deben cumplirse:

$$ {}^{F}\dot{R}^{M} = {}^{F}\omega_{\times}^{M\,F}R^{M} = \left({}^{F}\mathbf{H}_{\omega}^{M}\right)_{\times}{}^{F}R^{M} = \frac{\partial^{F}R^{M}}{\partial q}\dot{q} = \frac{\partial^{F}R^{M}}{\partial q}\mathbf{N}\boldsymbol{u} \tag{58} $$

$$ {}^{F}\dot{p}^{M} = {}^{F}v^{M} = {}^{F}\mathbf{H}_{v}^{M}u = \frac{\partial^{F}p^{M}(q)}{\partial q}\dot{q} = \frac{\partial^{F}p^{M}(q)}{\partial q}\mathbf{N}\boldsymbol{u} \tag{59} $$

Donde $\mathbf{H}_{\omega}$ y $\mathbf{H}_{v}$ son las submatrices 3×n superior e inferior, respectivamente, de $\mathbf{H}$. De las **ecuaciones 58-59** se pueden obtener las siguientes conclusiones:

- La velocidad espacial V se encuentra relacionada con u mediante $\mathbf{H}$.
- V es la derivada con respecto del tiempo de q, a través de $X(q)$.
- $\dot{q}$ y u están relacionadas mediante $\mathbf{N}$.

Se deduce también, de la **ecuación 53**, que $\mathbf{H}$ depende solamente de la transformación X, y no de la definición de cada coordenada generalizada. Por este motivo, X y $\mathbf{H}$ no pueden ser arbitrarias.

Dado que el árbol de cuerpos en Simbody debe estar ordenado de padres a hijos del *Ground* hacia el exterior, en ocasiones el rol de padre e hijo puede estar intercambiado respecto al de un mecanismo real. A veces, para conservar el significado de las coordenadas generalizadas, sería conveniente expresar M (ligado al cuerpo B) respecto de F (ligado al cuerpo P), a pesar de que estos sistemas de referencia son definidos en el sentido opuesto al habitual: se debe emplear un movilizador inverso.

Para ello, en función de los términos ${}^{M}X^{F}$, ${}^{M}\mathbf{H}^{F}$, ${}^{M}\dot{\mathbf{H}}^{F}$ y $\mathbf{N}$ de un movilizador inverso, expresados en un sistema de referencia M fijo al cuerpo B (con derivadas respecto del tiempo expresadas en M), se debe expresar, con coordenadas y velocidades generalizadas idénticas, los términos ${}^{F}X^{M}$, ${}^{F}\mathbf{H}^{M}$, ${}^{F}\dot{\mathbf{H}}^{M}$ y $\mathbf{N}$ en el sistema de referencia F fijo al cuerpo P (con derivadas respecto del tiempo expresadas en F) para que Simbody pueda realizar los cálculos oportunos.

$\mathbf{N}$ es el termino más sencillo, ya que, para que q y u mantengan su significado original, $\mathbf{N}$ debe mantenerse igual. Por otro lado, para ${}^{F}X^{M}$, y también de forma sencilla:

$$ {}^{F}X^{M} = ({}^{M}X^{F})^{-1} = ({}^{F}R^{M}|-{}^{F}R^{M}\,{}^{F}p^{M}) \tag{60} $$

Para ${}^{M}\mathbf{H}^{F}$ y ${}^{M}\dot{\mathbf{H}}^{F}$ será algo más complejo. A partir de la **ecuación 53**, si se cambian los superíndices:

$$ {}^{M}V^{F} = \begin{pmatrix} {}^{M}\omega^{F} \\ {}^{M}v^{F} \end{pmatrix} = \begin{pmatrix} {}^{M}\mathbf{H}_{\omega}^{F} \\ {}^{M}\mathbf{H}_{v}^{F} \end{pmatrix} u \tag{61} $$

Se sabe que ${}^{F}\omega^{M} = -{}^{M}\omega^{F}$, por lo que ${}^{F}\mathbf{H}_{\omega}^{M} = {}^{M}\mathbf{H}_{\omega}^{F}$. Esta relación no puede aplicarse a la velocidad lineal. No obstante, a partir de la **ecuación 59**, se deduce la siguiente ecuación.

$$ {}^{F}v^{M} \triangleq {}^{F}\dot{p}^{M} \triangleq \frac{{}^{F}d}{dq}\,{}^{F}p^{M} \tag{62} $$

Y dado que ${}^{F}p^{M} = -\,{}^{M}p^{F}$, sustituyendo en la ecuación anterior, se obtiene la **ecuación 63**.

$$ {}^{F}v^{M} = -\frac{{}^{F}d}{dq}\,{}^{M}p^{F} = -({}^{M}\dot{p}^{F} + {}^{F}\omega^{M} \times {}^{M}p^{F}) = -({}^{M}v^{F} + {}^{M}p^{F} \times {}^{M}\omega^{F}) \tag{63} $$

Por último, como $v = \mathbf{H}_V u$ y $\omega = \mathbf{H}_\omega u$:

$$ {}^{F}\mathbf{H}_{v}^{M} = -\left({}^{M}\mathbf{H}_{v}^{F} + {}^{M}p_{\times}^{F}\,{}^{M}\mathbf{H}_{\omega}^{F}\right) \tag{64} $$

Hay que tener en cuenta que todas estas magnitudes están expresadas en el sistema de referencia M, por lo que hay que realizar un cambio de base:

$$ {}^{F}\mathbf{H}^{M} = -{}^{F}R^{M}\begin{pmatrix} {}^{M}\mathbf{H}_{\omega}^{F} \\ {}^{M}\mathbf{H}_{v}^{F} + {}^{M}p_{\times}^{F}\,{}^{M}\mathbf{H}_{\omega}^{F} \end{pmatrix} \tag{65} $$

Para obtener ${}^{F}\dot{\mathbf{H}}^{M}$, se deriva la ecuación anterior en función del tiempo:

$$ {}^{F}\dot{\mathbf{H}}^{M} = -{}^{F}\omega_{\times}^{M}\,{}^{F}R^{M}\begin{pmatrix} {}^{M}\mathbf{H}_{\omega}^{F} \\ {}^{M}\mathbf{H}_{v}^{F} + {}^{M}p_{\times}^{F}\,{}^{M}\mathbf{H}_{\omega}^{F} \end{pmatrix} - {}^{F}R^{M}\begin{pmatrix} {}^{M}\dot{\mathbf{H}}_{\omega}^{F} \\ {}^{M}\dot{\mathbf{H}}_{v}^{F} + {}^{M}p_{\times}^{F}\,{}^{M}\dot{\mathbf{H}}_{\omega}^{F} + {}^{M}\dot{p}_{\times}^{F}\,{}^{M}\mathbf{H}_{\omega}^{F} \end{pmatrix} \tag{66} $$

Y sustituyendo ${}^{F}\dot{p}^{M}$ de la **ecuación 62** y ${}^{F}\mathbf{H}^{M}$ de la **ecuación 65**:

$$ {}^{F}\dot{\mathbf{H}}^{M} = -{}^{F}\omega_{\times}^{M}\,{}^{F}\mathbf{H}^{M} - {}^{F}R^{M}\begin{pmatrix} {}^{M}\mathbf{H}_{\omega}^{F} \\ {}^{M}\mathbf{H}_{v}^{F} + {}^{M}p_{\times}^{F}\,{}^{M}\mathbf{H}_{\omega}^{F} \end{pmatrix} \tag{67} $$

Para evitar cálculos duplicados, se reordenarán cada una de las filas por separado en la **ecuación 65**:

$$ {}^{F}\mathbf{H}_{\omega}^{M} = -{}^{F}R^{M}\,{}^{M}\mathbf{H}_{\omega}^{F} \tag{68} $$

$$ {}^{F}\mathbf{H}_{v}^{M} = -{}^{F}R^{M}\,{}^{M}\mathbf{H}_{v}^{F} - {}^{F}R^{M}\,{}^{M}p_{\times}^{F}\,{}^{M}\mathbf{H}_{\omega}^{F} = -\left({}^{F}R^{M}\,{}^{M}\mathbf{H}_{v}^{F} - {}^{F}p_{\times}^{M}\,{}^{F}\mathbf{H}_{\omega}^{M}\right) \tag{69} $$

De donde ${}^{M}\mathbf{H}_{\omega}^{F} = -{}^{M}R^{F}\,{}^{M}\mathbf{H}_{\omega}^{F}$, por las **ecuaciones 16-17**, y ${}^{F}R^{M}\,{}^{M}p_{\times}^{F}{}^{M}R^{F} = ({}^{F}R^{M}\,{}^{M}p^{F})_{\times}$ usando la propiedad vista en la **ecuación 28**. Ahora, si se deriva respecto del tiempo:

$$ {}^{F}\dot{\mathbf{H}}_{\omega}^{M} = -{}^{F}R^{M}\,{}^{M}\dot{\mathbf{H}}_{\omega}^{F} + {}^{F}\omega_{\times}^{M}\,{}^{F}\mathbf{H}_{\omega}^{M} \tag{70} $$

$$ {}^{F}\dot{\mathbf{H}}_{v}^{M} = -\left({}^{F}R^{M}\,{}^{M}\dot{\mathbf{H}}_{v}^{F} - {}^{F}\omega_{\times}^{M}\,{}^{F}\mathbf{H}_{v}^{M} + {}^{F}p_{\times}^{M}\,{}^{F}\dot{\mathbf{H}}_{\omega}^{M} + \left({}^{F}v_{\times}^{M} - {}^{F}\omega_{\times}^{M}\,{}^{F}p_{\times}^{M}\right){}^{F}\mathbf{H}_{\omega}^{M}\right) \tag{71} $$

Finalmente, factorizando:

$$ {}^{F}\dot{\mathbf{H}}^{M} = -{}^{F}R^{M}\,{}^{M}\dot{\mathbf{H}}^{F} + {}^{F}\omega_{\times}^{M}\,{}^{F}\mathbf{H}^{M} - \begin{pmatrix} 0 \\ {}^{F}p_{\times}^{M}\,{}^{F}\dot{\mathbf{H}}_{\omega}^{M} + \left({}^{F}v_{\times}^{M} - {}^{F}\omega_{\times}^{M}\,{}^{F}p_{\times}^{M}\right){}^{F}\mathbf{H}_{\omega}^{M} \end{pmatrix} \tag{72} $$

Simbody utiliza las **ecuaciones 68, 69, 70 y 72**, en ese mismo orden, para realizar estos cálculos.

Todas las clases derivadas de *MobilizedBody* aceptan en sus constructores el parámetro enumerado *Direction*, que define si se debe invertir el movilizador. Sus posibles valores son:

- *Forward* = 0, movilizador directo, q y u se interpretan de F a M.
- *Reverse* = 1, movilizador inverso, q y u se interpretan de M a F.

Este parámetro es topológico y, por tanto, no puede ser modificado dinámicamente.

3.2.16. Teoría sobre restricciones

Una restricción C es modelada a través de un conjunto de m^C ecuaciones escalares que restringen los valores que pueden tomar las coordenadas y velocidades generalizadas, forzando relaciones algebraicas entre ellas o sus derivadas respecto del tiempo. Las restricciones normalmente se plantean para que afecten "directamente" solo a un número reducido n_b de cuerpos y n_m de movilizadores (entre uno y tres), a los que se llama cuerpos restringidos y movilizadores restringidos. Por motivos de eficiencia, Simbody debe conocer el conjunto completo $\{B_k^C, M_i^C\}$ de n_b^C cuerpos restringidos y n_m^C movilizadores restringidos por cada restricción C. Este conjunto es considerado información topológica y no podrá modificarse una vez se haya definido la restricción.

El conjunto de movilidades que puede aparecer en las ecuaciones de restricción engloba aquellas movilidades u_m^C asociadas a los movilizadores restringidos (afectadas directamente) más todas aquellas movilidades u_b^C asociadas al movimiento relativo entre los cuerpos restringidos (afectadas indirectamente). Mientras que el conjunto u_m^C suele ser bastante pequeño, el conjunto u_b^C puede llegar a ser notablemente más grande, afectando a todos los cuerpos anteriores en el árbol hasta el propio *Ground*. Para evitar incluir un número de movilidades innecesariamente grande, Simbody busca, para cada restricción C, atrás en el árbol de cuerpos hasta encontrar el ancestro común más cercano A^C, el cuerpo más alejado en el árbol desde el que se pueda llegar a todos los cuerpos restringidos, subiendo en el árbol. El propio *Ground* podría llegar a ser el ancestro común más cercano si no existieran otros cuerpos comunes.

Se denomina "rama k" al camino desde A^C hasta el k-ésimo cuerpo restringido, y estas diferentes ramas pueden solaparse entre sí, en función de la topología del árbol de cuerpos. El conjunto de velocidades generalizadas pertenecientes a la rama k se denota como $u_{b,k}^C$, donde $u_b^C = \cup_k u_{b,k}^C$; y el conjunto total de velocidades generalizadas afectadas por una restricción C será $u^C = \{u_m^C, u_b^C\}$. Estas serán las $n^C = |u^C|$ movilidades involucradas, mientras que las n_q^C coordenadas involucradas serán definidas como $q^C = \{q_m^C, q_b^C\}$ donde $q_b^C = \cup_k q_{b,k}^C$, $n_q^C = |q^C|$ y se cumple que $n_q^C \geq n^C$. La **Figura 14** muestra algunos de estos parámetros y los conceptos mencionados en este apartado.

El superíndice C se omitirá de ahora en adelante salvo cuando sea necesario por motivos de claridad. De forma resumida, una restricción generará m ecuaciones de restricción en n movilidades. La ecuación de restricción más básica es una relación entre aceleraciones (las derivadas respecto del tiempo $\dot{u}$ de las velocidades involucradas), llamada restricción de aceleración. Todas las ecuaciones de restricción en Simbody se reducen a restricciones de aceleración, y las m ecuaciones de restricción mencionadas formarán parte de las ecuaciones de movimiento de la dinámica. La i-ésima restricción de aceleración tiene la siguiente forma:

$$\mathrm{g}_i(t, q, u, \dot{u}) \triangleq \mathbf{g}_i \dot{u} - b_i(t.q.u) = 0 \qquad (73)$$

Donde b_i y g_i son funciones escalares y $\mathbf{g}_i = \mathbf{g}_i(q)$ es un vector fila de n elementos. Cada restricción debe disponer de un método eficiente de evaluar la función de error g_i. En aquellas que solo restringen movilizadores, se puede hacer directamente en función de movilidades u_m, pero cuando también se restringen cuerpos, las restricciones no aparecen de forma explícita como en la **ecuación 73**, sino en función de consecuencias físicas de $\dot{u}_b$, como aceleraciones de cuerpos. Se espera que la rutina escrita por el usuario pueda calcular el error de las ecuaciones en tiempo constante, y que Simbody posteriormente las consecuencias físicas de $\dot{u}_b$ tras un cálculo de O(*n*).

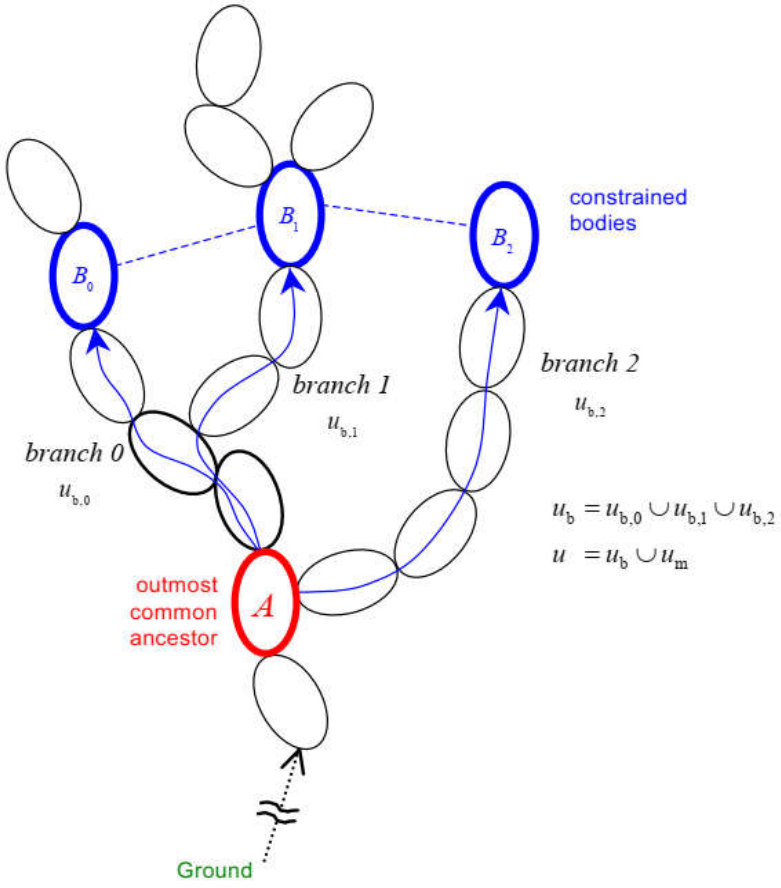

Figura 14. Topología de una restricción, mostrando tres cuerpos restringidos B_k y su ancestro común más cercano A_k, junto a las movilidades involucradas. (Fuente: Manual de teoría de Simbody *[21]*)

De manera similar, el significado de los multiplicadores de Lagrange λ es dado por:

$$f_i(q, \lambda_i) \triangleq \mathbf{g}_i^{\mathrm{T}} \lambda_i \tag{74}$$

Donde f_i es una función cuya solución es un vector columna de fuerzas generalizadas generadas por el multiplicador λ_i (escalar) asignado a la i-ésima ecuación de restricción. Cada restricción debe disponer también de un método para calcular eficientemente sus fuerzas dados los m multiplicadores λ. De nuevo, la **ecuación 74** no suele ser encontrada explícitamente en función de las fuerzas generalizadas sino de las fuerzas y momentos aplicados a los cuerpos, y se espera que las restricciones definidas por el usuario puedan ser calculadas en función de ellos en tiempo constante, para que Simbody convierta el resultado a fuerzas generalizadas en un cálculo de O(*n*).

Sin embargo, las ecuaciones de restricción pueden diferir según el nivel en el que son definidas: posición, velocidad o aceleración. Cuando una restricción es introducida en el nivel de posición, se llaman restricciones holonómicas y son normalmente no lineales. Las restricciones holonómicas se derivan una primera vez para producir una ecuación de restricción lineal de velocidades, y una segunda vez para producir una ecuación de restricción lineal de aceleraciones. Cuando la restricción es introducida en el nivel de velocidades, se llaman no holonómicas, son no lineales en cuanto a velocidades y se derivan una vez para generar una ecuación de restricción lineal de aceleraciones. Cuando una restricción es introducida en el nivel de aceleraciones, lo cual no es común, Simbody necesita que sea lineal en cuanto a aceleraciones generalizadas.

A continuación, se mostrarán las ecuaciones que definen los tres tipos de restricción.

Nivel de posición ($0 \leq j < m_p$):

$$\mathrm{p}_j(t,q) = 0 \tag{75}$$

$$\dot{\mathrm{p}}_j(t,q,u) \triangleq \mathbf{p}_j u - c_j(t,q) = 0 \tag{76}$$

$$\ddot{\mathrm{p}}_j(t,q,u,\dot{u}) \triangleq \mathbf{p}_j \dot{u} - b_{\mathrm{p},j}(t,q,u) = 0 \tag{77}$$

Nivel de velocidad ($0 \leq j < m_v$):

$$v_j(t,q,u) = 0 \tag{78}$$

$$\dot{v}_j(t,q,u,\dot{u}) \triangleq \mathbf{v}_j \dot{u} - b_{v,j}(t,q,u) = 0 \tag{79}$$

Nivel de aceleración ($0 \leq j < m_a$):

$$\dot{v}_j(t,q,u,\dot{u}) \triangleq \mathbf{v}_j \dot{u} - b_{v,j}(t,q,u) = 0 \tag{80}$$

Donde los vectores fila son:

$$\mathbf{p}_j(q) = \frac{\partial \ddot{\mathrm{p}}_j}{\partial \dot{u}} = \frac{\partial \dot{\mathrm{p}}_j}{\partial u} = \frac{\partial \mathrm{p}_j}{\partial q}\mathbf{N}^C \tag{81}$$

$$\mathbf{v}_j(q) = \frac{\partial \dot{\mathrm{v}}_j}{\partial \dot{u}} = \frac{\partial \mathrm{v}_j}{\partial u} \tag{82}$$

$$\mathbf{a}_j(q) = \frac{\partial \dot{\mathrm{a}}_j}{\partial \dot{u}} \tag{83}$$

Y los términos del resto al derivar son:

$$c_j(t,q) = -\frac{\partial \mathrm{p}_j}{\partial t} \tag{84}$$

$$b_{p,j}(t,q,u) = \dot{c}_j - \dot{\mathbf{p}}_j u \tag{85}$$

$$b_{v,j}(t,q,u) = -\left(\frac{\partial \mathrm{v}_j}{\partial t} + \frac{\partial \mathrm{v}_j}{\partial q}\dot{q}\right) \tag{86}$$

Donde $\mathbf{N}^C$ es una matriz $n_q^C \times n^C$ conformada por un subconjunto de filas y columnas de $\mathbf{N}$ (apartado **3.2.15**) tal que $\dot{q}^C = \mathbf{N}^C n^C$. Hay que destacar que las **ecuaciones 76 y 77** derivan de la **ecuación 75**, mientras que la **ecuación 78** deriva de la **ecuación 77**, y que, por tanto, no son independientes. Estas ecuaciones derivadas añaden un total de $2m_p + m_v$ ecuaciones de restricción extra a las m^C ecuaciones modeladas.

Todos los m_p vectores fila $\mathbf{p}_j$ se pueden combinar en una matriz $\mathbf{P}^C$. De manera análoga, los m_v vectores fila $\mathbf{v}_j$ se pueden combinar en la matriz $\mathbf{V}^C$, y los m_a vectores fila $\mathbf{a}_j$ se pueden combinar en la matriz $\mathbf{A}^C$. Todas estas luego pueden combinarse en una matriz de restricciones $\mathbf{G}^C$, donde las filas son, en realidad, cada uno de los vectores fila $\mathbf{p}_j$, $\mathbf{v}_j$ y $\mathbf{a}_j$:

$$\mathbf{G}^C = \begin{bmatrix} \mathbf{P}^C \\ \mathbf{V}^C \\ \mathbf{A}^C \end{bmatrix} \tag{87}$$

El cálculo de las fuerzas de restricción involucra productos matriz-vector de matrices de restricción y sus traspuestas, que Simbody necesita resolver de manera eficiente. Se espera un cálculo de ambos $\mathbf{Gv}$ y $\mathbf{Gw}$ de orden O(*n*+*m*), donde $\mathbf{G}$ es una matriz *m*×*n* y $\mathbf{v}$ y $\mathbf{w}$ son vectores columna de *n* elementos; cuando una multiplicación común es de O(*mn*). Simbody emplea los métodos vistos para definir las restricciones junto con operadores de O(*n*).

Con multiplicaciones matriz-vector de O(*n*+*m*), Las matrices $\mathbf{P}, \mathbf{V}$ y $\mathbf{A}$ pueden calcularse en tiempo por elemento constante (y no lineal). Realizando *m* llamadas a las rutinas disponibles, matrices de *m*×*n* se pueden calcular en O(*nm*+m^2)=O(*nm*), dado que $m \leq n$ y $nm+m^2 \leq 2nm$, lo que se considera dentro del intervalo óptimo teniendo en cuenta que también se deben construir las matrices.

Independientemente del nivel en que sea introducida, toda restricción añade una fila $\mathbf{g}$ tanto en la matriz de restricción $\mathbf{G}$ como en una de las matrices $\mathbf{P}, \mathbf{V}$ o $\mathbf{A}$. Estas pueden, entonces, ser obtenidas examinando las funciones $\mathbf{g_i}$ una vez las restricciones han sido expresadas en el nivel de aceleración: esto es, las **ecuaciones 77, 79 y 80**. Todas ellas, no son más que parte del conjunto mostrado en la **ecuación 73**, del que, derivando respecto de las aceleraciones generalizadas se obtienen los vectores filas necesarios (**ecuaciones 81-83**). Una alternativa es usar la función de fuerzas de restricción (**ecuación 74**) para obtener una columna equivalente a $\mathbf{g}_i^{\mathrm{T}}$ con un coste por columna de orden O(*n*). De esa manera, Simbody obtiene una fila de las matrices de restricción cada vez, y realizando m^C llamadas a la función de fuerzas de restricción de O(n^C), produciendo una matriz $\mathbf{G^C}$ explícita, con precisión de máquina, en un total de operaciones de orden O($m^C n^C$), lo que implica tiempo constante por elemento, al ser la matriz de $m^C \times n^C$. El conjunto de m^C escalares b_i mostrados en la **ecuación 73** forman el vector $\mathbf{b}$:

$$\mathbf{b^C} = \begin{bmatrix} b_0 \\ \vdots \\ b_{m^C-1} \end{bmatrix} \tag{88}$$

Este vector puede ser determinado explícitamente en un tiempo de O(*n*) fijando todas las aceleraciones generalizadas a cero.

Como ya se ha mencionado, no es común que las restricciones de aceleración estén escritas directamente en términos de aceleraciones generalizadas, sino en función de consecuencias físicas que atañen a los cuerpos restringidos. Estas pueden ser expresiones complicadas, pero se basan todas en una de las siguientes magnitudes:

- Aceleraciones lineales de puntos y aceleraciones angulares de vectores, fijos en el sistema de referencia del cuerpo restringido, relativas entre el cuerpo y su ancestro común o cualquier otro de los cuerpos restringidos.
- Aceleraciones de movilizador, en función de las aceleraciones generalizadas del movilizador restringido.

La clase base *Constraint* de Simbody dispone de herramientas para determinar las aceleraciones relativas entre cuerpos y su ancestro común más cercano, dado un conjunto $\dot{u}$ (manteniendo fijos q y u), de velocidades relativas sabiendo u (manteniendo q fijo) o de posiciones relativas sabiendo q. Las ecuaciones de error de restricciones personalizadas por del usuario son escritas utilizando dichas herramientas.

3.2.17. Estado y actualización

En el apartado **3.2.1** se introdujo el objeto Estado y se habló en general de la arquitectura de Simbody, mientras que el apartado **3.2.3** se centró concretamente en su proceso de cálculo por etapas. Como ya se explicó, durante un Estudio, el Sistema, a menudo, "actualiza" el Estado, entendiéndose esa "actualización" como tomar un nuevo conjunto de valores del Estado y usarlos para realizar los cálculos que dichos valores permitan.

Esos cálculos se pueden clasificar en tres categorías distintas: respuestas, operadores y solucionadores (*responses, operators* y *solvers*). Una respuesta es un valor numérico cuyo cálculo solo requiere valores almacenados en el Estado, como, por ejemplo, la distancia entre dos puntos, cada uno de distintos cuerpos. Un operador es calculado conociendo solo ciertas variables de estado, pero puede ser aplicado más tarde a otros datos para producir resultados numéricos. Un ejemplo sería un operador que, a partir de las posiciones y velocidades de cuerpos, pueda ser aplicado a un conjunto de fuerzas para determinar las aceleraciones que producen). Mientras que ni las respuestas ni los operadores producen cambios en el Estado, un solucionador lee y escribe datos en el Estado. Un solucionador sencillo podría, por ejemplo, leer datos y usarlos para establecer el valor de una de las variables de estado. Un esquema sobre el comportamiento de respuestas, operadores y solucionadores puede verse en la siguiente figura:

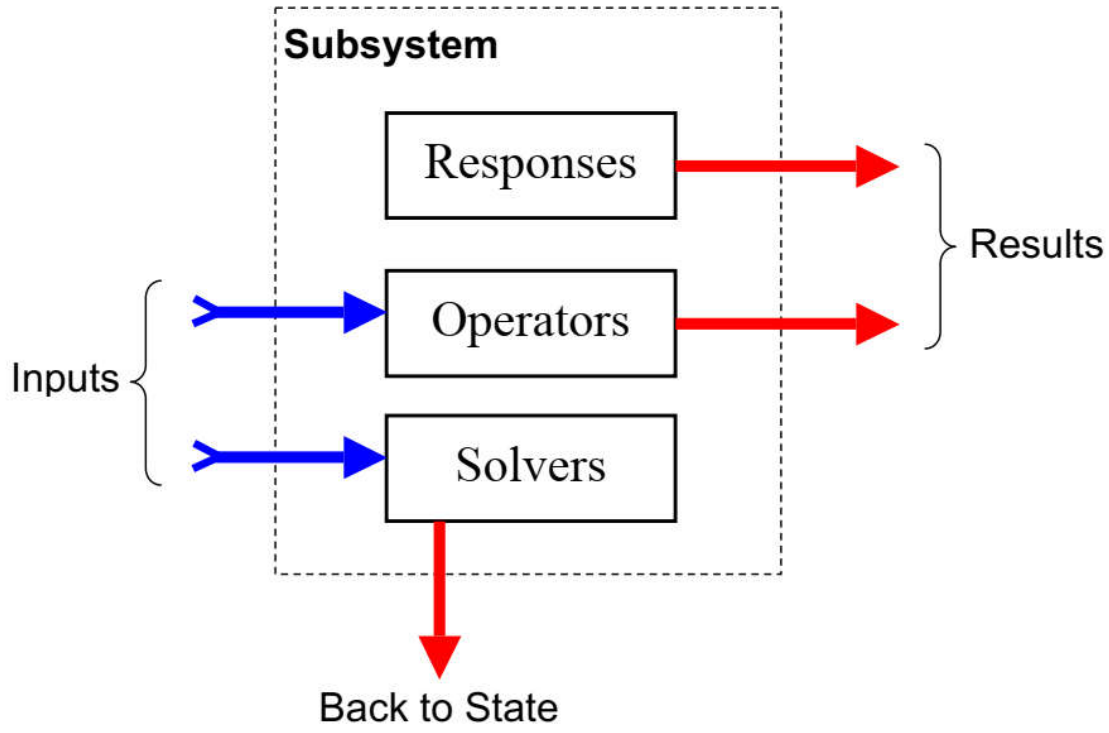

Figura 15. Respuestas, operadores y solucionadores. (Fuente: Manual de teoría de Simbody *[21]*)

Como ya se ha comentado, la actualización de un Estado puede ser un proceso costoso computacionalmente, y sus resultados numéricos pueden ser requeridos con cierta frecuencia en cálculos posteriores, por lo que se debe asegurar que la actualización se realice una única vez para cada conjunto de variables de Estado. La solución es almacenar estos resultados, y dado que estos están asociados a un Estado en particular y que un Sistema es un objeto de solo lectura, el lugar más apropiado es el propio objeto Estado, en la caché de actualización.

De esta manera, los resultados son siempre accesibles y se elimina la posibilidad de que los cálculos asociados a un Estado hayan sido realizados utilizando los valores de otro, produciendo errores difíciles de arreglar, pues los valores de Estados consecutivos son muy similares y estos "pequeños" errores pasarían inadvertidos.

El estado de un Sistema en Simbody, como ya se sabe, es representado por un conjunto S de variables de estado, almacenadas físicamente en el objeto Estado, donde son divididas en tiempo $t \in S$ y dos subconjuntos disjuntos $x, y \subseteq S$, tal que $S \equiv \{t\} \cup x \cup y$. Durante el *time stepping*, t es la variable independiente y x e y contienen variables de estado dependientes que difieren en el tipo de dato: y contiene las variables de estado de tipo real q, u y z; mientras que x contiene otros tipos de datos como booleanos, enteros y estructuras.

Las variables de estado están divididas según la etapa de computación en la que son utilizadas. Como se muestra en la **Figura 8**, aquellas etapas afectadas por variables de estado van desde la etapa número dos (Modelo) a la etapa número ocho (Aceleración). Las divisiones en las variables de estado se denotarán por S^{model}, $S^{instance}$, S^{time}, S^{pos}, S^{vel}, S^{force} y S^{acc}, de acuerdo con cada una de las etapas, y donde:

$$S^{time} \triangleq x^{time} \cup t \tag{89}$$

$$S^{pos} \triangleq x^{pos} \cup y^{pos} = x^{pos} \cup q \tag{90}$$

$$S^{vel} \triangleq x^{vel} \cup y^{vel} = x^{vel} \cup u \tag{91}$$

$$S^{force} \triangleq x^{force} \cup y^{force} = x^{force} \cup z \tag{92}$$

Durante las otras etapas solo se usan variables de x, por lo que:

$$S^{model} \triangleq x^{model} \tag{93}$$

$$S^{instance} \triangleq x^{instance} \tag{94}$$

$$S^{acc} \triangleq x^{acc} \tag{95}$$

La partición y consiste solo de las variables de estado asociadas a las etapas de Posición, Velocidad y Fuerza, por lo que dichas divisiones han sido renombradas (Como puede apreciarse en las **ecuaciones 90-92**) como q, u y z, cumpliéndose que $y = q \cup u \cup z$.

El objeto Estado recoge una colección de recursos necesarios durante la actualización, que son asignados a petición de los Subsistemas, y por ello, se organizan por Subsistemas. Otros recursos son incluidos en conjuntos globales que representan la totalidad del Sistema y permiten un manejo eficiente de su estado cuando la composición de los Subsistemas es irrelevante. Todo recurso tiene una etapa de asignación, que indica que el recurso debe asignarse en dicha etapa y desasignarse cuando esa etapa quede invalidada. Esto significa que el número de recursos en un Sistema puede variar según la etapa actual y la configuración de las variables de estado.

Cuando los recursos deben estar siempre presentes para un determinado Sistema, son asignados durante la etapa Topología. Un ejemplo son el conjunto de variables de estado que pueden afectar a las opciones de modelado. Cuando el Estado es actualizado a la etapa Modelo, se asignan nuevos recursos, relativos a los parámetros de modelado. A continuación, al actualizar a la etapa Instancia, se asignarán los recursos necesarios para el resto de los cálculos posteriores. A partir de este punto, no se asignarán nuevos recursos, pero pueden variar los valores de estado de los recursos asignados. La **Tabla 3** muestra los recursos soportados por el objeto Estado:

Recurso	Asignación	Descripción	Ejemplos
Recursos integrados	Topología	Todo Estado tiene recursos integrados. Hay que tener que cuenta que la etapa de un Sistema no puede ser mayor que la menor de las etapas de los Subsistemas	t, t_{prev} Etapa actual del Sistema Etapa actual de Subsistemas
Variables dinámicas (q, u, z)	Topología Modelo Instancia	Variables de estado de tipo real agrupadas en un array global y, dividido a su vez en los subvectores q, u y z. Una variable de estado adicional con el tamaño del grupo es fijada en la etapa Instancia	$\dot{y} = \{\dot{q}, \dot{u}, \dot{z}\}$, $\ddot{q}$ $y_{prev} = \{q_{prev}, u_{prev}, z_{prev}\}$ Tamaños
Variable de tipo estructura x	Topología Modelo	Variable privada perteneciente a un único Subsistema y capaz de contener un dato de cualquier tipo. Debe asignarse antes de la etapa que la invalide.	Valor previo x_{prev}
Variables de restricciones	Topología Modelo Instancia	Entradas en caché con arrays de *m* escalares que contienen los errores de las restricciones y los multiplicadores. Una variable adicional con el valor de *m* es fijada en la etapa Instancia.	$y_{err} = \{q_{err}, u_{err}\}$ $\dot{u}_{err}$, λ, *m*
Variables de activadores de evento	Topología Modelo Instancia	Entrada en caché con array de escalares que representan el valor de las funciones activadoras de evento. Son globales y su tamaño es fijado en la etapa Instancia	e, e_{prev} Número de activadores
Variables de evento	Topología Modelo Instancia	Conjunto de identificadores de eventos para un Subsistema e identificadores globales del Sistema correspondiente.	Número de eventos
Entradas en caché	Topología Modelo Instancia	Variable privada perteneciente a un único Subsistema y capaz de contener un dato de cualquier tipo. Debe asignarse antes de la etapa de la que depende	Identificador de Subsistema

Tabla 3. Recursos del Estado.

3.2.18. Ecuaciones de movimiento

Se ha explicado durante este trabajo que un SMC es representado por un conjunto de ecuaciones con las que una simulación sí puede trabajar. Una parte de esas ecuaciones son las ecuaciones de movimiento, que no son más que las derivadas respecto del tiempo de las variables de estado.

Estas ecuaciones de movimiento pueden escribirse en función de las n velocidades generalizadas $u = \bigcup_b u[b]$, y las n_q coordenadas generalizadas $q = \bigcup_b q[b]$, donde b es el conjunto de los cuerpos y $u[b]$ y $q[b]$ los conjuntos disjuntos de $n[b]$ velocidades y $n_q[b]$ coordenadas de cada uno de los cuerpos $\mathbf{B}[b]$. Normalmente existirán también un conjunto de ecuaciones diferenciales del modelo de fuerzas, asociadas a las n_z variables auxiliares z; y un conjunto de ecuaciones discretas asociadas a las variables discretas, aunque en este apartado se centrará en las ecuaciones continuas.

El número total de movilidades n de un SMC en Simbody será la suma de las movilidades sin restringir de cada uno de los cuerpos, es decir, $n = \sum_b n[b]$. El número neto de movilidades tras aplicar las restricciones será $n_{net} = n - m_{net}$, donde m_{net} es el número de ecuaciones de restricción de aceleración independientes generadas por las m restricciones, cumpliéndose que $m_{net} \leq m$.

Las velocidades generalizadas estaban relacionadas con magnitudes físicas del sistema, mientras que las coordenadas generalizadas se eligen para facilitar el cálculo numérico. Esto causa que el número de movilidades que añade un movilizador sea igual al número de velocidades generalizadas, y que estas últimas sean mutuamente independientes. Puede ocurrir que $n_q[b] \geq n[b]$ y, por tanto, que las coordenadas generalizadas puedan no ser independientes. La solución es introducir, por conveniencia, un parámetro $n_{quat}[b]$ definido tal que:

- $n_{quat}[b] = 1$, si $\mathbf{B}[b]$ usa un cuaternión.
- $n_{quat}[b] = 0$, en caso contrario.

Y el número total de cuaterniones en el sistema es $n_{quat} = \sum_b n_{quat}[b]$.

En los sistemas sin restricciones, donde las variables q, u y z son definidas dinámicamente mediante ecuaciones diferenciales, las ecuaciones de movimiento están formadas por las **ecuaciones 55 y 56** y las dos siguientes:

$$\mathbf{M}(q)\dot{u} = \mathbf{f}_{\text{app}}(t, q, u, z) - \mathbf{f}_{\text{bias}}(q, u) \tag{96}$$

$$\dot{z} = \dot{z}(t, q, u, z, \dot{u}) \tag{97}$$

Donde $\mathbf{M}$ es una matriz n×n simétrica que recoge todas las propiedades de inercia del sistema en su configuración actual, mientras que $\mathbf{f}_{\text{app}}$ es un vector 1×n de fuerzas y momentos aplicados, convertidos en n fuerzas generalizadas que actúan directamente sobre las movilidades. $\mathbf{f}_{\text{bias}}$ es otro vector 1×n de fuerzas giroscópicas y de Coriolis inducidas por la velocidad relativa entre cuerpos. El vector de fuerzas aplicadas $\mathbf{f}_{\text{app}}$ será dividido de la siguiente manera:

$$\mathbf{f}_{\text{app}} = \mathbf{f}_{\text{mob}} + J^{\text{T}} \bullet \mathbf{F}_{\text{body}} \tag{98}$$

Aquí $\mathbf{F}_{\text{body}}$ es un vector n_B×1 donde cada elemento es el vector de fuerzas espaciales (6×1) correspondiente a cada uno de los cuerpos; mientras que $\mathbf{f}_{\text{mob}}$ es un vector n×1 que contiene las fuerzas que son aplicadas directamente a las movilidades. $J^{\text{T}} \bullet$ es un operador de O(n) para convertir coordenadas cartesianas a coordenadas internas, conceptualmente una matriz n_B×n de vectores espaciales para convertir fuerzas espaciales en fuerzas de movilidad. En la **ecuación 55**, $\mathbf{N}$ es una matriz n_q×n diagonal por bloques e invertible de conversión entre $\dot{q}$ y u, aunque en la práctica se usa para convertir velocidades angulares en derivadas de cuaterniones o derivadas de ángulos de Euler. Son necesarias n_{quat} ecuaciones adicionales de normalización (**ecuación 56**) cuando el sistema de ecuaciones planteado en la **ecuación 55** sea rectangular (distinto número de ecuaciones que de incógnitas), debido a que $n \leq n_q$.

Por otro lado, están las ecuaciones de las trayectorias $q(t)$, $u(t)$ y $z(t)$ que intenta determinar un *time stepper*, que son:

$$q(t) = q_0(t) + \int_{\tau=t_0}^{t} \dot{q}(\tau)\, d\tau \tag{99}$$

$$u(t) = u_0(t) + \int_{\tau=t_0}^{t} \dot{u}(\tau)\, d\tau \tag{100}$$

$$z(t) = z_0(t) + \int_{\tau=t_0}^{t} \dot{z}(\tau) d\tau \tag{101}$$

Simbody resuelve las **ecuaciones 55-56 y 96-97** analíticamente, mientras que las **ecuaciones 99-101** solo pueden ser resueltas numéricamente.

La **ecuación 96** no es más que una versión de la segunda ley de Newton, pues solo se relacionan fuerzas con aceleraciones. Su solución formal sería la siguiente:

$$\dot{u} = \mathbf{M}^{-1}(\mathbf{f}_{\text{app}} - \mathbf{f}_{\text{bias}}) \tag{102}$$

Aunque en la práctica, no es resuelta así por Simbody. La estructura de $\mathbf{M}$ se puede aprovechar para calcular las aceleraciones en tiempo de O(*n*), mientras que realizar su inversa sería de O(n^3), y simplemente la formación de la matriz $\mathbf{M}$, con aproximadamente $n^2/2$ elementos únicos, sería una operación de O(n^2).

Cuando se introducen restricciones en un sistema, se introducen con ellas fuerzas de restricción desconocidas, que provocan otras incógnitas en el sistema. Estas incógnitas se conocen como multiplicadores de Lagrange, y son representadas con un vector λ de m elementos. Estos multiplicadores son convertidos a fuerzas de movilidad por medio de una matriz $\mathbf{G}$, modificando la **ecuación 96** de la siguiente manera:

$$\mathbf{M}\dot{u} + \mathbf{G}^{\mathrm{T}}\lambda = \mathbf{f}_{\text{app}} - \mathbf{f}_{\text{bias}} \tag{103}$$

$$\mathbf{G}\dot{u} = \mathbf{b} \tag{104}$$

Donde $\mathbf{G} = \mathbf{G}(q)$ tiene un tamaño *m*×*n* y $\mathbf{b} = \mathbf{b}(t, q, u)$ es un vector de *m* elementos. Las **ecuaciones 103 y 104** forman un sistema de *n*+*m* ecuaciones y *n*+*m* incógnitas y, por tanto, puede resolverse para encontrar un conjunto de aceleraciones que satisfaga las restricciones impuestas. Simbody utiliza los resultados obtenidos en la **ecuación 102**, que puede verse como un caso particular en el que se elimina λ y se resuelve para $\dot{u}$.

3.3. Filtro de partículas

Para esta sección se ha utilizado como referencias la tesis doctoral "*Contributions to Localization, Mapping and Navigation in Mobile Robotics*" [30], en especial los capítulos 3 y 4, que tratan los aspectos teóricos necesarios para un filtrado de partículas óptimo; y el artículo "*Multibody dynamic systems as Bayesian Networks: applications to robust state estimation of mechanisms*" [2], que aborda el problema concreto de estimación del estado de un mecanismo de cuatro barras, utilizando medidas ruidosas de sensores, desde el punto de vista de modelos gráficos.

3.3.1. Fundamentos de estimadores de estado

Sea un proceso estocástico Y, se quiere estimar en tiempo discreto su estado oculto y, es decir, se desea conocer el conjunto de estados $y_1, y_2, \cdots, y_t$ siendo $t \in \mathbb{N}$ el número de pasos tomados por el filtro. Es importante que este proceso cumpla la propiedad de Markov: la probabilidad de ocurrencia de un estado debe depender únicamente del estado inmediatamente anterior:

$$p(y_{t+1}|y_1, y_2, \cdots, y_t) = p(y_{t+1}|y_t) \tag{105}$$

O expresado de otra manera: $y_{t+1} \perp\!\!\!\perp y_1, y_2, \cdots, y_{t-1}$.

Cuando el proceso Y modele el comportamiento de un SMC, podrá ser descrito con un conjunto de coordenadas dependientes $\mathbf{q}$, que pueden ser determinadas a partir de un conjunto mínimo de coordenadas generalizadas $\mathbf{z}$. El primer conjunto englobará $\mathbf{q_t}$, $\dot{\mathbf{q}}_t$ y $\ddot{\mathbf{q}}_t$, que se corresponden con los vectores de posición, velocidad y aceleración de un mecanismo en un paso concreto t; mientras que el segundo conjunto englobará sus equivalentes generalizadas, $\mathbf{z_t}$, $\dot{\mathbf{z}}_t$ y $\ddot{\mathbf{z}}_t$. El sistema estará sometido además a un conjunto de fuerzas generalizadas agrupadas en un vector $\mathbf{Q_t}$, cuya longitud se corresponde con la del vector $\mathbf{q_t}$. Por último, el sistema dispondrá de un conjunto de sensores, cuyas lecturas estarán agrupadas en un vector $\mathbf{o_t}$.

Dado que en las ecuaciones de movimiento existe una fuerte relación entre $\mathbf{z_t}$ y $\dot{\mathbf{z}}_t$, y que las aceleraciones $\ddot{\mathbf{z}}_\mathbf{t}$ aparecen en función de las anteriores y de $\mathbf{Q_t}$, pero no de estados y fuerzas anteriores, se puede formar una variable auxiliar $\mathbf{x_t} = \{\mathbf{z_t}, \dot{\mathbf{z}}_t\}$, que sin perder información, reduzca el número de variables y deje a $\ddot{\mathbf{z}}$ como variable independiente en función de las fuerzas $\mathbf{Q_t}$ y el estado $\mathbf{x_t}$. Por otro lado, las lecturas $\mathbf{o_t}$ de los sensores dependen exclusivamente del estado actual $\mathbf{x_t}$ del sistema y de sus aceleraciones $\ddot{\mathbf{z}}$, ambas variables ocultas, mientras que las mismas lecturas y las fuerzas $\mathbf{Q_t}$ son variables visibles del sistema.

La **Figura 16** ilustra la evolución de un SMC a lo largo del tiempo, mostrando las variables comentadas arriba:

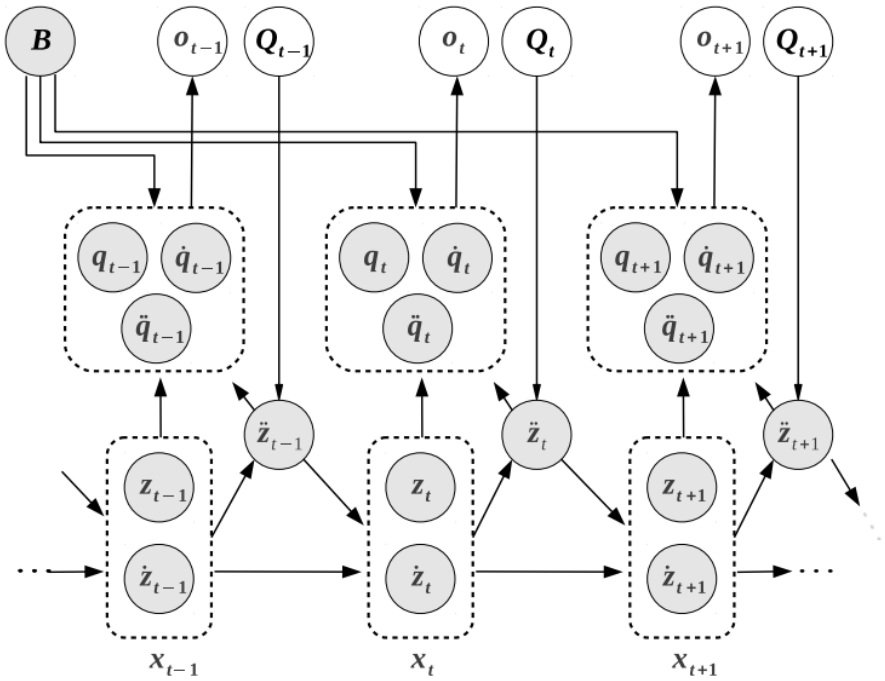

Figura 16. Grafo de la evolución de un SMC en tiempo discreto. Aquí *B* es una variable auxiliar que representa la configuración o modo de funcionamiento de un mecanismo. Los nodos sombreados representan las variables ocultas del sistema. (Fuente: *[2]*)

Las coordenadas dependientes $\mathbf{q}$ son funciones deterministas de las coordenadas generalizadas $\mathbf{z}$. Por este motivo, en un problema de estimación de estado, tiene sentido centrarse en inferir las coordenadas generalizadas y cuando $\mathbf{q}$ sea necesario, calcularlas rápidamente a partir de $\mathbf{z}$. Esta exclusión de variables en un modelo estadístico se conoce como marginalización, y a pesar de tener sus pros y contras, en este caso, reduce la dimensión del problema. El esquema visto en la **Figura 16** se puede reducir tras marginalizar las variables dependientes, como se puede ver en la siguiente figura:

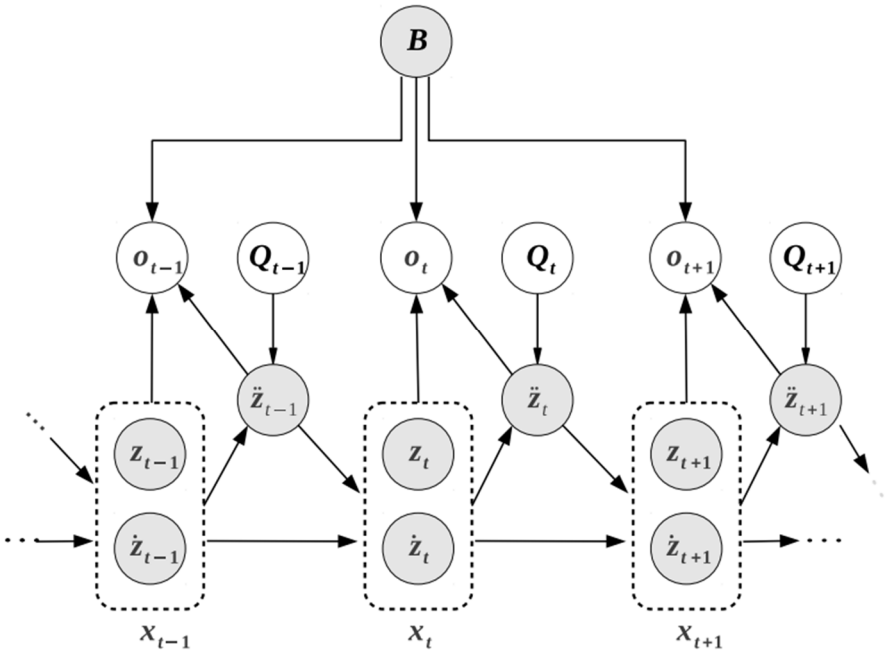

Figura 17. Grafo reducido tras marginalizar las coordenadas dependientes. (Fuente: *[2]*)

El objetivo de un estimador de estado será realizar inferencia estadística para estimar la función de distribución *a posteriori* de las variables ocultas ($\mathbf{x_t}$ y $\ddot{\mathbf{z}}_t$) de un sistema en un determinado paso de tiempo discreto t, dada la secuencia de lecturas de los sensores $\mathbf{o_{1:t}}$ y de fuerzas conocidas $\mathbf{Q_{1:t}}$. Esto es, al fin y al cabo, determinar la FDP:

$$p(\mathbf{x_t}, \ddot{\mathbf{z}}_t | \, \mathbf{Q_{1:t}}, \mathbf{o_{1:t}}) \tag{106}$$

Por la propia definición de probabilidad condicionada, se obtiene:

$$p(\mathbf{x_t}, \ddot{\mathbf{z}}_t | \, \mathbf{Q_{1:t}}, \mathbf{o_{1:t}}) = \underbrace{p(\ddot{\mathbf{z}}_t | \, \mathbf{x_t}, \mathbf{Q_{1:t}}, \mathbf{o_{1:t}})}_{T_1} \underbrace{p(\mathbf{x_t} | \, \mathbf{Q_{1:t}}, \mathbf{o_{1:t}})}_{T_2} \tag{107}$$

Y aplicando el teorema de Bayes al primer término, T_1:

$$T_1 \propto p(\mathbf{o_t} | \mathbf{x_t}, \ddot{\mathbf{z}}_t, \mathbf{Q_{1:t}}, \mathbf{o_{1:t-1}}) p(\ddot{\mathbf{z}}_t | \, \mathbf{x_t}, \mathbf{Q_{1:t}}, \mathbf{o_{1:t-1}}) \tag{108}$$

Donde se ha ignorado el factor de proporcionalidad (el término faltante del teorema de Bayes) ya que no afecta a la estimación. Por otro lado, la independencia condicional existente entre variables permite simplificar la expresión anterior:

$$T_1 \propto \underbrace{p(\mathbf{o_t} | \mathbf{x_t}, \ddot{\mathbf{z}}_t)}_{T_{1.1}} \underbrace{p(\ddot{\mathbf{z}}_t | \, \mathbf{x_t}, \mathbf{Q_t})}_{T_{1.2}} \tag{109}$$

Donde el primer término, $T_{1.1}$, es la verosimilitud de la observación de los sensores, más conocido por su denominación en inglés, *observation likelihood*; mientras que el segundo, $T_{1.2}$, es el modelo probabilístico de movimiento. El término $T_{1.1}$ es muy importante, pues es el único momento en el que las predicciones del estimador son contrastadas con los datos obtenidos de los sensores, para realizar correcciones mientras se monitorea el movimiento del mecanismo.

El término T_2 representa la distribución marginal la cual omite todos los estados previos $\mathbf{x_{1:t-1}}$ y aceleraciones $\ddot{\mathbf{z}}_{1:t}$. Sin embargo, tanto la propiedad de Markov como el modelo gráfico de la **Figura 17**, nos deja claro que un estado presente $\mathbf{x_t}$ debe depender del estado anterior $\mathbf{x_{t-1}}$ y de $\ddot{\mathbf{z}}_{t-1}$, por lo que deben aparecer en la expresión de $\mathbf{x_t}$. Para ello, se puede aplicar el teorema de la probabilidad total, que aprovechando las independencias condicionales entre variables para eliminar aquellos términos que no aportan información relevante:

$$T_2 = \int \int_{-\infty}^{\infty} p(\mathbf{x_t} | \mathbf{x_{t-1}}, \ddot{\mathbf{z}}_{t-1}) p(\mathbf{x_{t-1}}, \ddot{\mathbf{z}}_{t-1} | \, \mathbf{Q_{1:t-1}}, \mathbf{o_{1:t-1}}) d\mathbf{x_{t-1}}, d\ddot{\mathbf{z}}_{t-1} \tag{110}$$

El término T_2 toma como entrada la FDP *a posteriori* del paso anterior $t-1$ y junto con T_1 permite estimar una nueva FDP corregida para t. Aquí se encuentra la naturaleza recursiva del estimador empleado. El siguiente paso es implementar las ecuaciones anteriores para poder hacer un uso práctico de ellas (apartado **3.3.2**). Las dos maneras fundamentales de hacerlo es con distribuciones paramétricas o con distribuciones no paramétricas. El primer tipo supone distribuciones gaussianas multivariantes para todas las variables, lo que permite modelar todas las distribuciones como funciones analíticas con pocos parámetros (un vector de medias y una matriz de covarianzas). En este grupo se encontrarían la familia de filtros de Kalman. Por otro lado, empleando distribuciones no paramétricas, se encuentra el filtro de partículas, una alternativa flexible que permite discernir entre las distintas configuraciones de un mecanismo y que no necesita conocer el estado inicial del sistema. El algoritmo y funcionamiento del FP se podrá encontrar con detalle en el apartado **3.3.3**.

3.3.2. Modelos probabilísticos

El primer término que implementar es $p(\mathbf{o_t}|\mathbf{x_t}, \ddot{\mathbf{z}}_t)$, el *observation likelihood*. Primero, se denotará como L el número de sensores instalados en el mecanismo, que pueden ser del tipo que sean, mientras sus medidas $\mathbf{o_t}$ puedan ser modeladas de acuerdo con la siguiente expresión:

$$\mathbf{o_t} = \begin{bmatrix} \mathbf{o_t^1} \\ \vdots \\ \mathbf{o_t^L} \end{bmatrix} = \begin{bmatrix} h^{\mathbf{1}}(\mathbf{x_t}, \ddot{\mathbf{z}}_t) + n_t^1 \\ \vdots \\ h^{\mathbf{L}}(\mathbf{x_t}, \ddot{\mathbf{z}}_t) + n_t^L \end{bmatrix} = \mathbf{h}(\mathbf{x_t}, \ddot{\mathbf{z}}_t) + \mathbf{n_t} \tag{111}$$

Siendo $\mathbf{h}(\cdot)$ el modelo que predice las lecturas de los sensores a partir de un estado dinámico $(\mathbf{x_t}, \ddot{\mathbf{z}}_t)$, y $\mathbf{n_t}$ un ruido gaussiano aditivo que modela los errores de medida, calibración, perturbaciones eléctricas, etc. El ruido de cada sensor será independiente con aquel de los otros sensores, pero para generalizar, se puede suponer que la distribución del ruido sigue una gaussiana multivariante con matriz de covarianza $\boldsymbol{\Sigma_s}$, es decir:

$$\mathbf{n_t} \sim N(0, \boldsymbol{\Sigma_s}) \tag{112}$$

Siendo $N(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ una distribución gaussiana multivariante con media $\boldsymbol{\mu}$ y covarianza $\boldsymbol{\Sigma}$.

Para derivar la expresión del *likelihood* es necesario información adicional que será mencionada en el apartado **3.3.3**, concretamente si los valores $\mathbf{x_t}$ y $\ddot{\mathbf{z}}_t$ tienen asociados alguna incertidumbre o no. Como el estimador usado será un filtro de partículas, dichos valores serán hipótesis conocidas. De esta manera, todas las variables de entrada del modelo $\mathbf{h}(\cdot)$ son perfectamente conocidas y no añaden incertidumbres adicionales al ruido de la predicción del sensor, quedando la expresión final del *likelihood* como:

$$p(\mathbf{o_t}|\mathbf{x_t}, \ddot{\mathbf{z}}_t) = N(\mathbf{h}(\mathbf{x_t}, \ddot{\mathbf{z}}_t), \boldsymbol{\Sigma_s}) \tag{113}$$

A continuación, se implementará el modelo probabilístico del movimiento, utilizando las coordenadas generalizadas.

Las ecuaciones de movimiento utilizadas por un estimador recursivo serán una función que, para cada estado cinemático y dinámico del sistema, devolverá las aceleraciones correspondientes. Es decir:

$$\ddot{\mathbf{z}}_t = \mathbf{f}(\mathbf{x_t}, \mathbf{Q_t}) + \mathbf{v_t} \tag{114}$$

Donde $\mathbf{v_t}$ es un ruido gaussiano aditivo con covarianza $\boldsymbol{\Sigma_a}$ que modela los efectos de la fricción (que no es modelada), de fuerzas internas y externas inesperadas, pequeños errores en los parámetros físicos del mecanismo, etc. Se espera que $\mathbf{f}(\cdot)$ sea una función que relacione las aceleraciones del sistema con su estado $\mathbf{x_t}$ y las fuerzas aplicadas $\mathbf{Q_t}$, por ejemplo:

$$\mathbf{f}(\mathbf{x_t}, \mathbf{Q_t}) = \left(\mathbf{R^T M R}\right)^{-1} \mathbf{R^T}(\mathbf{Q_t} - \mathbf{MSc}) \tag{115}$$

Siendo $\mathbf{M}$ la matriz de masas del sistema, $\mathbf{R}$ una matriz de cambio de velocidades dependientes e independientes ($\dot{\mathbf{q}} = \mathbf{R}\dot{\mathbf{z}}$) y $\mathbf{Sc}$ un producto que relaciona $\ddot{\mathbf{q}}$ y $\ddot{\mathbf{z}}$**,** y tiene que ver con aceleraciones relativas entre los sistemas de referencia.

De nuevo, como $\mathbf{x_t}$ y $\mathbf{Q_t}$ no tendrán asociadas incertidumbres por ser hipótesis conocidas del estimador utilizado, por lo que la única fuente de incertidumbre será la función de ruido $\mathbf{v_t}$ y la expresión final del modelo probabilístico de movimiento será:

$$p(\ddot{\mathbf{z}}_t|\ \mathbf{x}_t, \mathbf{Q}_t) = N(\mathbf{f}(\mathbf{x}_t, \mathbf{Q}_t), \boldsymbol{\Sigma}_a) \tag{116}$$

Respecto a la ecuación **110**, se distinguen dos términos, el primero de ellos, $p(\mathbf{x_t}|\mathbf{x_{t-1}}, \ddot{\mathbf{z}}_{t-1})$, representa el modelo de transición del sistema: define como el estado $\mathbf{x}$ evoluciona a lo largo del tiempo afectado por las aceleraciones $\ddot{\mathbf{z}}$, mediante la integración numérica de las coordenadas independientes correspondientes a cada uno de los pasos t. Teniendo en cuenta otra vez que las variables involucradas serán hipótesis conocidas del estimador, un modelo simple y determinista puede ser utilizado, donde la transición sigue una delta de Dirac:

$$p(\mathbf{x_t}|\mathbf{x_{t-1}}, \ddot{\mathbf{z}}_{t-1}) = \delta(\mathbf{x_t} - \hat{\mathbf{x}}_t) \tag{117}$$

Donde $\hat{\mathbf{x}}_t$ es el producto de la integración numérica de $\mathbf{x_{t-1}}$ y $\ddot{\mathbf{z}}_{t-1}$, y $\delta(\cdot)$ es la delta de Dirac, una FDP que cumple la siguiente propiedad:

- $\delta(a) = 1$, cuando $a = 0$.
- $\delta(a) = 0$, en caso contrario.

El segundo término de la ecuación **110**, $p(\mathbf{x_{t-1}}, \ddot{\mathbf{z}}_{t-1}|\ \mathbf{Q_{1:t-1}}, \mathbf{o_{1:t-1}})$, representa la distribución *a posteriori* del paso anterior $t-1$, utilizada como distribución *a priori* en el paso actual. Este término no necesita implementación, pues, habiendo modelado todos los términos anteriores, se comienza la estimación utilizando las condiciones iniciales como distribución *a priori* durante el primer paso.

3.3.3. Algoritmo del filtro de partículas

Como ya se ha comentado anteriormente, un FP emplea distribuciones no paramétricas para modelar las distribuciones de probabilidad vistas en el apartado **3.3.1**. Inicialmente, como establece el IS, estas distribuciones objetivo (*target*) se modelarán con un conjunto finito de hipótesis ponderadas, llamadas partículas. El objetivo del FP es aproximar la distribución objetivo con la densidad ponderada de partículas en el espacio de estados posibles. En condiciones ideales, todos los pesos serán prácticamente idénticos ya que todas las partículas serán igual de significativas. Entonces, se puede considerar un conjunto de M partículas x, a partir de una distribución de importancia $p(x)$, de forma que:

$$x^{[i]} \sim p(x); \quad i = \{1, \cdots, M\} \tag{118}$$

Donde $x^{[i]}$ es el conjunto de partículas extraídas a partir de $p(x)$, asociadas a un conjunto de pesos $w^{[i]}$. La FDP entonces puede ser aproximada como una combinación de deltas de Dirac para el estado de cada partícula:

$$p(x) \approx \sum_{i=1}^{M} w^{[i]} \delta\left(x - x^{[i]}\right) \tag{119}$$

De esta manera, la densidad de partículas tenderá hacia la FDP objetivo cuando el número de partículas tienda a infinito. Esto significa que el conjunto de partículas se puede considerar una muestra estadística obtenida de la FDP objetivo. A nivel práctico, no todas las partículas tendrán el mismo peso (el caso ideal), y la FDP objetivo no será conocida (pues es la función que se quiere estimar), por lo que las partículas deberán ser distribuidas uniformemente a lo largo del espacio de estados posibles, y actualizar sus pesos conforme a evaluaciones puntuales realizadas, empleando las lecturas los sensores (aquí cobra importancia el *observation likelihood*, **ecuación 113**). Esta nueva densidad ponderada de partículas convergerá hacia la FDP deseada conforme el número de partículas aumente. La **Figura 18** muestra todas las distribuciones de hipótesis y FDPs comentadas en este párrafo:

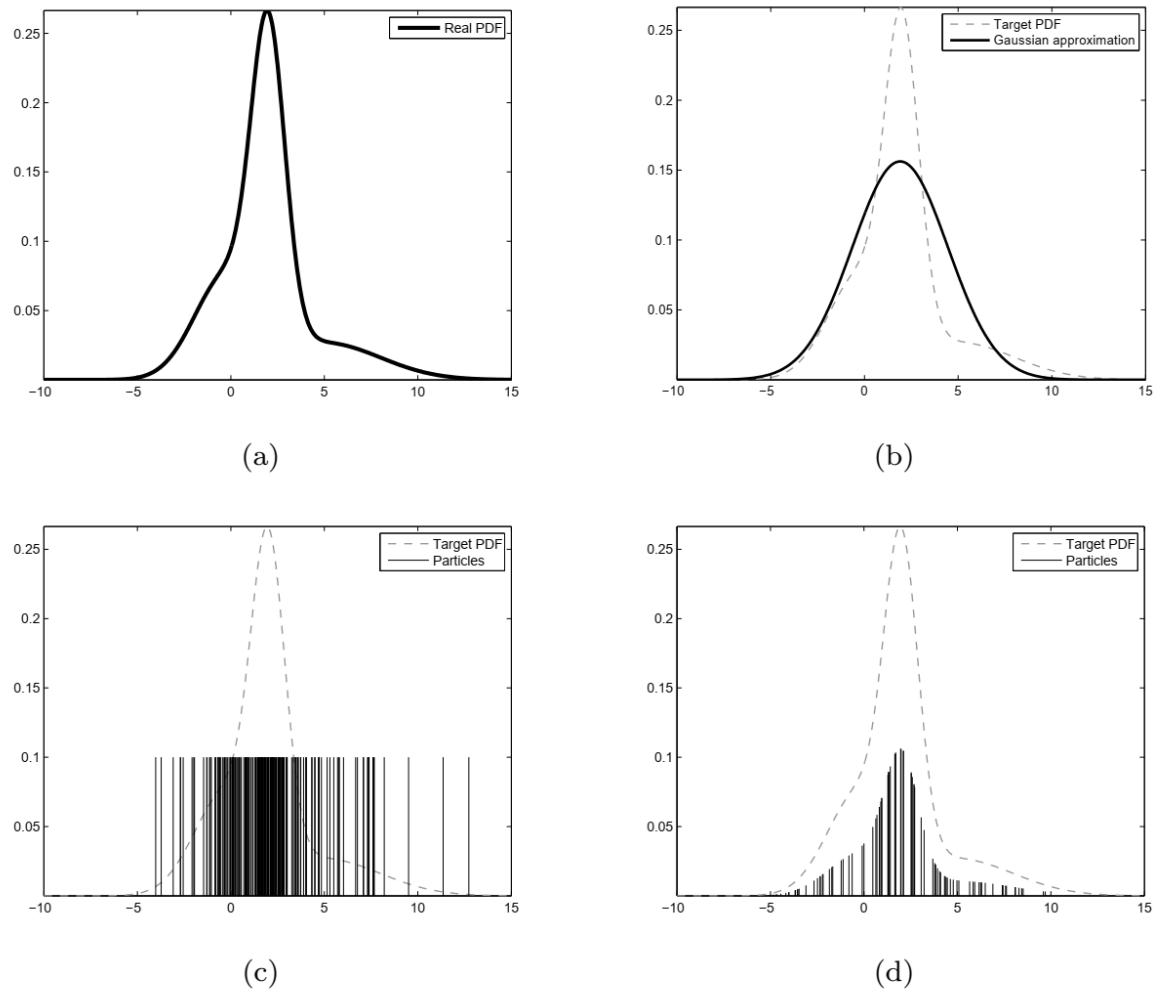

Figura 18. Distribuciones y FDPs para un espacio de estados unidimensional. El eje de abscisas representa los estados posibles mientras que el eje de ordenadas representa los pesos de cada partícula. La FDP objetivo es mostrada en (a), mientras que (b) sería una aproximación gaussiana, como la empleada por las técnicas que utilizan distribuciones paramétricas. (c) y (d) muestra dos aproximaciones empleando distribuciones no paramétricas con partículas. Mientras que (c) representaría el caso ideal de partículas con pesos idénticos, (d) representa un caso práctico en el que la FDP no es conocida y los pesos son mayores alrededor de la media. (Fuente: *[2]*)

Para usar el conjunto de partículas como estimador de estado, es necesaria una ecuación de filtrado recursiva que permita propagar la predicción de estado efectuada por el filtro de un paso al siguiente. Una de las ecuaciones propuestas más simples puede verse a continuación:

$$p(\mathbf{x_{t+1}}|\mathbf{o_{1:t+1}}) \propto p(\mathbf{o_t}|\mathbf{x_{t+1}}) \int_{-\infty}^{\infty} p(\mathbf{x_{t+1}}|\mathbf{x_t}) p(\mathbf{x_t}|\mathbf{o_{1:t}}) d\mathbf{x_t} \tag{120}$$

Cuyos términos son similares a los vistos en las **ecuaciones 109 y 110**, aunque no son exactamente iguales. Su implementación requiere la aplicación de los principios del IS a la **ecuación 110**, donde, la integración numérica se aplicará a cada una de las partículas (**ecuación 117**) y, la FDP *a posteriori*, como se vio en la **ecuación 119**, se puede representar como combinación de deltas de Dirac:

$$p(\mathbf{x_{t+1}}|\,\mathbf{Q_{1:t+1}}, \mathbf{o_{1:t+1}}) = \int \int_{-\infty}^{\infty} \delta\left(\mathbf{x_{t+1}} - \hat{\mathbf{x}}_{\mathbf{t+1}}^{[i]}\right) \sum_{i=1}^{M} w^{[i]} \delta\left(\{\mathbf{x_t}, \ddot{\mathbf{z}}_t\} - \left\{\mathbf{x}_{\mathbf{t}}^{[i]}, \ddot{\mathbf{z}}_{\mathbf{t}}^{[i]}\right\}\right) d\mathbf{x_t}, d\ddot{\mathbf{z}}_t \tag{121}$$

Que se puede expresar finalmente como:

$$p(\mathbf{x_{t+1}}|\,\mathbf{Q_{1:t+1}}, \mathbf{o_{1:t+1}}) = \sum_{i=1}^{M} w^{[i]} \delta\left(\{\mathbf{x_{t+1}}, \ddot{\mathbf{z}}_{t+1}\} - \left\{\mathbf{x}_{\mathbf{t+1}}^{[i]}, \ddot{\mathbf{z}}_{\mathbf{t+1}}^{[i]}\right\}\right) \tag{122}$$

La implementación final de la ecuación de filtrado permite obtener un estimador de estado funcional, técnica conocida como SIS, mencionada en el apartado **2.4**. Como también se comentó en dicho apartado, la varianza de los pesos crecerá a lo largo del tiempo, dando lugar a que la mayoría de las partículas tengan un peso nulo y que prácticamente una única partícula tenga un peso de la unidad.

Para evitar esta degeneración del filtro, se introdujo la técnica SIR, que propone realizar un remuestreo o *resample* para remplazar las partículas con poco peso por copias de las partículas más probables. Este proceso de *resampling* se puede ver como una función que tome el conjunto de partículas y pesos al finalizar un determinado paso t y devuelva un conjunto de partículas actualizadas para llevar a cabo el paso $t+1$. Es necesario introducir otro concepto para saber cuándo realizar el *resample*: el tamaño efectivo de la muestra o *effective sample size* (ESS). El ESS, también denotado por n_{eff}, representa el número de partículas significativas del conjunto, que aportan información relevante al problema. El ESS puede ser calculado de la siguiente forma:

$$n_{eff} = \frac{\left(\sum_{i=1}^{M} w^{[i]}\right)^2}{\sum_{i=1}^{M} {w^{[i]}}^2} \tag{123}$$

Expresión que, en el caso de utilizar pesos normalizados, puede simplificarse:

$$n_{eff} = \left(\sum_{i=1}^{M} {w^{[i]}}^2\right)^{-1} \tag{124}$$

Cuando el valor del ESS caiga por debajo de un cierto umbral (establecido por el usuario, un ejemplo podría ser $M/2$) será momento de realizar un *resample*.

Fue discutido anteriormente que uno de los problemas con las técnicas que emplean distribuciones paramétricas son incapaces de estimar un estado cuyas condiciones iniciales se desconocen. La razón es que, en ese sentido, este tipo de estimadores imponen fuertes restricciones iniciales con objeto de aproximar las distribuciones a gaussianas. En cambio, una de las ventajas de un FP es su capacidad para tratar con ambigüedades y con la falta de información sobre el estado del sistema.

Matemáticamente, cada una de las componentes de las partículas para el instante $t = 0$, es decir, $\mathbf{z}_0^{[i]}$, $\dot{\mathbf{z}}_0^{[i]}$ y $\ddot{\mathbf{z}}_0^{[i]}$, será generada como una muestra aleatoria a partir de una distribución que, generalizando y suponiendo que todos los posibles estados son igual de probables, será una distribución uniforme $U(a, b)$ tal que cualquier punto $x \in (a, b)$ tendrá la misma densidad de probabilidad. Por tanto, las condiciones iniciales en un problema general de estimación con FP serán:

$$\mathbf{z}_0^{[i]} \sim U(\mathbf{z}_{min}, \mathbf{z}_{max}) \tag{125}$$

$$\dot{\mathbf{z}}_0^{[i]} \sim U(\dot{\mathbf{z}}_{min}, \dot{\mathbf{z}}_{max}) \tag{126}$$

$$\ddot{\mathbf{z}}_0^{[i]} \sim U(\ddot{\mathbf{z}}_{min}, \ddot{\mathbf{z}}_{max}) \tag{127}$$

Donde $(\mathbf{z}_{min}, \mathbf{z}_{max})$, $(\dot{\mathbf{z}}_{min}, \dot{\mathbf{z}}_{max})$ y $(\ddot{\mathbf{z}}_{min}, \ddot{\mathbf{z}}_{max})$ son los intervalos de valores iniciales posibles de posición, velocidad y aceleración generalizadas. Para problemas multidimensionales, el número de partículas requerido para cubrir el espacio de estados posibles sería excesivo, y computacionalmente no sería viable su aplicación a un problema de estimación en tiempo real. Sin embargo, para un caso unidimensional como el que se va a tratar en este trabajo, el FP es una solución robusta y razonable.

Para finalizar, se sintetizará toda la información vista en esta sección con el objetivo de describir el algoritmo SIR final para la estimación con FP. Utilizaremos como base el fragmento en pseudocódigo mostrado en la **Figura 19:**

- Previamente, deberá haberse definido los vectores de partículas $\mathbf{s}$ y de pesos $\mathbf{w}$, y las condiciones iniciales. Estas consistirán en repartir el estado cinemático y dinámico de las partículas a lo largo del espacio de estados posibles (**ecuaciones 124-126**), y asignar pesos iguales a cada una de ellas (esto es, $w^{[i]} = M^{-1}$, para todo i hasta M).
- Se realizará una llamada al procedimiento "FiltradoDePartículas", pasando como variables de entrada los vectores de partículas y pesos.
- Se calculará, para cada una de las partículas, su estado cinemático posterior, mediante el procedimiento "IntegraciónNumérica" (correspondiente a la **ecuación 117**), su estado dinámico (aceleraciones) posterior, mediante "ModProbMovimiento" (**ecuación 116**) y su peso modificado, mediante "ActualizarPesos" (**ecuación 113**). Se construye también el vector de partículas para el paso posterior.
- Se normalizan los pesos de las partículas, para que sumen la unidad.
- Se calcula el ESS, y si este cae por debajo del umbral del 50% de partículas, se efectúa un *resample*.
- Aquí finaliza el procedimiento "FiltradodePartículas", que será llamado otra vez con los valores posteriores calculados en la iteración anterior, y así, sucesivamente.

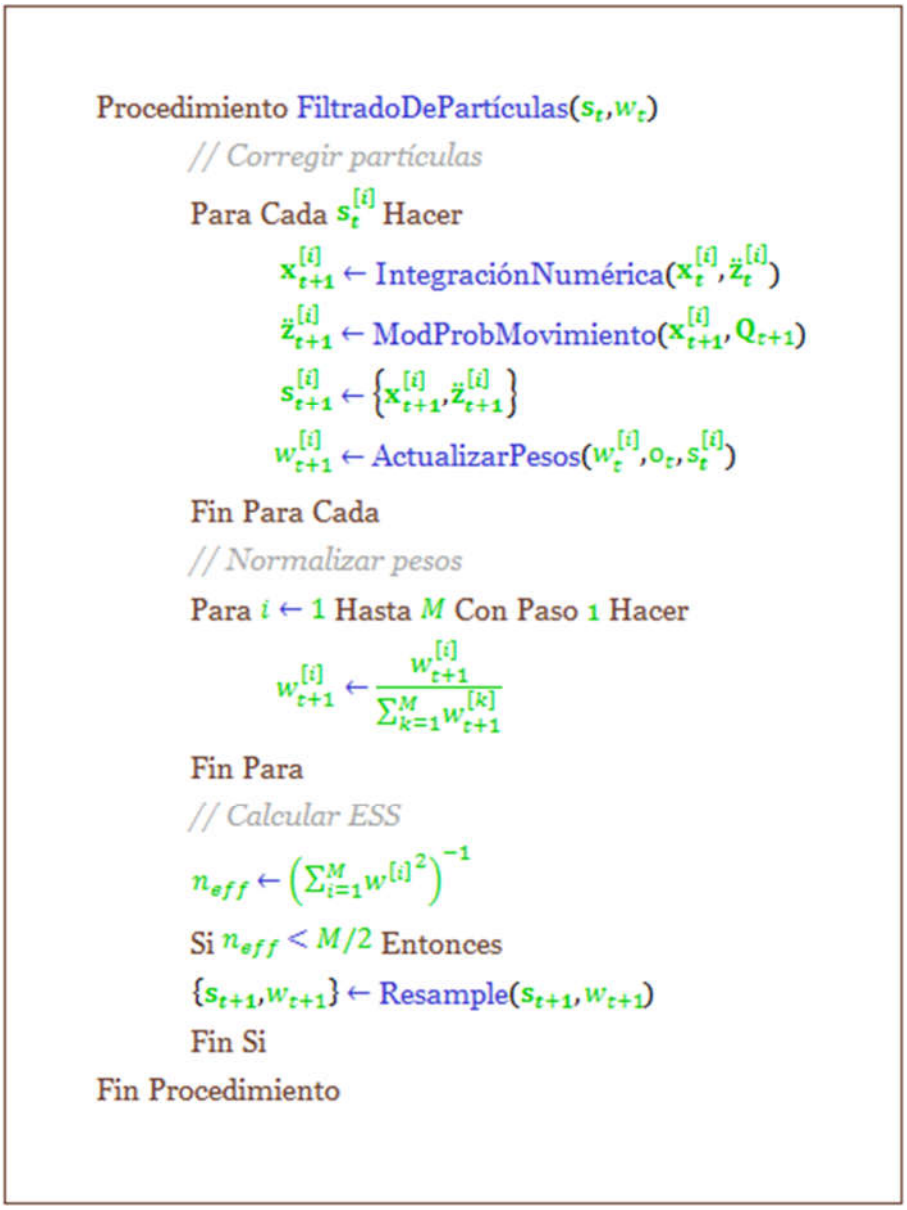

Figura 19. Algoritmo SIR de filtrado de partículas, escrito en pseudocódigo. El procedimiento descrito se encargará de propagar, tanto las partículas como sus pesos, de un paso t al paso siguiente $t+1$.

Capítulo 4:

Resultados y discusión

4. Resultados y discusión

La mayor parte del esfuerzo y el tiempo dedicado en este trabajo se ha destinado al desarrollo de una librería en C++ para implementar estimadores de estado, basados en FPs. Esta librería utiliza Simbody para la simulación física de sistemas multicuerpo, por lo que, siguiendo la línea de Simbody a la hora de nombrar sus librerías y paquetes internos de software, se ha decidido nombrar el producto de este trabajo como "SimTKpf". Para evaluar su potencial como librería para la implementación de un estimador de estado funcional, se ha decidido estimar la posición de un mecanismo sencillo: un mecanismo de cuatro barras, un problema unidimensional que debería poder llevarse a cabo en tiempo real con filtros de partículas [2].

Dicho mecanismo será completamente simulado, así como el sensor acoplado cuyas lecturas serán empleadas para corregir el filtro. Como consecuencia, ha sido necesario desarrollar una librería adicional para la simulación de mecanismos de cuatro barras, utilizando las capacidades que ofrece Simbody. Esta segunda librería se ha llamado "Fourbar", y su código fuente, así como el de SimTKpf, se encontrará disponible en un repositorio en Github [31], con código abierto.

Ambas librerías se dividen en archivos de código fuente en C++ (con extensión "**.cpp**") y archivos de cabecera (con extensión "**.h**"). Los primeros contienen definiciones de métodos de clase, miembros estáticos y funciones principales; mientras que los segundos contienen identificadores, declaraciones y definiciones de funciones inline y plantillas, que no pueden ser definidos en los archivos de código fuente.

SimTKpf no necesita instalación para poder ser utilizada, aunque sí será necesario que Simbody haya sido instalado previamente y que su ruta se haya añadido a las variables de entorno del SO.

4.1. SimTKpf

El archivo de cabecera principal de SimTKpf es "**SimTKpf.h**". Incluyendo dicho archivo al inicio de un programa es suficiente para utilizar todas las utilidades tanto de SimTKpf como de Simbody.

Uno de los objetos más importantes de la librería es la clase *ParticleDynState*, que representa una de las muchas partículas utilizadas por el filtro. Su nombre hace referencia a que es una partícula con estado dinámico, y, realmente, no es más que un objeto *State* de Simbody asociado a una variable de tipo double para representar su peso logarítmico (esto es, $ln\, w$), además de algunos operadores y *Setters & Getters*, que permiten el acceso y escritura tanto del peso como del *State* de la partícula.

ParticleDynState dispone de dos constructores. El primero es un constructor personalizado, que recibe como entradas un *State* y un valor de peso, y los usa para asignar sus variables miembro. Si se emplea este constructor, se puede omitir el peso y este se asignará como cero, pero no se puede omitir el *State* y pasar como variable solo el peso. El segundo es el constructor por defecto, que no toma ninguna variable. El peso se asigna como cero, y se llama al constructor por defecto del objeto *State*, que crea un Estado vacío, sin variables de estado. Cuando se utiliza el constructor por defecto, antes de realizar operaciones con el Estado de la partícula, se debe realizar una llamada a la función *setStateDefault(),* que recibe como entrada el objeto *MultibodySystem* e inicializa el Estado de la partícula. Su efecto realmente es asignar al Estado el retorno de la función *MultibodySystem*::*getStateDefault()*.

Sin embargo, *ParticleDynState* no es una clase pensada para su utilización directa por el usuario. En su lugar, se define la clase *ParticleList*, cuya estructura incluye un vector de *ParticleDynState*, una variable double que almacenará el ESS del conjunto y diversos métodos de clase.

ParticleList dispone de tres constructores. El primero, su constructor predeterminado, mantendrá el vector de partículas vacío. El segundo constructor recibe como entrada el número de partículas, es decir, la longitud del vector (técnicamente recibe como entrada una variable de tipo size_t). El vector es redimensionado para contener el número de elementos indicado y se llama al constructor por defecto de cada uno de ellos. El tercer constructor toma directamente un vector de partículas, y lo asigna a su variable miembro. Si el Estado de las partículas queda vacío, hay que procurar inicializarlo. *ParticleList* no cuenta con una método para ello, pero permite acceder a las partículas individualmente y de esta manera, a sus funciones miembro. El ESS, en cualquier caso, es asignado como cero cuando se construye un objeto *ParticleList*.

ParticleList cuenta con un método para propagar el estado de las partículas. La función *advanceStates()* se encarga de ello, recibiendo como entrada un objeto *TimeStepper* de Simbody y el intervalo de tiempo que se usará como paso. Este *TimeStepper* implementa los modelos de ecuación probabilística de movimiento e integración numérica (salvo por la función de ruido gaussiana) vistos en las ecuaciones **116 y 117**. *ParticleList* también permite normalizar pesos (linealmente, para que $\sum_{i=1}^{M} w^{[i]} = 1$, o logarítmicamente, efectuando un cambio de escala para que el mayor $ln\, w^{[i]}$ valga cero), asignar pesos iguales (linealmente, asignando M^{-1}, o logarítmicamente, asignando cero $ln\, w^{[i]} = 0$ para todo i) o calcular el ESS como decimal, empleando la **ecuación 123** y dividiendo por M.

Por último, *ParticleList* implementa una función de *resample*, mediante selección y remplazamiento: se normalizan los pesos logarítmicamente, se construye una función de distribución acumulada de pesos $F(x)$ y una función escalonada uniforme $G(x)$, y se seleccionan las partículas tal que $F(i) = G(j)$, siendo j el índice del vector de partículas escogidas para su posterior remplazamiento.

Sin embargo, si se decide utilizar el objeto *ParticleList* para implementar un FP, hay que tener en cuenta algunas consideraciones. Se ha mencionado anteriormente que la función *advanceStates()* implementa la **ecuación 116** sin la función de ruido gaussiano: el usuario debería encargarse de definirla (sencillo, pues Simbody implementa funciones gaussianas y uniforme) y aplicarla a las variables de estado de las partículas.

ParticleList tampoco incorpora una función para actualizar pesos (**ecuación 113**), por lo que debe hacerse manualmente. SimTKpf ofrece una función para evaluar el valor de la función normal, llamada *LogNormalProb()*, la cual deberá recibir como entrada el valor de la lectura del sensor acoplado al mecanismo, la desviación de dicho sensor y la predicción (conocida como *belief*) de cada una de las partículas. Para realizar esta predicción, SimTKpf incorpora las clases *Measuring_Instrument* y *Simbody_Instrument*, para la implementación de sensores simulados. Ambas son clases abstractas a partir de las cuales crear la clase derivada que represente el sensor deseado, disponen de una variable double para almacenar la lectura del instrumento, de una función *read()* que devuelve dicha variable y de una función *measure()* puramente virtual que debe sobrecargarse con un método para cambiar el valor de la lectura.

La diferencia entre ellas radica en que *Simbody_Instrument* es un clase derivada de *Measuring_Instrument*, cuyo constructor solicita como entradas un objeto *State* (el Estado del mecanismo a medir), un objeto *MultibodySystem* (el Sistema necesario para actualizar el Estado), un objeto *SimbodyMatterSubsystem* (el Subsistema de materia que contiene la topología y los cuerpos del sistema) y un valor double que representa la desviación del instrumento y será utilizado para añadir ruido gaussiano a la lectura. De esta manera, *Simbody_Instrument* supondrá un paso intermedio en la implementación de sensores reales para medición en un SMC, mientras que *Measure_Instrument* es una clase más genérica con mayor campo de aplicación.

Un ejemplo es la clase *Stopwatch* de SimTKpf, un cronometro simulado que permite medir intervalos de tiempo tanto real como de uso de la CPU, aprovechando las capacidades de Simbody con un uso más intuitivo. El constructor de la clase *Stopwatch* recibe como entrada únicamente una variable enumerada *StopwatchMode* cuyos valores son *Real_Time* o *CPU_Time*; e inicializa la lectura a cero. La clase dispone de los siguientes miembros:

- La variable double heredada de *Measure_Instrument* que almacena la lectura.
- Una variable double donde internamente se almacena la referencia para el instante de tiempo cero.
- La función *start()*, que inicializa la referencia y sirve para comenzar o reanudar la cuenta.
- La función *measure()* heredada de *Measure_Instrument*, definida para añadir a la lectura la diferencia entre el instante actual y el de la referencia.
- La función *stop()*, que simplemente llama a *measure()*, pero resulta más intuitiva e indica al usuario que se ha detenido la cuenta.
- La función *restart()*, que resetea la lectura a cero.

Teniendo todo esto en cuenta, y volviendo de nuevo a los inconvenientes que supone el uso de *ParticleList*, SimTKpf ofrece un último objeto, *ParticleFilter*, la representación completa de un FP que reúne en una única clase todas los métodos y variables que se debían definir por separado cuando se hacía uso de *ParticleList*. La finalidad de *ParticleFilter* es reducir el esfuerzo por parte del usuario final, de forma que, una vez definido el FP, solo necesite preocuparse por la construcción del SMC (apartado **3.2.14**) y la elaboración del bucle de control, ya que para el *time stepping* del Estado de referencia, en caso de que el mecanismo sea simulado, SimTKpf define la función *advance()*, cuyo comportamiento es idéntico al de *ParticleList::advanceStates()*.

El constructor de *ParticleFilter* acepta dos variables de entrada: una de tipo size_t con el número de partículas que manejará el filtro y un objeto de tipo *PF_Options*, que reúne cinco valores de tipo double actualmente. Dichos valores son, por orden:

- **SIMULATION_TIME_STEP**: intervalo de tiempo transcurrido entre pasos del filtro.
- **SENSOR_STDDEV**: desviación del sensor utilizado por cada partícula, cuya lectura se tomará como predicción en la **ecuación 113**.
- **SENSOR_STDDEV_MOD**: constante que multiplica el valor de SENSOR_STDDEV, y puede modificarse en tiempo real. Su objetivo es aumentar la desviación del sensor cuando el filtro confía demasiado en sus predicciones (*overconfident*), o disminuirla, cuando no es lo suficientemente restrictivo al contrastar sus propias predicciones (*underconfident*).
- **MOTION_STDDEV**: desviación de la función de ruido gaussiano aplicada en el modelo probabilístico de movimiento, definido en la **ecuación 116**.
- **RESAMPLE_STDDEV**: desviación de la función de ruido gaussiano aplicable a las variables de estado de las partículas tras el *resample*, para diferenciarlas entre sí y que no sean copias exactas.

Para evitar duplicación de código innecesaria, los *Setters & Getters* de *PF_Options* toman una variable enumerada llamada *PF_Options_index*, que indicará cuál de las opciones anteriores será escrita, leída o actualizada.

Un objeto *ParticleFilter* almacena su *PF_Options* correspondiente y un *ParticleList*, el cual construye pasando como variable el número de partículas. *ParticleFilter* no solo define todas las funciones de *ParticleList* para poder hacer uso de ellas directamente, sino que, además, suple sus carencias. Implementa la función *updateStates()*, que recibe como entradas un *TimeStepper* y un *Assembler*, y se ocupa de avanzar el estado de las partículas, añadir un ruido gaussiano a las coordenadas generalizadas y realizar un análisis de ensamblado. También implementa una función de propagación de pesos, *updateWeights()*, que tiene la particularidad de ser una plantilla, pues necesita recibir como entrada cualquier clase derivada de *Measure_Instrument* definida por el usuario para que cada partícula pueda realizar una predicción (en conjunción con la función *updateStates()*). La función además recibe como entrada un dato de tipo double representando el valor empírico con el que se contrastarán las predicciones de cada partícula (denominado *Ground truth*), que puede provenir de un sensor en un mecanismo real o de un sensor simulado, como el que posee cada partícula.

Por último, remarcar que se define una excepción (*PF_Options_exception*), en el caso de utilizar un *PF_Option_index* incorrecto durante el uso de los *Setter & Getters* de la clase *ParticleFilter*.

4.2. Fourbar

De manera equivalente a SimTKpf, Fourbar hace uso de un archivo de cabecera principal, "**Fourbar.h**". Dicho archivo se debe incluir al inicio de un programa para utilizar todas sus funcionalidades (incluye "**SimTKpf.h**", por lo que no es necesario añadir ni SimTKpf ni Simbody).

El archivo más importante de la librería es "**Fourbar.cpp**", pues es el primer archivo nombrado que contiene una simulación en Simbody y la implementación completa del algoritmo para la estimación de estado con FP; y al ser compilado, generará un archivo ejecutable homónimo.

Una vez se definen completamente las variables utilizadas (entre ellas, un *ParticleFilter*), se procede a la creación del SMC. Este proceso de creación abarca la declaración del Sistema principal (*MultibodySystem*) y la de sus Subsistemas (*SimbodyMatterSubsystem* y *GeneralForceSubsystem*), la adición de un campo gravitacional (*Force::UniformGravity*) al Subsistema de fuerzas, la definición de las propiedades de cada cuerpo (*Body*), la creación del cuerpo y su movilizador (*MobilizedBody*) y la adición de restricciones (*Constraint*). El modelo del sistema cuenta, tan solo de dos cuerpos: la barra de entrada y la barra libre, siguiendo la propuesta de Simbody de definir el menor número de movilidades y restricciones. La barra de salida y la condición de mecanismo cerrado se han conseguido mediante una restricción de distancia constante entre el último nudo de la barra libre y el punto correspondiente geométricamente al nudo donde confluyen las barras fija y de salida, mientras que la barra fija viene definida por la distancia entre el nudo del que parte la primera barra y el punto de la restricción anterior. Su representación gráfica mediante el visualizador de Simbody puede verse en la siguiente figura:

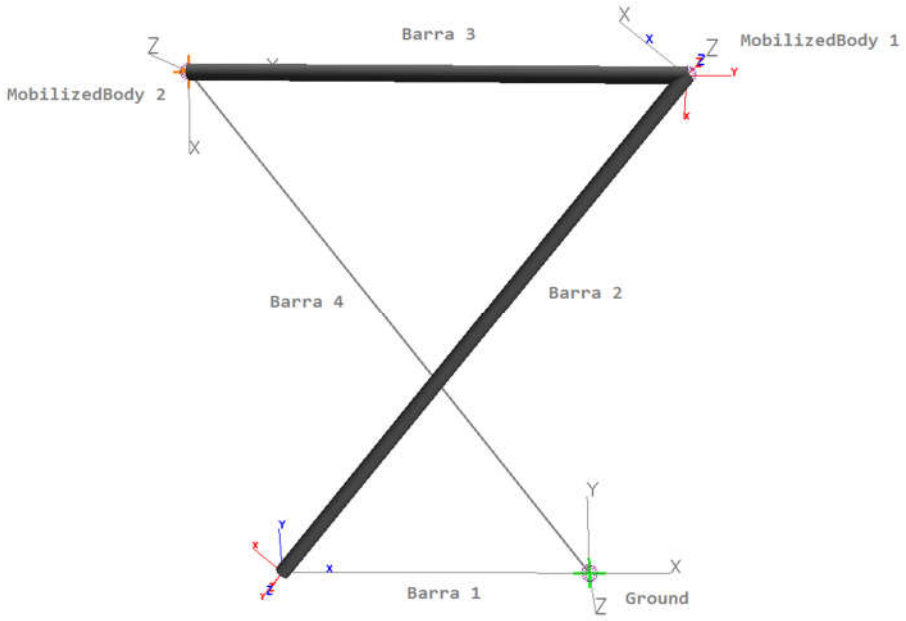

Figura 20. Topología utilizada en las simulaciones de este trabajo. La barra número 2 podría haberse modelado con otra restricción de longitud constante, pero se habría perdido la coordenada generalizada correspondiente a su ángulo, y resultaría más complicado posicionar el mecanismo en la posición deseada.

A continuación, se define un *Assembler* de Simbody y se realiza una llamada a *assemble_Fourbar()*, función perteneciente a Fourbar. Esta función se encargará del análisis de ensamblado, personalizado para el caso del mecanismo de cuatro barras. Su funcionamiento es simple: determina los límites del ángulo de rotación de la barra de entrada y, mediante una distribución uniforme entre dichos valores, fija la posición de la barra y calcula un conjunto de coordenadas generalizadas que cumplan las restricciones impuestas. Se deben pasar como variables el *Assembler* que realizará el análisis, el Estado cuyas coordenadas generalizadas se utilizarán, el Sistema que inicializará el Estado y un objeto *GrashofCondition*, definido por Fourbar, que determina los límites del ángulo de rotación.

Los miembros de *GrashofCondition* son un string que describe la configuración del mecanismo, y un bool que representa si la barra de entrada puede girar 360º completos. Su constructor acepta dos variables de los mismos tipos, pero la forma de inicializar un *GrashofCondition* es con la función *evaluateGrashof()*, que construye todas las posibles opciones y devuelve aquella correspondiente al vector de longitudes de las barras que recibe como entrada. Estas opciones dependen de la categoría a la que pertenezcan las barras de entrada y salida [26]:

- ***Crank***: manivela que puede girar los 360º completos.
- ***Rocker***: balancín que puede girar un intervalo que no incluye ni 0º ni 180º.
- ***0-Rocker***: balancín que puede girar un intervalo que incluye 0º pero no 180º.
- ***π-Rocker***: balancín que puede girar un intervalo que incluye 180º pero no 0º.

La condición final se evalúa calculando los siguientes términos:

$$T_1 = g + h - a - b \tag{128}$$

$$T_2 = b + h - a - h \tag{129}$$

$$T_3 = b + h - a - g \tag{130}$$

Donde g, h, a y b son las longitudes, respectivamente, de las barras fija, libre, de entrada y de salida (*ground*, *floating*, *input* y *output*). La configuración correspondiente puede consultarse en la siguiente tabla:

T_1	T_2	T_3	Configuración (Entrada – Salida)
−	−	+	Crank – Crank
+	+	+	Crank – Rocker
+	−	−	Rocker – Crank
−	+	−	Rocker – Rocker
−	−	−	0-Rocker – 0-Rocker
−	+	+	π-Rocker – π-Rocker
+	−	+	π-Rocker – 0-Rocker
+	+	−	0-Rocker – π-Rocker
0	0	0	Crank – Crank (Mecanismo plegable)

Tabla 4. Configuraciones de un mecanismo de cuatro barras. En las cuatro primeras opciones entran los comunmente denominados "Mecanismos de cuatro barras de Grashof", mientras que la última es un caso especial en el que el mecanismo puede doblarse de manera que sus cuatro barras se encuentren dispuestas a lo largo de una linea recta.

Tras el análisis de ensamblado, se establecen pesos iguales para todas las partículas, se definen el integrador y el *time stepper*, y una serie de objetos de tipo *Gyroscope*, que simulan

un giroscopio acoplado en la barra de entrada del mecanismo para medir su velocidad angular. Es una clase derivada de *Simbody_Instrument*, que, además de los parámetros heredados del constructor de *Simbody_Instrument* necesita de un dato de tipo double que represente la longitud de la barra fija del mecanismo, necesaria para calcular las posiciones cartesianas de los nudos del mecanismo. *Gyroscope* sobrescribe la función *measure()*, y determina la velocidad angular de la siguiente manera:

$$\omega = \frac{\mathrm{r} \times \mathrm{v}}{|\mathrm{r}|^2} = -\frac{(\mathrm{v}_2 - \mathrm{v}_1) \times (\mathrm{p}_2 - \mathrm{p}_1)}{|\mathrm{p}_2 - \mathrm{p}_1|^2} \tag{131}$$

Donde ω es el vector velocidad angular, $\mathrm{r} = \mathrm{p}_2 - \mathrm{p}_1$ es el vector posición de un nudo del mecanismo a su contiguo (en cuya barra se encuentra el giróscopo), y $\mathrm{v} = \mathrm{v}_2 - \mathrm{v}_1$ el vector velocidad de la barra, calculado como diferencia de las velocidades de los nudos en un instante dado. Hay que destacar que la segunda expresión permitirá, en función de las posiciones y velocidades cartesianas de los nudos, calcular ω con su sentido correcto, siguiendo la regla de la mano derecha. Las magnitudes cartesianas serán obtenibles a partir de $\mathbf{q}$ y $\mathbf{u}$ actualizando el Estado hasta la etapa de Velocidad.

Tanto el Estado que simulará el *Ground truth* como el Estado de cada una de las partículas tendrá asociado un *Gyroscope* único. Posteriormente, comenzaría el algoritmo iterativo de estimación de estado: se avanzan el Estado de referencia y de las partículas, se toman las lecturas de los giróscopos, se actualizan los pesos y se calcula el ESS. Si este es menor que 0.5, se efectúa el *resample* y se suma un ruido a la velocidad generalizada correspondiente a la primera barra del mecanismo (al variar la velocidad, indirectamente variarán las posiciones en la iteración posterior).

A continuación, se muestran dos capturas del archivo en ejecución:

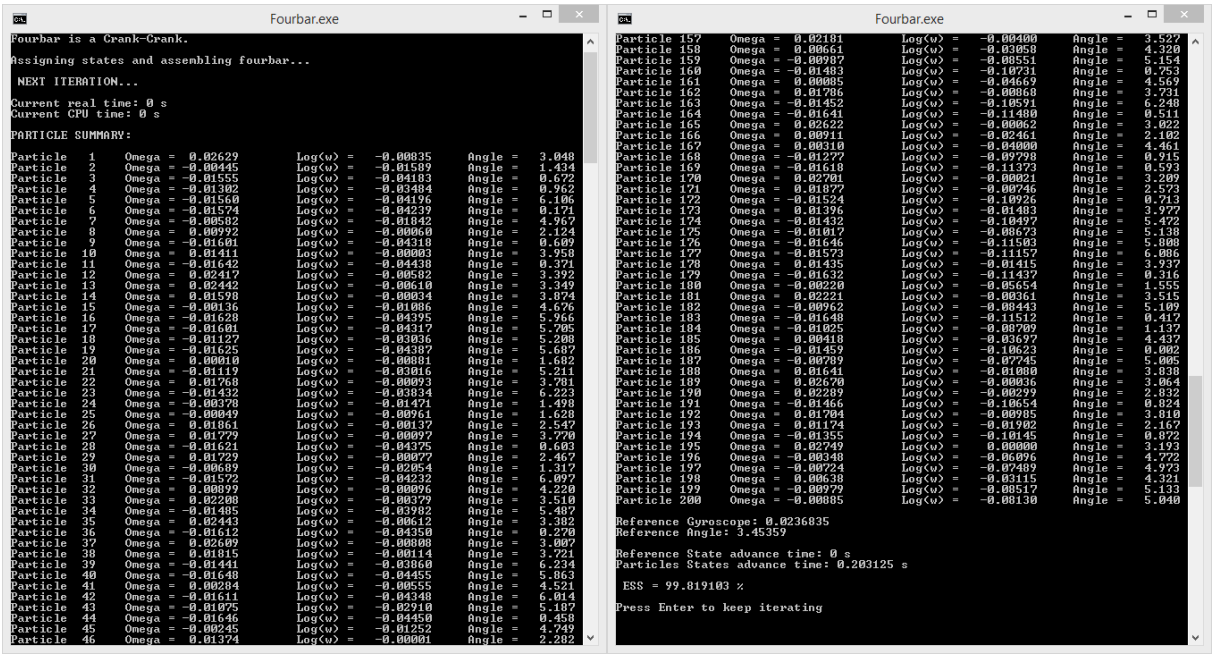

Figura 21. Capturas de Fourbar.exe al inicio de su ejecución, tras realizar una llamada sin argumentos.

A lo largo de este proceso recursivo, como se aprecia en la **Figura 21**, se imprime por pantalla información relativa al tiempo transcurrido (real y simulado), al ángulo, peso y velocidad

angular de los distintos Estados y a los pasos realizados por el programa. Cabe destacar que el archivo generado a partir de **Fourbar.cpp** admite el uso de "*flags*", argumentos pasados en el momento de realizar una llamada, para modificar la ejecución del programa. Los *flags* definidos hasta el momento en **Fourbar.cpp** son "*--quiet*", "*--nopause*" y "*--txtwrite*", que modificarán la salida de información por pantalla, las pausas al final de cada iteración y la recopilación de datos en ficheros de texto con formato **.txt**. Este último identificador habilitará el uso de las funciones *Angle_write()*, *Weight_write()* y *Omega_write()*, que generarán una nueva línea en archivos de texto individuales con valores de ángulos, pesos y lecturas de los giróscopos, respectivamente. Estos pueden usarse para análisis posteriores, representación gráfica o incluso depuración de la misma librería.

Existen otros dos archivos cuya compilación genera ejecutables, **Fourbar_StdDevTest.cpp** y **Fourbar_StdDevTest.cpp**, pero su estructura y propósito serán comentar durante la siguiente sección.

4.3. Análisis

Una vez explicadas ambas librerías, el siguiente paso es verificar su robustez, precisión y viabilidad a la hora de ser aplicada a un caso práctico.

Primero se realizará un ensayo en el cual será medido el tiempo real transcurrido para un tiempo simulado establecido. Dicho tiempo se puede dividir en tres procesos críticos de la simulación: la transición del estado de referencia utilizado como *Ground truth*, la transición del estado de cada una de las partículas, y el *resample*. Estos tiempos se pueden obtener utilizando la clase Stopwatch que ofrece *SimTKpf*.

El número de partículas se mantendrá constante (N=200), así como el resto de los parámetros, y se medirá, para una serie de pasos:

- El tiempo simulado transcurrido (se elegirá uno alrededor de 6 segundos).
- El tiempo real transcurrido (o tiempo de CPU).
- El mayor tiempo transcurrido durante la transición del estado de referencia en una iteración (el peor caso posible).
- El mayor tiempo transcurrido durante la transición del estado de las partículas en una iteración (el peor caso posible).
- El mayor tiempo transcurrido durante el *resample* en una iteración (el peor caso posible).

Los pasos escogidos han sido 6, 9, 12, 18, 25, 50, 100, 200, 300, 500 y 1000 milisegundos. El resultado ha sido representado gráficamente en la **Figura 22**, con doble escala logarítmica para su correcta visualización.

Analizando la primera gráfica, puede observarse que el tiempo real transcurrido es bastante mayor del deseado para pasos pequeños del orden de 10 ms, pero que, sin embargo, para un determinado número de partículas, existirá un paso para el que la simulación se ejecute en tiempo real, alrededor de 290 ms para 200 partículas, aunque lo prudente sería dejar cierto margen (un paso alrededor de un tercio de segundo, en este caso). Esto ocurre porque el tiempo real transcurrido disminuye logarítmicamente respecto al paso.

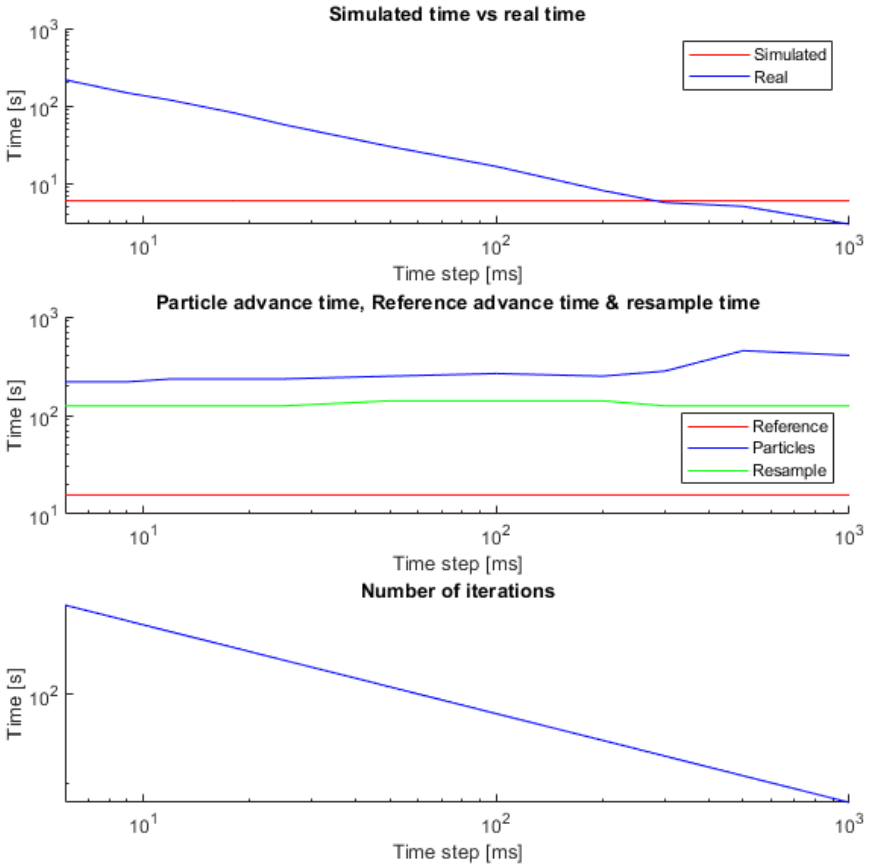

Figura 22. Gráfica de tiempos transcurridos durante la simulación para una serie de pasos.

De la segunda gráfica se deduce que el tiempo transcurrido en cada iteración es casi constante, independientemente del paso transcurrido, por lo que, para un tiempo de simulación establecido, el tiempo real disminuirá al incrementar el paso de simulación (mayor tiempo simulado por segundo). Un dato que destacar es que el tiempo de transición del *Ground truth* máximo es siempre 15,63 ms, mientras que el de las partículas es siempre un múltiplo de este, y menor que el número de partículas. La explicación podría ser que, al realizarse la transición del Estado por medio del *time stepper* y su integrador, estemos midiendo el tiempo tomado por un paso interno.

Curiosamente, el tiempo de transición del *Ground truth* medido era cero en ciertas ocasiones, lo que puede deberse a que un solo paso interno del integrador cruce varios pasos de simulación. Esto explicaría el incremento del tiempo de transición de las partículas al aumentar el paso: no todas las partículas requieren un paso interno del integrador durante una iteración, pero su frecuencia aumenta conforme lo hace el paso de simulación. El tiempo de *resample* ha sido casi constante durante todo el ensayo, aumentando o disminuyendo ligeramente de manera aleatoria. El motivo se puede deber a mejores o peores casos del algoritmo de ordenación y sustitución.

En la última gráfica se ha plasmado el número de iteraciones (tiempo simulado dividido por el paso de simulación). Su forma es similar a la gráfica de tiempo real, confirmando la idea de que el tiempo transcurrido, para un número de partículas constante, depende exclusivamente del número de iteraciones realizadas.

En el siguiente ensayo, comprobaremos la eficacia del estimador para distintos números de partículas, así como para distintos valores de la desviación del ruido gaussiano aplicado en el modelo de transición (La opción de *PF_Options* llamada MOTION_STDDEV). Aquí entran en juego los archivos **Fourbar_StdDevTest.cpp** y **Fourbar_StdDevTest.cpp**, comentados en la sección anterior. Ambos comparten la estructura de **Fourbar.cpp** al comienzo, pero definen un vector de valores de número de partículas y de ruido gaussiano de transición, respectivamente, modificando el valor de dichas variables de manera dinámica a lo largo de la simulación. Para cada valor del vector, se realizan un determinado número de simulaciones (determinado por la variable *tries*), al final de las cuales se comprobará si el filtro ha convergido, es decir, si el ángulo de la barra de entrada de la suma de las hipótesis ponderadas se encuentra dentro de un umbral alrededor del ángulo del *Ground truth*.

Presumiblemente, a mayor paso de simulación, mayor número de iteraciones será necesario hasta que sea posible la convergencia del filtro. Por este motivo, se elegirá un paso sin tener en cuenta el resultado del ensayo anterior. Se elegirá 6 ms de paso y 1020 ms de simulación (correspondiente a 170 pasos), como tiempo más que suficiente para comprobar la convergencia del filtro.

Primero, se muestra los resultados del ensayo de número de partículas, para 10, 20, 50, 100, 200, 500, 1000 y 2000 partículas, en la siguiente figura:

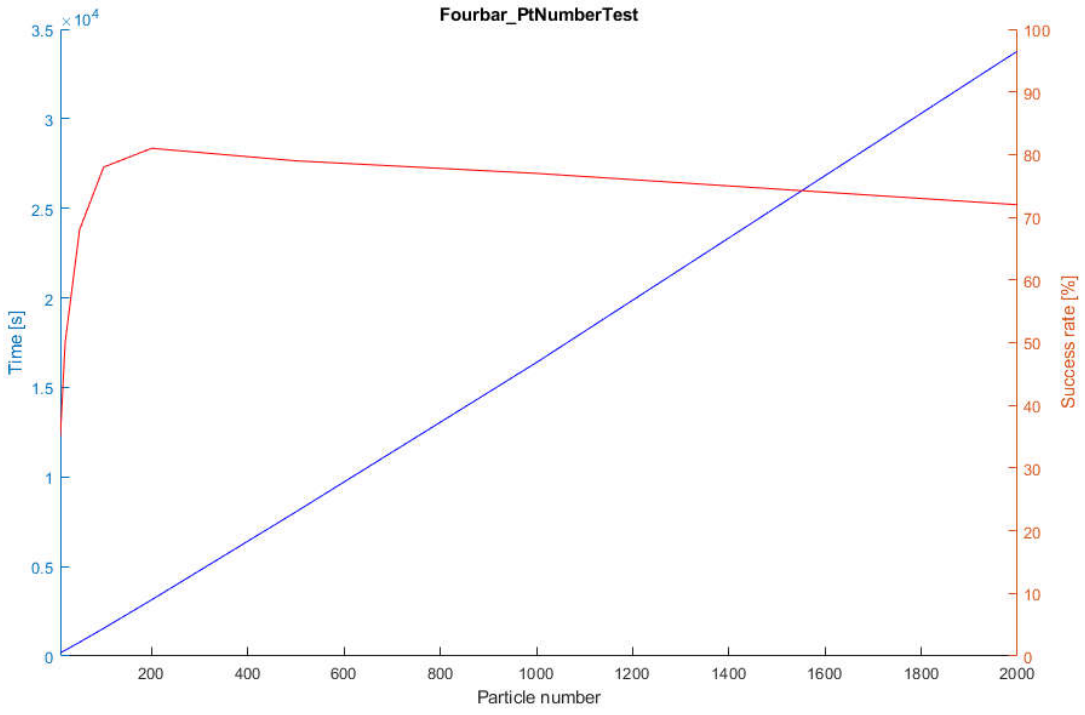

Figura 23. Número de convergencias y tiempo de ensayo según el número de partículas.

Como se puede observar, el porcentaje de convergencia aumenta drásticamente en cuanto el número de partículas aumenta ligeramente y cubre el espacio de estados posibles (los 360º de un ángulo completo). No obstante, este porcentaje desciende levemente a partir de un cierto número de partículas, posiblemente porque existe una mayor varianza de hipótesis, que provoca que el filtro todavía no converja en el tiempo de ensayo establecido. El tiempo de simulación es directamente proporcional al número de partículas, con una complejidad de O(*n*), para *n* partículas.

A continuación, los resultados del ensayo para la desviación del ruido de transición, representados gráficamente en la siguiente figura:

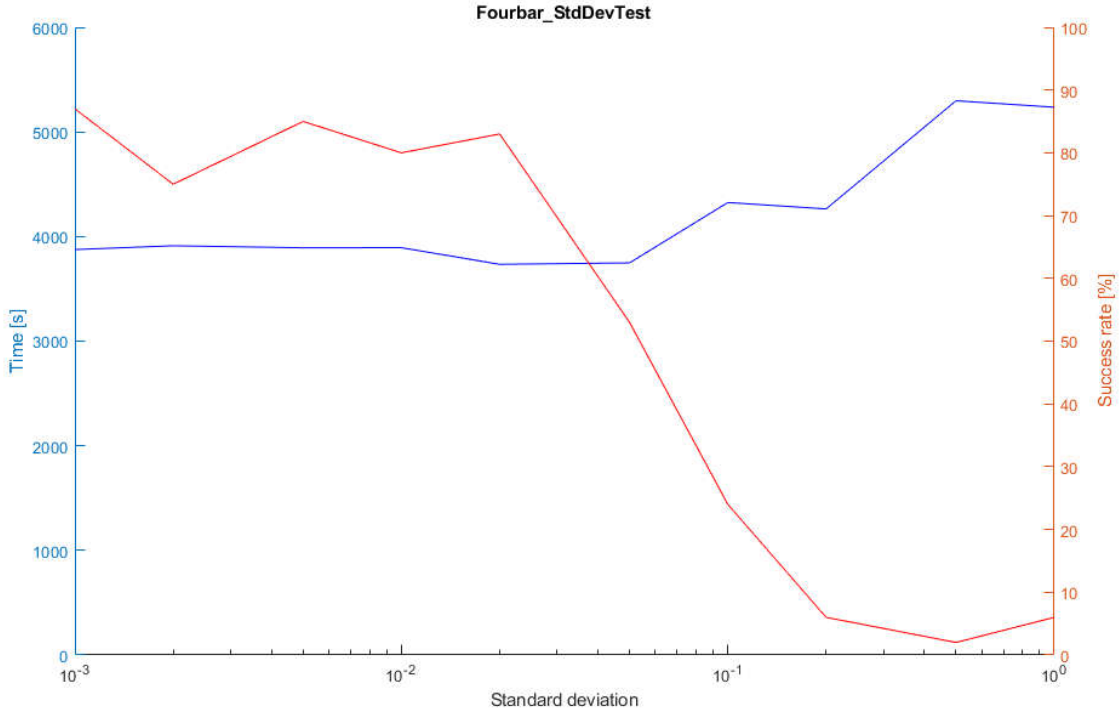

Figura 24. Número de convergencias y tiempo de ensayo según la desviación de transición.

Se han representado valores de 10^{-3}, 2×10^{-3}, 5×10^{-3}, 10^{-2}, 2×10^{-2}, 5×10^{-2}, 10^{-1}, 2×10^{-1}, 5×10^{-1} y 1 rad.

El tiempo de ensayo aumenta a partir de un cierto valor (5×10^{-2}). La causa es que la varianza de partículas es tan elevada que las partículas más proximas al mecanismo de referencia en un instante, no lo son en el siiguiente instante, provocando la degeneración del filtro y un mayor números de *resamples* al disminuir el ESS. Por este mismo motivo, el porcentaje de convergencia disminuye a partir del mismo valor.

A continuación, vamos a representar la trayectoria del mecanismo simulado, así como de las distintas partículas, a lo largo del tiempo, para unos valores de la desviación del modelo de transición de 0 y de 2×10^{-2} radianes, un número de partículas de 200 y un paso de 6 ms. El resultado puede verse en la **Figura 25**.

Como puede verse en dicha figura, cuando la desviación es cero (mecanismo ideal, sin rozamiento, posición posterior de las partículas determinable con exactitud en todo momento), el filtro converge de forma perfecta rápidamente. Cuando la desviación es distinta de cero, se puede apreciar que la convergencia es algo más lenta, y que siempre existe un cierto umbral de partículas alrededor del ángulo del mecanismo estimado. También observamos como este umbral se ensancha en los instantes donde la velocidad del mecanismo disminuye. Este efecto se debe a que las variaciones de ángulo son muy pequeñas de una iteración a otra, y el filtro no puede predecir una posición concreta, aumentando la varianza de partículas en dichos momentos.

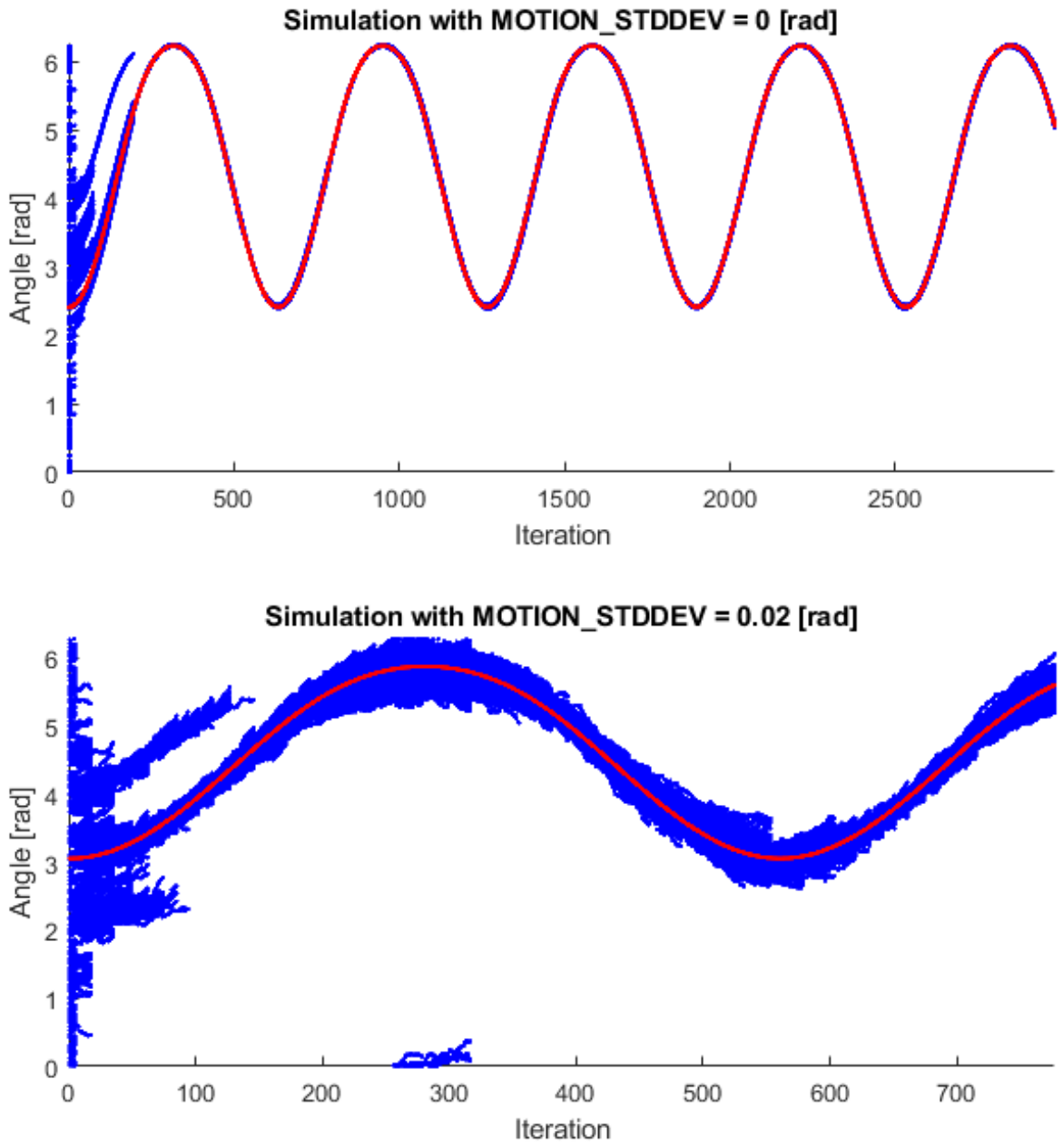

Figura 25. Trayectoria del mecanismo cuatro barras simulado (rojo), así como la distribución de partículas (azul). En cada uno de los casos, la posición del mecanismo de referencia era escogida aleatoriamente.

Por último, comentar de nuevo que la topología utilizada en todos los ensayos es exactamente la mostrada en la **Figura 20**, es decir, longitudes de las barras fija, de entrada, libre y de salida, respectivamente, de 2, 4, 3 y 4 metros (por el sistema de unidades utilizado).

Capítulo 5:

Conclusiones y trabajos futuros

5. Conclusiones

En vista de los resultados obtenidos en el capítulo anterior, se confirma que el filtro de partículas es una técnica muy flexible, que converge a pesar de que ni la posición del mecanismo ni su configuración son conocidas a priori, en contraposición a la familia de filtros de Kalman, que imponen unas condiciones iniciales muy estrictas.

En cambio, se puede apreciar que es una técnica más costosa computacionalmente. Inicialmente, se pensaba usar un paso temporal de 6 ms durante la simulación, pero en el estado en el que se encuentra la librería actualmente, el tiempo de ejecución resulta considerablemente mayor que el tiempo que se pretende simular, alrededor de 40 y 50 veces mayor. Como solución, se ha establecido un paso mayor, de 300 ms, para poder realizar una estimación en tiempo real con cierto margen de holgura. El mecanismo de cuatro barras es un mecanismo con un grado de libertad y, por tanto, su espacio de estados es unidimensional. Sin embargo, su aplicación en tiempo real se ha visto comprometida por la implementación del FP, a pesar de todas las decisiones de diseño de la librería con el propósito de aumentar su eficiencia, y de todas las optimizaciones posteriores realizadas. Como consecuencia, su aplicación en tiempo real para problemas multidimensionales resulta completamente inviable.

A pesar de todo, se puede decir que se ha cumplido el objetivo de desarrollar una librería que permita la implementación del FP para estimar el estado de un mecanismo sencillo. La librería, sin embargo, tiene mucho potencial de mejora, y su eficiencia, sin ninguna duda, puede ser optimizada todavía más. Se sugieren a continuación varias propuestas de mejora de la librería que, por la carga lectiva que supone este trabajo, no han podido llevarse a cabo, pero al menos serán comentadas, ordenadas de mayor importancia a menor:

- Reescribir la librería para hacer uso de los eventos de Simbody, una utilidad muy útil que se usó en los comienzos de la librería junto al visualizador de Simbody, pero que posteriormente fue eliminada. El bucle utilizado actualmente para la inferencia recursiva podría ser sustituido por una única llamada al *time stepper*, y tanto la transición de los Estados de referencia y de las partículas, como la actualización de pesos, entre otros procedimientos, podrían ser realizados mediante eventos programados. La ecuación activadora de evento responsable del *resample* sería muy sencilla: $\mathrm{n}_{\mathrm{eff}} - 0{,}5 = 0$.
- Aprovechando que se ha mencionado el visualizador de Simbody, resultaría interesante encontrar una manera de incorporarlo en la simulación para visualizar en tiempo real el estado de las partículas. No es posible actualmente ya que no se ha encontrado la manera de representar varios Estados diferentes en una sola ventana de simulación, de forma simultánea. Una alternativa que se probó fue definir un Sistema con un mecanismo para cada partícula y un solo Estado (en contraposición con el actual, que usa un solo mecanismo y un Estado para cada partícula), pensando que, en lugar de muchas simulaciones cortas para pocos cuerpos, sería más eficiente realizar una única simulación algo más larga con muchos cuerpos al mismo tiempo, pero el tiempo de ejecución aumentaba de manera prohibitiva con el número de partículas.
- Visualizar (con ayuda de los scripts de MATLAB) los casos en los que el filtro no converge en los ensayos de número de partículas y desviación del ruido de transición, para encontrar el motivo por el que el filtro no converge el 100% de las veces.

- Aprovechando la capacidad de Simbody para realizar multiplicaciones matriz-vector con O(n) y matriz-matriz con $O(n^2)$, averiguar si es posible sustituir en el código algún bucle por un sistema matricial, para reducir el tiempo de ejecución al máximo. Se considera empezar por los métodos que contienen la clase *ParticleDynState*, la función de *resample*, las sumas de ruido aditivo y las funciones de asignar, normalizar y actualizar pesos.
- Implementar un procedimiento para *assemble_Fourbar()* que funcione en el caso de mecanismos de cuatro barras en los que la barra de entrada no puede girar los 360º completos. Este procedimiento debería determinar los límites de giro de la barra. Se ha probado a ensamblar el mecanismo en incrementos de 1º y ver en qué casos el ángulo "post-ensamblado" no coincide con el ángulo de "pre-ensamblado", pero es muy costoso computacionalmente y no funciona.
- Durante el desarrollo del algoritmo de filtro de partículas, se utilizaba *ParticleList* en las simulaciones, en lugar de *ParticleFilter*, que posteriormente se creó para reunir en una sola clase todos los procedimientos necesarios para tratar las partículas. *ParticleList* quedó como una clase redundante, por lo que se propone eliminarla y reunir todos los procedimientos comentados en *ParticleFilter*.
- En algunas partes concretas, la librería está influenciada por el objetivo de implementar un estimador en un mecanismo de cuatro barras, como en la función *updateStates()*, donde se suma un ruido aditivo a la primera coordenada generalizada, que, en un movilizador de torsión, como el empleado para el mecanismo de cuatro barras, corresponde al ángulo. En el *resample* ocurre algo similar, añadiendo el ruido a la primera velocidad generalizada. Modificar dichas partes mediante plantillas para hacerlas más genéricas.
- Una vez generalizado, incorporar el *resample* a la clase *ParticleFilter*.
- Actualmente, el funcionamiento de *Gyroscope::measure()* y *assemble_Fourbar()* depende de la manera exacta en la que se haya modelado el mecanismo (orden de las barras, planos y direcciones en los que se ha posicionado, horizontalidad, etc.). La solución permite dos alternativas: generalizar dichas funciones (más complicado), o definir toda la topología para que el usuario no tenga que hacerlo, a través de una llamada a una función a la que se le pasen todos los parámetros necesarios (más sencilla, pero restrictiva).
- Generalizar aún más la clase *Measure_Instrument*, convirtiéndola en una plantilla cuyas lecturas puedan ser de cualquier tipo, como enteros o booleanos, no solo double.
- Definir un generador de números aleatorios global para todos los ruidos aditivos, haciendo innecesarias la variable estáticas de la clase *Simbody_Instrument* y las distribuciones gaussianas correspondientes a cada uno de los ruidos.
- Crear un espacio de nombres (*namespace*) para todas las clases y métodos, para organizarlas mejor y evitar conflictos, al igual que hace Simbody. Podría elegirse o no ser el mismo (SimTK).
- Eventualmente, crear un directorio de Tests que sean ejecutados la primera vez que se compile la librería, como hace Simbody, y que sea controlable a través de una opción de compilación en Cmake.
- Una vez la librería Fourbar sea totalmente independiente de la implementación concreta en **Fourbar.cpp**, crear un directorio de ejemplos y colocar en él dicho archivo. Otra opción de compilación enCmake puede añadirse en este caso.
- Reunir los scripts de MATLAB del repositorio en un directorio como se ha hecho con todos los archivos de SimTKpf y de Fourbar. Adicionalmente, convertirlos en funciones para que puedan ser llamados con parámetros variables.

Un posible trabajo futuro muy interesante podría ser aplicar la librería para la estimación de un mecanismo de cuatro barras real. Dicho trabajo ayudaría a ampliar la librería, agregando clases y métodos para la lectura de sensores reales, y a generalizarla aún más, para englobar aquellos casos en los que no se utilice un *Ground truth* simulado. Esto implicaría volver opcional la definición y transición de un Estado de referencia, así como permitir que la simulación realice una pausa hasta que se produzca la adquisición de datos.

Este trabajo podría tratar con más detalle el proceso de ajuste de todas las variables del filtro para conseguir una estimación lo más eficaz posible. En este trabajo, dicho proceso, conocido en la literatura como *tuning*, se ve reflejado en los archivos **Fourbar_StdDevTest.cpp** y **Fourbar_PtNumberTest.cpp**, que evalúan la eficacia del filtro teniendo en cuenta el número de partículas y la dispersión del modelo de transición de las partículas. Los resultados podrían compararse con los obtenidos en este trabajo, siendo de especial interés el efecto entre el uso de un mecanismo simulado y uno real.

Bibliografía

[1] E. Sanjurjo, M. Á. Naya, J. L. Blanco-Claraco y J. L. Torres-Moreno, «Accuracy and efficiency comparison of various nonlinear Kalman filters applied to multibody models,» *Nonlinear Dynamics,* vol. 88, nº 3, pp. 1935-1951, 2017.

[2] J. L. Blanco, J. L. Torres y A. Giménez-Fernández, «Multibody dynamic systems as Bayesian Networks: applications to robust state estimation of mechanisms,» *Multibody System Dynamics,* vol. 34, nº 2, pp. 103-128, 2015.

[3] S. Y. Chen, «Kalman Filter for Robot Vision: A Survey,» *IEEE Transactions on Industrial Electronics,* vol. 59, nº 11, pp. 4409-4420, 2012.

[4] C. Guardiola, S. Hoyas, B. Pla y D. Blanco-Rodríguez, «Solución analítica de un filtro de Kalman estacionario para la observación de deriva en modelos de emisiones de NOx en motores diesel de automoción,» *Revista Iberoamericana De Automática e Informática Industrial,* vol. 12, nº 2, pp. 230-238, 2015.

[5] J. L. Torres-Moreno, J. L. Blanco-Claraco, A. Giménez-Fernández, E. Sanjurjo y M. Á. Naya, «Online Kinematic and Dynamic-State Estimation for Constrained Multibody Systems Based on IMUs,» *Sensors (Switzerland),* vol. 16, nº 3, p. 333, 2016.

[6] Z. Fan, H. Ji y Y. Zhang, «Iterative particle filter for visual tracking,» *Signal Processing: Image Communication,* vol. 36, pp. 140-153, 2015.

[7] S. Li, S. Zhao, B. Cheng, E. Zhao y J. Chen, «Lightweight Particle Filter for Robust Visual Tracking,» *IEEE Access,* vol. 6, pp. 32310-32320, 2018.

[8] M. L. Felis, «RBDL: an efficient rigid-body dynamics library using recursive algorithms,» *Autonomous Robots,* vol. 41, nº 2, pp. 495-511, 2017.

[9] J. Wojtusch, J. Kunz y O. von Stryk, «MBSlib-An Efficient Multibody Systems Library for Kinematics and Dynamics Simulation, Optimization and Sensitivity Analysis,» *IEEE Robotics and Automation Letters,* vol. 1, nº 2, pp. 954-960, 2016.

[10] M. A. Sherman, A. Seth y S. L. Delp, «Simbody: multibody dynamics for biomedical research,» *Procedia IUTAM,* vol. 2, pp. 241-261, 2011.

[11] Z. Gao, I. Gibson, C. Ding, J. Wang y J. Wang, «Virtual Lumbar Spine of Multi-Body Model Based on Simbody,» *Procedia Technology,* vol. 20, pp. 26-31, 2015.

[12] A. A. Zobova, T. Habra, N. Van der Noot, H. Dallali, N. G. Tsagarakis, P. Fisette y R. Ronsse, «Multi-physics modelling of a compliant humanoid robot,» *Multibody System Dynamics,* vol. 39, nº 1, 2017.

[13] E. Catto, «Box2D | A 2D Physics Engine for Games,» [En línea]. Available: https://box2d.org/. [Último acceso: 23 Octubre 2019].

[14] «Gazebo,» [En línea]. Available: http://gazebosim.org. [Último acceso: 23 Octubre 2019].

[15] E. Coumans, «Bullet Physics SDK: real-time collision detection and multi-physics simulation for VR, games, visual effects, robotics, machine learning etc.,» [En línea]. Available: https://pybullet.org/wordpress/. [Último acceso: 23 Octubre 2019].

[16] «SimTK: Welcome,» [En línea]. Available: https://simtk.org/. [Último acceso: 20 Julio 2019].

[17] «Simbody: Multibody Physics API: Project Home - SimTK,» [En línea]. Available: https://simtk.org/projects/simbody/. [Último acceso: 2019 Julio 2019].

[18] «Simmath: SimTK Core Mathematical Software: Project Home - SimTK,» [En línea]. Available: https://simtk.org/projects/simmath/. [Último acceso: 20 Julio 2019].

[19] «SimTKcommon: essential objects for the SimTK Core toolkit: Project Home - SimTK,» [En línea]. Available: https://simtk.org/projects/simtkcommon. [Último acceso: 20 Julio 2019].

[20] «OpenSim: Project Home - SimTK,» [En línea]. Available: https://simtk.org/projects/opensim. [Último acceso: 20 Julio 2019].

[21] «High-performance C++ multibody dynamics/physics library for simulating articulated biomechanical and mechanical systems like vehicles, robots, and the human skeleton,» Github, [En línea]. Available: https://github.com/simbody/simbody. [Último acceso: 20 Julio 2019].

[22] «cplusplus.com - The C++ Resources Network,» [En línea]. Available: www.cplusplus.com. [Último acceso: 21 Julio 2019].

[23] B. Stroustrup, The C++ Programming Language, 1985.

[24] B. Stroustrup y M. A. Ellis, The Annotated C++ Reference Manual, 1990.

[25] A. Avello, Teoría de Máquinas, 2da Edición, Navarra: Tecnun - Universidad de Navarra, 2014.

[26] J. M. McCarthy y G. S. Soh, Geometric Design of Linkages, Springer, 2010.

[27] E. Sanjurjo, J. L. Blanco, J. L. Torres y M. A. Naya, «Testing the efficiency and accuracy of multibody-based state observers,» de *ECCOMAS Thematic Conference on Multibody Dynamics*, Barcelona, Catalonia, Spain, June 29 - July 2, 2015.

[28] R. E. Kalman, «A New Approach to Linear Filtering and Prediction Problems,» *Transactions of the ASME - Journal of Basic Engineering,* vol. 82, pp. 35-45, 1960.

[29] G. Welch y G. Bishop, «An Introduction to the Kalman Filter,» TR 95-041. Department of Computer Science, Chapel Hill, University of North Carolina, 2006.

[30] J. L. Blanco Claraco, Contributions to Localization, Mapping and Navigation in Mobile Robotics, Tesis doctoral en Ingeniería en Telecomunicación. Universidad de Málaga, 2009.

[31] «Particle Filter estimators using C++ Multibody Dinamics library Simbody,» [En línea]. Available: https://github.com/r-aguilera/SimTKpf. [Último acceso: 14 Noviembre 2019].

Anexo

Anexo

Se adjuntarán en este anexo los archivos de código correspondientes a las bibliotecas SimTKpf y Fourbar, así como los scripts de MATLAB utilizados durante su desarrollo y análisis. Estos archivos corresponden al lanzamiento de la versión 1.0 del proyecto, alojado en un repositorio de Github [31].

Los ficheros adjuntos son los siguientes:

- Fourbar.h
- assemble_Fourbar.h
- Grashof_condition.h
- Gyroscope.h
- Txt_write.h
- SimTKpf.h
- Measuring_Instrument.h
- Particle_Classes.h
- Particle_Filter.h
- PF_Options.h
- PF_Utilities.h
- Simbody_Instrument.h
- Stopwatch.h
- assemble_Fourbar.cpp
- Fourbar.cpp
- Fourbar_PtNumberTest.cpp
- Fourbar_StdDevTest.cpp
- Grashof_condition.cpp
- Gyroscope.cpp
- Txt_write.cpp
- Measuring_Instrument.cpp
- Particle_Classes.cpp
- Particle_Filter.cpp
- PF_Options.cpp
- PF_utilities.cpp
- Simbody_Instrument.cpp
- Stopwatch.cpp
- TXTreadAngles.m
- TXTreadAngles_with_pause.m
- TXTreadOmegas.m
- TXTreadWeightedAngles.m

Fourbar.h

```
#ifndef _FOURBAR_
#define _FOURBAR_

#include "Simbody.h"
```

```
#include "SimTKpf.h"
#include "Fourbar/assemble_Fourbar.h"
#include "Fourbar/Grashof_condition.h"
#include "Fourbar/Gyroscope.h"
#include "Fourbar/Txt_write.h"

#endif
```

assemble_Fourbar.h

```
#ifndef _ASSEMBLE_FOURBAR_
#define _ASSEMBLE_FOURBAR_

#include "Simbody.h"
#include "Fourbar/Grashof_condition.h"
#include "SimTKpf/Particle_Filter.h"
#include "SimTKpf/PF_utilities.h"

// Assemble fourbar at a random yet attainable angle
void assemble_Fourbar(GrashofCondition&, SimTK::MultibodySystem&, SimTK::Assembler&,
SimTK::State&, ParticleFilter&);

#endif
```

Grashof_condition.h

```
#ifndef _GRASHOF_CONDITION_
#define _GRASHOF_CONDITION_

#include <string>
#include <vector>

// Grashof condition class. It will contain a description string about if input & output
bars of fourbar are crank, rocker,
// 0-rocker or Pi-rocker; and a bool containing if input bar is a crank and can fully
rotate.
// - Crank: it can fully rotate 360º.
// - Rocker: it can rotate a range which does not include 0º or 180º.
// - 0-Rocker: it can rotate a range which includes 0º degrees but not 180º.
// - Pi-Rocker: it can rotate a range which includes 180º but not 0º.
class GrashofCondition {
public:
	GrashofCondition(std::string, bool);
	std::string get_description();
	bool get_isCrank();
private:
	bool isCrank;
	std::string description;
};

// Evaluation of Grashof condition, given the bar lenghts [ground, input, floating,
output]. Returns a GrashofCondition.
GrashofCondition evaluateGrashof(std::vector<double>&);

#endif
```

Gyroscope.h

```
#ifndef _GYROSCOPE_
#define _GYROSCOPE_

#include "Simbody.h"
#include "SimTKpf/Simbody_Instrument.h"

// Class representing a virtual gyroscope measuring angular velocity
// in the first not-ground link of a four bar linkage

class Gyroscope : public Simbody_Instrument {
public:
	// Inherited constructor given the lenght of the ground bar
	Gyroscope(const SimTK::MultibodySystem&, const SimTK::SimbodyMatterSubsystem&,
SimTK::State&, double, double);

	void measure() override;

private:
	double m_GroundBarLenght;
};

#endif
```

Txt_write.h

```
#ifndef _TXT_WRITE_
#define _TXT_WRITE_

#include "Simbody.h"
#include "SimTKpf/PF_utilities.h"
#include "SimTKpf/Particle_Classes.h"
#include "Fourbar/Gyroscope.h"

// Functions to write data to txt files

void Angle_write(SimTK::State&, ParticleList&);
void Weight_write(ParticleList&);
void Omega_write(Gyroscope&, std::vector<Gyroscope>&);

#endif
```

SimTKpf.h

```
#ifndef _SIMTKPF_
#define _SIMTKPF_

#include "Simbody.h"
#include "SimTKpf/Particle_Filter.h"
#include "SimTKpf/Measuring_Instrument.h"
#include "SimTKpf/Particle_Classes.h"
#include "SimTKpf/PF_Options.h"
#include "SimTKpf/PF_utilities.h"
#include "SimTKpf/Simbody_Instrument.h"
#include "SimTKpf/Stopwatch.h"

#endif
```

Measuring_Instrument.h

```
#ifndef _MEASURING_INSTRUMENT_
#define _MEASURING_INSTRUMENT_

// General class for virtual ideal measuring instruments:

class Measuring_Instrument {
public:
	virtual void measure()=0;    // Update instrument reading. Pure virtual, must be
overriden
	virtual const double read(); // Return instrument reading

protected:
	double reading;
};

#endif
```

Particle_Classes.h

```
#ifndef _PARTICLE_CLASSES_
#define _PARTICLE_CLASSES_

#include "Simbody.h"

// PF particles will belong to this class:

class ParticleDynState {
public:
	ParticleDynState();   // Default constructor. State will keep empty

	ParticleDynState(const SimTK::State&, double);    // Constructor given a state.
Weight will be zero if unspecified

	void operator =(const ParticleDynState&);  // Operator to copy particles

	bool operator <(const ParticleDynState&);  // Operator necessary to sort particles

	void setWeight(const double&);                    // Set weight value
	const double getWeight() const;            // Get read-only weight value
	double& updWeight();                       // Get writable weight value

	void setState(const SimTK::State&);        // Set State
	const SimTK::State& getState() const;      // Get read-only State
	SimTK::State& updState();                  // Get writable State

	void setStateDefault(const SimTK::MultibodySystem&);     // Set State from system
default State

private:
	SimTK::State   state; // Particle state
	double         logw;  // Particle state weight
};
```

```
// Class for particle vectors with useful member functions to manage their weight and state
and vector size:
class ParticleList {
public:
	ParticleList();
	ParticleList(std::size_t particle_number);
	ParticleList(std::vector <ParticleDynState>& ParticleVector);

	const ParticleDynState getParticle(std::size_t) const;	// Return a read-only
particle
	const ParticleDynState operator () (std::size_t) const;	// Operator equal to
getParticle

	ParticleDynState& updParticle(std::size_t);		// Return a writable particle
	ParticleDynState& operator [] (std::size_t);		// Operator equal to updParticle

	void addParticle(const ParticleDynState&);		// Add particle to ParticleList
	void operator << (const ParticleDynState&);		// Operator equal to addParticle

	void deleteParticle();					// Delete particle in the last
position
	void deleteParticle(std::size_t);			// Delete particle in specific
position
	void deleteParticle(std::size_t, std::size_t);	// Delete particles between two
positions

	const std::vector<ParticleDynState>& getAllParticles() const;	// Return the whole
particle list as a read-only vector
	std::vector<ParticleDynState>& updAllParticles();		// Return the whole
particle list as a writable vector
	std::size_t size() const;
		// Return size of the particle list

	void advanceStates(SimTK::TimeStepper& ts, const double dt);	// Evolve all
particles State

	void setEqualWeights();		// Set equal logarithmic weights to all particles
	void setEqualLinearWeights();// Set equal linear weights that sum 1 to all particles

	void normalizeWeights();		// Normalize particles' logarithmic weights
	void normalizeLinearWeights();	// Make the particles' weight distribution a
probability distribution function (Sum must be 1):

	void calculateESS();		// Calculate the Effective Sample Size (ESS) of the
particle list
	const double getESS() const; // Return the previously calculated ESS

	void resample();		// Replace particles according to their weights

private:
	std::vector<ParticleDynState> particles;
	double ESS;
};

#endif
```

Particle_Filter.h

```
#ifndef _PARTICLE_FILTER_
#define _PARTICLE_FILTER_

#include "Simbody.h"
#include "SimTKpf/PF_Options.h"
```

```
#include "SimTKpf/Particle_Classes.h"
#include "SimTKpf/Simbody_Instrument.h"
#include "SimTKpf/PF_utilities.h"

// Main SimTKpf class, for estimation based on particle filter algorithm:
class ParticleFilter {
public:
	// Constructor given the particle number and his PF_Options
	ParticleFilter(std::size_t, PF_Options&);

	void setOptions(PF_Options);				// Write filter options values
	const PF_Options getOptions() const;		// Return filter options as read-only
PF_Options class
	PF_Options& updOptions();					// Return filter options as writable
PF_Options class

	void setOneOption(PF_Options_index, double);	// Write single option value
	const double getOneOption(PF_Options_index) const;// Return single option as read-
only double value
	double& updOneOption(PF_Options_index);			// Return single option as
writable double value

	const ParticleList getParticleList() const;		// Return particles as read-only
ParticleList class
	ParticleList& updParticleList();				// Return particles as writable
ParticleList class

	const std::vector<ParticleDynState> getParticleVec() const;	// Return particles
as read-only std::vector
	std::vector<ParticleDynState>& updParticleVec();			// Return particles
as writable std::vector

	std::size_t getParticleNumber() const;	// Return particle number value

	void setEqualWeights();					// Set equal logarithmic weights to all
particles
	void setEqualLinearWeights();		// Set equal linear weights that sum 1 to all
particles

	void normalizeWeights();			// Normalize particles' logarithmic weights
	void normalizeLinearWeights();		// Make the particles' weight distribution a
probability distribution function (Sum must be 1):

	void calculateESS();		// Calculate the Effective Sample Size (ESS) of the
particle list
	double getESS() const;		// Return the Effective Sample Size (ESS) value
	// Advance particles state and add noise.

	void updateStates(SimTK::TimeStepper&, SimTK::Assembler&);

	// Make a prediction and update weights accordingly.
	template<class customInstrument>
	void updateWeights(double, std::vector<customInstrument>&);

	// Replace particles according to their weights.
	void resample();

private:
	ParticleList Particles;
	PF_Options& Filter_Options;
};

// Template Implementation
template<class customInstrument>
void ParticleFilter::updateWeights(double GroundTruthInstr_reading,
```

```
std::vector<customInstrument>& InstrVec) {

	double Instrument_StdDev = getOneOption(PF_Options_index::SENSOR_STDDEV);
	double Modified_Instrument_StdDev = Instrument_StdDev *
getOneOption(PF_Options_index::SENSOR_STDDEV_MOD);
	double belief;

	for (std::size_t i = 0; i < Particles.size(); i++) {
		InstrVec[i].measure();
			// Update instrument reading
		belief = InstrVec[i].read();
	// Make the prediction
		Particles[i].updWeight() += LogNormalProb(belief,	// Update weights
			GroundTruthInstr_reading, Modified_Instrument_StdDev);
	}
	Particles.normalizeWeights();
}

#endif
```

PF_Options.h

```
#ifndef _PF_OPTIONS_
#define _PF_OPTIONS_

#include <exception>

// Auxiliary enumeration class for accesing a single particle filter option:
enum class PF_Options_index {
	SIMULATION_TIME_STEP,
	SENSOR_STDDEV,
	SENSOR_STDDEV_MOD,
	MOTION_STDDEV,
	RESAMPLE_STDDEV
};

// Exception in case of unexpected behaviour using PF_Option_index class:
class PF_Options_exception : public std::exception {
	virtual const char* what() const throw(){
		return "PF_Option_index used does not match existing ones. \nDefault
statement reached in switch statements of ParticleFilter.cpp methods:\nsetOption, getOption
or updOption";
	}
};

// Options class for particle filter algorithm:
class PF_Options {
public:
	PF_Options();
	// 1 - Time step between two filter iterations
	// 2 - Standard deviation of Ground Truth sensor's noise
	// 3 - Decimal who modifies the sensor standard deviation during updating stage
	// 4 - Standard deviation of noise added in prediction stage
	// 5 - Standard deviation of noise added after the resampling stage
	PF_Options(double, double, double, double, double);

	void setOptions(double, double, double, double, double);
	void operator =(PF_Options);

	void setOneOption(PF_Options_index, double);	// Write single option value
```

```
        const double getOneOption(PF_Options_index) const;// Return single option as read-
only double value
        double& updOneOption(PF_Options_index);           // Return single option as
writable double value

private:
        double SIMULATION_TIME_STEP;
        double SENSOR_STDDEV;
        double SENSOR_STDDEV_MOD;
        double MOTION_STDDEV;
        double RESAMPLE_STDDEV;
};

#endif
```

PF_Utilities.h

```
#ifndef _PF_UTILITIES_
#define _PF_UTILITIES_

#include "Simbody.h"

// Function to make a single State progress.
void advance(SimTK::State&, SimTK::TimeStepper&, const double);

// Return the exponent of a gaussian distribution function evaluated at x, given the mean
and standart deviation s:
double LogNormalProb(const double, const double, const double);

// Return the given angle, expressed in the range [0, 2*Pi). Input/Output in radians!
inline double to2Pi(double Angle) {

        bool isNegative = Angle < 0;
        double Output_Angle = fmod(Angle, 2 * SimTK::Pi);
        if (isNegative) Output_Angle += 2 * SimTK::Pi;

        return Output_Angle;
}

// Return a integer to be used as a seed in random functions. Nanoseconds from last boot
are turn into a integer. Updates every 0.001 ms.
inline int getSeed() {
        timespec ts;
        long long nanoseconds;

        clock_gettime(CLOCK_MONOTONIC, &ts);
        nanoseconds = SimTK::timespecToNs(ts);

        return static_cast<int>(nanoseconds);
}

#endif
```

Simbody_Instrument.h

```
#ifndef _SIMBODY_INSTRUMENT_
#define _SIMBODY_INSTRUMENT_

#include "Simbody.h"
#include "SimTKpf/Measuring_Instrument.h"
```

```
// Class for not-ideal measuring instruments given Simbody systems and a State:

class Simbody_Instrument : public Measuring_Instrument {
public:
	Simbody_Instrument(const SimTK::MultibodySystem&, const
SimTK::SimbodyMatterSubsystem&,
		SimTK::State&, double);

	static void setGlobalSeed(int);	// Give RNG a new seed

	const double read() override;

protected:
	const SimTK::SimbodyMatterSubsystem& m_matter;
	const SimTK::MultibodySystem& m_system;
	double m_StdDev;
	SimTK::State& rf_state;
	static SimTK::Random::Gaussian RNG;
};

#endif
```

Stopwatch.h

```
#ifndef _STOPWATCH_
#define _STOPWATCH_

#include "Simbody.h"
#include "SimTKpf/Measuring_Instrument.h"

// Mode variable to decide if measuring real time or CPU time:

enum class StopwatchMode : bool {Real_Time, CPU_Time};

// Class that actually measure time. Member reading is the time elapsed between start() and
stop() functions

class Stopwatch : public Measuring_Instrument {
public:
	Stopwatch(StopwatchMode);	// Constructor given the mode
	void start();		// Initiate or reanude the count (take a time reference)
	void restart();			// Restart the count (reading set to zero)
	void stop();		// Stop the count (update the reading)
	void measure() override;	// Just stop() implementation

private:
	double ref_time;			// Zero time reference from a start() call
	StopwatchMode mode;
};

#endif
```

assemble_Fourbar.cpp

```
#include "Simbody.h"
#include "Fourbar/assemble_Fourbar.h"
#include "Fourbar/Grashof_condition.h"
#include "SimTKpf/Particle_Filter.h"
#include "SimTKpf/PF_utilities.h"

//TODO: assembly if input bar is not crank
```

```
void assemble_Fourbar(GrashofCondition& config, SimTK::MultibodySystem& system,
SimTK::Assembler& assembler, SimTK::State& RefState, ParticleFilter& Filter) {
	double values[2];	// Assembly angle limit values
	double limits[2];	// Limits for random angle function

	if (config.get_isCrank()) {		// If input bar is crank, assembly range is 0 to
2*Pi
		limits[0] = 0;
		limits[1] = 2 * SimTK::Pi;
	}
	else {
		bool thisAssemblySuccessed = false; // True if fourbar can be assembled in
the current iteration
		bool lastAssemblySuccessed = false; // Stores previous thisAssemblySuccessed
value
		bool is0rocker = false;			// True if fourbar can be
assembled if angle is zero

		for (std::size_t n = 0, i = 0; n < 2; ++i) {
			// Try to assemble fourbar at 'i' degrees
			try {
				thisAssemblySuccessed = true;
				RefState = system.getDefaultState();
				RefState.updQ()[0] = i * SimTK::Pi / 180;
				assembler.assemble(RefState);
			}
			// Detect if assembly was not possible
			catch (const std::exception& e) {
				thisAssemblySuccessed = false;
			}

			if (i == 0) is0rocker = thisAssemblySuccessed;
			else
				if (thisAssemblySuccessed != lastAssemblySuccessed) {	//
Set values if there is a change in "assemblability"
					if (lastAssemblySuccessed)	values[n] = (i-1) *
SimTK::Pi / 180;
					else					values[n] =
i * SimTK::Pi / 180;
				++n;
				}
			lastAssemblySuccessed = thisAssemblySuccessed;
		}
		if (is0rocker) {	// Correct values before calling random function
			limits[0] = values[1] - 2 * SimTK::Pi;
			limits[1] = values[0];
		}
	}
	SimTK::Random::Uniform randomAngle(limits[0], limits[1]);
	randomAngle.setSeed(getSeed());

	// Assemble reference state
	RefState = system.getDefaultState();
	RefState.updQ()[0] = randomAngle.getValue();
	assembler.assemble(RefState);

	// Assemble particles states
	for (std::size_t i = 0; i < Filter.getParticleNumber(); ++i) {
		Filter.updParticleVec()[i].updState() = system.getDefaultState();
		Filter.updParticleVec()[i].updState().updQ()[0] = randomAngle.getValue();
		assembler.assemble(Filter.updParticleVec()[i].updState());
	}
}
```

Fourbar.cpp

```
#include "Simbody.h"
#include "SimTKpf.h"
#include "Fourbar.h"

int main(int argc, char *argv[])
{

	double SIMULATION_TIME = 60;
	double SIM_TIME_STEP = 0.006;
	double GYROSCOPE_STDDEV = 0.01;
	double UPDATING_STDDEV_MOD = 10;
	double MOTION_STDDEV = 0.02;
	double RESAMPLE_STDDEV = 0.02;
	PF_Options PARTICLE_FILTER_OPTIONS(
		SIM_TIME_STEP,
		GYROSCOPE_STDDEV,
		UPDATING_STDDEV_MOD,
		MOTION_STDDEV,
		RESAMPLE_STDDEV
	);
	std::size_t PARTICLE_NUMBER = 200;
	ParticleFilter filter(PARTICLE_NUMBER, PARTICLE_FILTER_OPTIONS);

	std::vector <double> BAR_LENGHTS = { // Examples: Double Crank: (2, 4, 3, 4), Crank-
Rocker: (4, 2, 3, 4)
		2,		// Bar1 lenght
		4,		// Bar2 lenght
		3,		// Bar3 lenght
		4 };	// Bar4 lenght
	GrashofCondition FOURBAR_CONFIGURATION = evaluateGrashof(BAR_LENGHTS);

	bool OUTPUT_IS_ENABLED = true;
	bool PAUSE_IS_ENABLED = true;
	bool TXT_WRITING_IS_ENABLED = false;

	try {
		// Handle flags
		for (int i = 1; i < argc; ++i) {
			if (strcmp("--quiet", argv[i])==0)		OUTPUT_IS_ENABLED = false;
			else if (strcmp("--nopause", argv[i])==0)  PAUSE_IS_ENABLED = false;
			else if (strcmp("--txtwrite", argv[i])==0)
								TXT_WRITING_IS_ENABLED = true;
		}

		// Create the system.
		SimTK::MultibodySystem system;
		SimTK::SimbodyMatterSubsystem matter(system);
		SimTK::GeneralForceSubsystem forces(system);
		SimTK::Force::UniformGravity gravity(forces, matter, SimTK::Vec3(0, -9.8,
0));

		SimTK::Body::Rigid Body1, Body2;

		Body1.setDefaultRigidBodyMassProperties(SimTK::MassProperties(1.0,
SimTK::Vec3(-BAR_LENGHTS[1] / 2, 0, 0), SimTK::Inertia(1)));
		SimTK::MobilizedBody::Pin Bar1(matter.Ground(),
SimTK::Transform(SimTK::Vec3(-BAR_LENGHTS[0], 0, 0)),
			Body1, SimTK::Transform(-SimTK::Vec3(BAR_LENGHTS[1], 0, 0)));

		Body2.setDefaultRigidBodyMassProperties(SimTK::MassProperties(1.0,
SimTK::Vec3(-BAR_LENGHTS[2] / 2, 0, 0), SimTK::Inertia(1)));
```

```
		SimTK::MobilizedBody::Pin Bar2(Bar1, SimTK::Transform(SimTK::Vec3(0)),
			Body2, SimTK::Transform(-SimTK::Vec3(BAR_LENGHTS[2], 0, 0)));

		SimTK::Constraint::Rod(matter.Ground(), Bar2, BAR_LENGHTS[3]);

		// Initialize the system and reference state.
		system.realizeTopology();
		SimTK::State RefState;

		// Create and add assembler.
		SimTK::Assembler assembler(system);
		assembler.setSystemConstraintsWeight(1);
		assembler.lockMobilizer(Bar1.getMobilizedBodyIndex());

		// We will random assign the reference state and particles states
		if (OUTPUT_IS_ENABLED) {
			std::cout << FOURBAR_CONFIGURATION.get_description() << "\n\nAssigning
states and assembling fourbar..." << std::endl;
		}
		assemble_Fourbar(FOURBAR_CONFIGURATION, system, assembler, RefState, filter);
		filter.setEqualWeights();

		// Simulate it.
		SimTK::RungeKuttaMersonIntegrator integ(system);
		SimTK::TimeStepper ts(system, integ);

		Gyroscope gyr(system, matter, RefState, BAR_LENGHTS[0], GYROSCOPE_STDDEV);
	// Gyroscope used by reference state
		std::vector<Gyroscope> pargyr;
					// Vector of gyroscopes used by particles

		pargyr.reserve(PARTICLE_NUMBER);

		for (std::size_t i = 0; i < PARTICLE_NUMBER; i++) {
				// Arrange gyroscope vector
			pargyr.push_back(Gyroscope(system, matter,
filter.updParticleVec()[i].updState(), BAR_LENGHTS[0], 0));
		}
		Gyroscope::setGlobalSeed(getSeed());

		SimTK::Random::Gaussian resample_noise(0, RESAMPLE_STDDEV);	// Gaussian
noise added during resampling
		resample_noise.setSeed(getSeed());

		Stopwatch Sw_total_time(StopwatchMode::CPU_Time);
		Stopwatch Sw_ref_advance(StopwatchMode::CPU_Time);
		Stopwatch Sw_part_advance(StopwatchMode::CPU_Time);
		Stopwatch Sw_resample(StopwatchMode::CPU_Time);
		Sw_total_time.start();

		for (double time = 0; time <= SIMULATION_TIME; time += SIM_TIME_STEP) { //
Loop to slowly advance simulation

			Sw_total_time.start();	// Start the total CPU time measure

			if(OUTPUT_IS_ENABLED){
			std::cout << "\n NEXT ITERATION...\n\nCurrent real time: " << time <<
" s\nCurrent CPU time: "
				<< Sw_total_time.read()  << " s" << std::endl;
			}

			if (TXT_WRITING_IS_ENABLED) {
				Angle_write(RefState, filter.updParticleList());
				Weight_write(filter.updParticleList());
				Omega_write(gyr, pargyr);
			}
```

```
                        Sw_ref_advance.start();                    // Start the reference
state advance time measure
                        advance(RefState, ts, SIM_TIME_STEP);      // Advance the reference
state
                        Sw_ref_advance.stop();                     // Stop the reference state
advance time measure

                        Sw_part_advance.start();                   // Start the particles
state advance time measure
                        filter.updateStates(ts, assembler); // Advance particles state
                        Sw_part_advance.stop();                    // Stop the particles state
advance time measure

                        gyr.measure();
                        // Update reference gyroscope reading
                        filter.updateWeights<Gyroscope>(gyr.read(), pargyr);     // Update
particles gyroscope reading and weights
                        filter.calculateESS();
                        // Calculate particles ESS

                        if(OUTPUT_IS_ENABLED){
                                printf("\nPARTICLE SUMMARY:\n");
                                for (std::size_t i = 0; i < PARTICLE_NUMBER; i++)
                                        printf("\nParticle %3.u \tOmega =%9.5f\tLog(w) =
%10.5f\tAngle = %7.3f",
                                                static_cast<unsigned int>(i + 1),
                                                static_cast<double>(pargyr[i].read()),

                static_cast<double>(filter.updParticleVec()[i].getWeight()),
                static_cast<double>(to2Pi(filter.updParticleVec()[i].getState().getQ()[0]))
                                                );
                                std::cout << "\n\nReference Gyroscope: " << gyr.read() <<
std::endl;
                                std::cout << "Reference Angle: " << to2Pi(RefState.getQ()[0])
<< std::endl;
                                std::cout << "\nReference State advance time: " <<
Sw_ref_advance.read() <<  " s" << std::endl;
                                std::cout << "Particles States advance time: " <<
Sw_part_advance.read() << " s" << std::endl;
                                printf("\n ESS = %f %%\n", static_cast<double>(filter.getESS()
* 100));
                        }

                        if (filter.getESS() < 0.5) {        // Resample when < 50 % of
effective particles

                                Sw_resample.start();  // Start the reample time measure
                                filter.resample();    // Resample
                                Sw_resample.stop();   // Stop the resample time meause

                                // Add noise to angular velocity
                                for (std::size_t i = 0; i < PARTICLE_NUMBER; i++) {
                                        double Rate =
Bar1.getRate(filter.updParticleVec()[i].updState());
                                        Bar1.setRate(filter.updParticleVec()[i].updState(),
Rate + resample_noise.getValue());
                                }
                                if (OUTPUT_IS_ENABLED) {
                                        std::cout << "\nResample done! " << "Resample time: "
<< Sw_resample.read() << std::endl;
                                }
                                Sw_resample.restart(); // Refresh the resample time measuse
                        }
```

```
                        if (PAUSE_IS_ENABLED) {
                                if (OUTPUT_IS_ENABLED) {
                                        std::cout << "\nPress Enter to keep iterating" <<
std::endl;
                                }
                                getchar();
                        }

                        Sw_ref_advance.restart();    // Refresh the reference state advance
time measure
                        Sw_part_advance.restart();   // Refresh the particles state advance
time measure
                        Sw_total_time.stop();        // Stop the total CPU time measure
                }

                if(PAUSE_IS_ENABLED){
                        if (OUTPUT_IS_ENABLED) {
                                std::cout << "\nSIMULATION ENDED. Press Enter to close this
window." << std::endl;
                        }
                        getchar();
                }
        }
        catch (const std::exception& e) {
                std::cout << "Error: " << e.what() << std::endl;
                getchar();
                return 1;
        }
}
```

Fourbar_PtNumberTest.cpp

```
#include <fstream>
#include <string>
#include "Simbody.h"
#include "SimTKpf.h"
#include "Fourbar.h"

int main() {
        const double SIMULATION_TIME = 1.020;             // 170 timesteps
        double SIM_TIME_STEP = 0.006;
        double GYROSCOPE_STDDEV = 0.01;
        double UPDATING_STDDEV_MOD = 10;
        double MOTION_STDDEV = 0.02;
        double RESAMPLE_STDDEV = 0.02;
        PF_Options PARTICLE_FILTER_OPTIONS(
                SIM_TIME_STEP,
                GYROSCOPE_STDDEV,
                UPDATING_STDDEV_MOD,
                MOTION_STDDEV,
                RESAMPLE_STDDEV
        );
        std::size_t PARTICLE_NUMBER;

        std::vector <double> BAR_LENGHTS = { // Examples: Double Crank: (2, 4, 3, 4), Crank-
Rocker: (4, 2, 3, 4)
                2,              // Bar1 lenght
                4,              // Bar2 lenght
                3,              // Bar3 lenght
                4 };    // Bar4 lenght
        GrashofCondition FOURBAR_CONFIGURATION = evaluateGrashof(BAR_LENGHTS);
```

```
try {
		// Create the system.
		SimTK::MultibodySystem system;
		SimTK::SimbodyMatterSubsystem matter(system);
		SimTK::GeneralForceSubsystem forces(system);
		SimTK::Force::UniformGravity gravity(forces, matter, SimTK::Vec3(0, -9.8,
0));

		SimTK::Body::Rigid Body1, Body2;

		Body1.setDefaultRigidBodyMassProperties(SimTK::MassProperties(1.0,
SimTK::Vec3(-BAR_LENGHTS[1] / 2, 0, 0), SimTK::Inertia(1)));
		SimTK::MobilizedBody::Pin Bar1(matter.Ground(),
SimTK::Transform(SimTK::Vec3(-BAR_LENGHTS[0], 0, 0)),
			Body1, SimTK::Transform(-SimTK::Vec3(BAR_LENGHTS[1], 0, 0)));

		Body2.setDefaultRigidBodyMassProperties(SimTK::MassProperties(1.0,
SimTK::Vec3(-BAR_LENGHTS[2] / 2, 0, 0), SimTK::Inertia(1)));
		SimTK::MobilizedBody::Pin Bar2(Bar1, SimTK::Transform(SimTK::Vec3(0)),
			Body2, SimTK::Transform(-SimTK::Vec3(BAR_LENGHTS[2], 0, 0)));

		SimTK::Constraint::Rod(matter.Ground(), Bar2, BAR_LENGHTS[3]);

		// Initialize the system and reference state.
		system.realizeTopology();
		SimTK::State RefState;

		// Create and add assembler.
		SimTK::Assembler assembler(system);
		assembler.setSystemConstraintsWeight(1);

		// Define test variables
		std::fstream File;
		SimTK::String filename = "Fourbar_PtNumberTest.txt";                    //
This file will contain results when test finishes
		SimTK::String savename = "Fourbar_PtNumberTest_Savestate.txt";  // This file
will contain variables data in case test get interrupted
		std::array<std::size_t, 10> pt_numbers = { // PARTICLE_NUMBER values to test
			10,           20,           50,
			100,   200,   500,
			1000,  2000,  5000,
			10000
		};
		std::size_t tries = 100;     // Number of convergence tries in a simulation
		int convs = 0;                      // Number of convergences in a simulation
		double CPUtimestart;        // Reference to check CPU time
		double CPUtotaltime = 0;

		// Check if test is in progress
		int saved_try = 0;

		File.open(savename, std::ios::in);
		if (File) {
			SimTK::String fileline;

			std::getline(File, fileline);
			std::stringstream(fileline) >> saved_try;

			std::getline(File, fileline);
			std::stringstream(fileline) >> convs;

			std::getline(File, fileline);
			std::stringstream(fileline) >> CPUtotaltime;
		}
		CPUtimestart = SimTK::cpuTime();
```

```
            std::cout << "Press Q to quit test\n" << std::endl;

            for (std::size_t ptnumber_i = static_cast<int>(floor(saved_try / tries));
ptnumber_i < pt_numbers.size(); ++ptnumber_i) {

                    // We will define the particle number and ParticleFilter variable
                    PARTICLE_NUMBER = pt_numbers[ptnumber_i];
                    ParticleFilter filter(PARTICLE_NUMBER, PARTICLE_FILTER_OPTIONS);

                    std::cout << "Starting test with PARTICLE_NUMBER = " <<
PARTICLE_NUMBER << std::endl;

                    for (std::size_t try_n = static_cast<int>(fmod(saved_try, tries));
try_n < tries; ++try_n) {

                            // We will random assign the reference state and particle
states
                            assemble_Fourbar(FOURBAR_CONFIGURATION, system, assembler,
RefState, filter);
                            filter.setEqualWeights();

                            // Simulate it.
                            SimTK::RungeKuttaMersonIntegrator integ(system);
                            SimTK::TimeStepper ts(system, integ);

                            Gyroscope gyr(system, matter, RefState, BAR_LENGHTS[0],
GYROSCOPE_STDDEV);    // Gyroscope used by reference state
                            std::vector<Gyroscope> pargyr;
                                                    // Vector of gyroscopes used by
particles

                            pargyr.reserve(PARTICLE_NUMBER);

                            for (std::size_t i = 0; i < PARTICLE_NUMBER; i++) {
                                    // Arrange gyroscope vector
                                    pargyr.push_back(Gyroscope(system, matter,
filter.updParticleVec()[i].updState(), BAR_LENGHTS[0], 0));
                            }
                            Gyroscope::setGlobalSeed(getSeed());

                            SimTK::Random::Gaussian resample_noise(0, RESAMPLE_STDDEV);
        // Gaussian noise added during resampling
                            resample_noise.setSeed(getSeed());

                            CPUtimestart = SimTK::cpuTime();

                            for (double time = 0; time <= SIMULATION_TIME; time +=
SIM_TIME_STEP) {      // Loop to slowly advance simulation

                                    advance(RefState, ts, SIM_TIME_STEP);
                    // Advance reference state
                                    filter.updateStates(ts, assembler);
                    // Advance particles state
                                    gyr.measure();
                                    // Update reference gyroscope reading
                                    filter.updateWeights<Gyroscope>(gyr.read(), pargyr);
        // Update particles gyroscope reading and weights
                                    filter.calculateESS();
                                    // Calculate particles ESS

                                    if (filter.getESS() < 0.5) {        // Resample when <
50 % of effective particles
                                            filter.resample();
                                            for (std::size_t i = 0; i < PARTICLE_NUMBER;
i++) { // Noise added to angular velocity
```

```
                                                double Rate =
Bar1.getRate(filter.updParticleVec()[i].updState());

       Bar1.setRate(filter.updParticleVec()[i].updState(), Rate +
resample_noise.getValue());
                                                }
                                        }
                                }       // Here ends iterations loop

                                // Convergence criteria:
                                double Ref_Angle, Estimated_Angle = 0;

                                Ref_Angle = to2Pi(RefState.getQ()[0]); // Get reference State
wanted angle

                                filter.normalizeLinearWeights();

                                for (std::size_t i = 0; i < PARTICLE_NUMBER; i++) {

                                        double partAngle =
to2Pi(filter.updParticleVec()[i].getState().getQ()[0]);  // Get particle State wanted angle
                                        double partLinW =
exp(filter.updParticleVec()[i].getWeight());             // Get particle linear weight

                                        Estimated_Angle += partAngle * partLinW;
                                }

                                if (abs(Ref_Angle - Estimated_Angle) <= 5 * SimTK::Pi / 180 ||
                                abs(Ref_Angle - Estimated_Angle) >= 355 * SimTK::Pi / 180) {
                                        ++convs;
                                }

                                CPUtotaltime += SimTK::cpuTime() - CPUtimestart;
                                saved_try++;

                                File.open(savename, std::ios::out);
                                File << saved_try << std::endl;
                                File << convs << std::endl;
                                File << CPUtotaltime << std::endl;
                                File.close();

                                CPUtimestart = SimTK::cpuTime();

                        }       // Here ends tries loop

                        std::cout << "\tAchieved test in " << CPUtotaltime << " s.\n" <<
std::endl;

                        File.open(filename, std::ios::app);
                        File << "PtNumber: " << PARTICLE_NUMBER << "\t " << "\tTest time: " <<
CPUtotaltime << " s"
                                << "\t" << convs << "/" << tries << " success" << std::endl;
                        File.close();

                        convs = 0;
                        CPUtotaltime = 0;
                        CPUtimestart = SimTK::cpuTime();

                }       // Here ends Particle Number values loop

                File.open(savename, std::ios::out);
                File << saved_try << std::endl;
                File << convs << std::endl;
                File << CPUtotaltime << std::endl;
                File.close();
```

```
		std::cout << "\nSIMULATION ENDED. Press Enter to close this window." <<
std::endl;
		getchar();
		return 0;

	}	// Here ends all tests

	catch (const std::exception& e) {
		std::cout << "\nError: " << e.what() << std::endl;
	}
}
```

Fourbar_StdDevTest.cpp

```
#include <fstream>
#include <string>
#include "Simbody.h"
#include "SimTKpf.h"
#include "Fourbar.h"

int main() {
	const double SIMULATION_TIME = 1.020;		// 170 timesteps
	double SIM_TIME_STEP = 0.006;
	double GYROSCOPE_STDDEV = 0.01;
	double UPDATING_STDDEV_MOD = 10;
	double MOTION_STDDEV = 0;
	double RESAMPLE_STDDEV = 0.02;
	PF_Options PARTICLE_FILTER_OPTIONS(
		SIM_TIME_STEP,
		GYROSCOPE_STDDEV,
		UPDATING_STDDEV_MOD,
		MOTION_STDDEV,
		RESAMPLE_STDDEV
	);
	std::size_t PARTICLE_NUMBER = 200;
	ParticleFilter filter(PARTICLE_NUMBER, PARTICLE_FILTER_OPTIONS);

	std::vector <double> BAR_LENGHTS = { // Examples: Double Crank: (2, 4, 3, 4),
Crank-Rocker: (4, 2, 3, 4)
		2,		// Bar1 lenght
		4,		// Bar2 lenght
		3,		// Bar3 lenght
		4 };	// Bar4 lenght
	GrashofCondition FOURBAR_CONFIGURATION = evaluateGrashof(BAR_LENGHTS);

	try {
		// Create the system.
		SimTK::MultibodySystem system;
		SimTK::SimbodyMatterSubsystem matter(system);
		SimTK::GeneralForceSubsystem forces(system);
		SimTK::Force::UniformGravity gravity(forces, matter, SimTK::Vec3(0, -9.8,
0));

		SimTK::Body::Rigid Body1, Body2;

		Body1.setDefaultRigidBodyMassProperties(SimTK::MassProperties(1.0,
SimTK::Vec3(-BAR_LENGHTS[1] / 2, 0, 0), SimTK::Inertia(1)));
		SimTK::MobilizedBody::Pin Bar1(matter.Ground(),
SimTK::Transform(SimTK::Vec3(-BAR_LENGHTS[0], 0, 0)),
			Body1, SimTK::Transform(-SimTK::Vec3(BAR_LENGHTS[1], 0, 0)));
```

```
            Body2.setDefaultRigidBodyMassProperties(SimTK::MassProperties(1.0,
SimTK::Vec3(-BAR_LENGHTS[2] / 2, 0, 0), SimTK::Inertia(1)));
            SimTK::MobilizedBody::Pin Bar2(Bar1, SimTK::Transform(SimTK::Vec3(0)),
                  Body2, SimTK::Transform(-SimTK::Vec3(BAR_LENGHTS[2], 0, 0)));

            SimTK::Constraint::Rod(matter.Ground(), Bar2, BAR_LENGHTS[3]);

            // Initialize the system, reference state and particles.
            system.realizeTopology();
            SimTK::State RefState;
            ParticleList particles(PARTICLE_NUMBER);

            // Create and add assembler.
            SimTK::Assembler assembler(system);
            assembler.setSystemConstraintsWeight(1);

            // Define test variables
            std::fstream File;
            SimTK::String filename = "Fourbar_StdDevTest.txt";
      // This file will contain results when test finishes
            SimTK::String savename = "Fourbar_StdDevTest_Savestate.txt"; // This file
will contain variables data in case test get interrupted
            std::array<double, 10> deviations = {    // MOTION_STDDEV values to test
                  0.001, 0.002, 0.005,
                  0.01,  0.02,  0.05,
                  0.1,   0.2,   0.5,
                  1
            };
            std::size_t tries = 100;    // Number of convergence tries in a simulation
            int convs = 0;                          // Number of convergences in a
simulation
            double CPUtimestart;       // Reference to check CPU time
            double CPUtotaltime = 0;

            // Check if test is in progress
            int saved_try= 0;

            File.open(savename, std::ios::in);
            if (File) {
                  SimTK::String fileline;

                  std::getline(File, fileline);
                  std::stringstream(fileline) >> saved_try;

                  std::getline(File, fileline);
                  std::stringstream(fileline) >> convs;

                  std::getline(File, fileline);
                  std::stringstream(fileline) >> CPUtotaltime;
            }
            CPUtimestart = SimTK::cpuTime();

            std::cout << "Press Q to quit test\n" << std::endl;

            for (std::size_t StdDev_i = static_cast<int>(floor(saved_try/tries));
StdDev_i < deviations.size(); ++StdDev_i) {

                  MOTION_STDDEV = deviations[StdDev_i];
                  filter.setOneOption(PF_Options_index::MOTION_STDDEV,
MOTION_STDDEV);
```

```
                    std::cout << "Starting test with MOTION_STDDEV = " <<
MOTION_STDDEV << std::endl;

                    for (std::size_t try_n = static_cast<int>(fmod(saved_try, tries));
try_n < tries; ++try_n) {

                            // We will random assign the reference state and particle
states
                            assemble_Fourbar(FOURBAR_CONFIGURATION, system, assembler,
RefState, filter);
                            filter.setEqualWeights();

                            // Simulate it.
                            SimTK::RungeKuttaMersonIntegrator integ(system);
                            SimTK::TimeStepper ts(system, integ);

                            Gyroscope gyr(system, matter, RefState, BAR_LENGHTS[0],
GYROSCOPE_STDDEV);   // Gyroscope used by reference state
                            std::vector<Gyroscope> pargyr;
                                                    // Vector of gyroscopes used by
particles

                            pargyr.reserve(PARTICLE_NUMBER);

                            for (std::size_t i = 0; i < PARTICLE_NUMBER; i++) {
                                    // Arrange gyroscope vector
                                    pargyr.push_back(Gyroscope(system, matter,
filter.updParticleVec()[i].updState(), BAR_LENGHTS[0], 0));
                            }
                            Gyroscope::setGlobalSeed(getSeed());

                            SimTK::Random::Gaussian resample_noise(0, RESAMPLE_STDDEV);
      // Gaussian noise added during resampling
                            resample_noise.setSeed(getSeed());

                            CPUtimestart = SimTK::cpuTime();

                            for (double time = 0; time <= SIMULATION_TIME; time +=
SIM_TIME_STEP) {     // Loop to slowly advance simulation

                                    advance(RefState, ts, SIM_TIME_STEP);
                    // Advance reference state
                                    filter.updateStates(ts, assembler);
                            // Advance particles state
                                    gyr.measure();
                                            // Update reference gyroscope reading
                                    filter.updateWeights<Gyroscope>(gyr.read(), pargyr);
      // Update particles gyroscope reading and weights
                                    filter.calculateESS();
                                    // Calculate particles ESS

                                    if (filter.getESS() < 0.5) {            // Resample
when < 50 % of effective particles
                                            filter.resample();
                                            for (std::size_t i = 0; i < PARTICLE_NUMBER;
i++) { // Noise added to angular velocity
                                                    double Rate =
Bar1.getRate(filter.updParticleVec()[i].updState());

      Bar1.setRate(filter.updParticleVec()[i].updState(), Rate +
resample_noise.getValue());
                                            }
```

```
                                }
                        }       // Here ends one iteration

                        // Convergence criteria:
                        double Ref_Angle, Estimated_Angle = 0;

                        Ref_Angle = to2Pi(RefState.getQ()[0]); // Get reference
State wanted angle

                        filter.normalizeLinearWeights();

                        for (std::size_t i = 0; i < PARTICLE_NUMBER; i++) {

                                double partAngle =
to2Pi(filter.updParticleVec()[i].getState().getQ()[0]);      // Get particle State
wanted angle
                                double partLinW =
exp(filter.updParticleVec()[i].getWeight());                        // Get particle
linear weight

                                Estimated_Angle += partAngle * partLinW;
                        }
                        if (abs(Ref_Angle - Estimated_Angle) <= 5 * SimTK::Pi / 180
||
                                abs(Ref_Angle - Estimated_Angle) >= 355 * SimTK::Pi
/ 180) {
                                ++convs;
                        }

                        CPUtotaltime += SimTK::cpuTime() - CPUtimestart;
                        saved_try++;

                        File.open(savename, std::ios::out);
                        File << saved_try << std::endl;
                        File << convs << std::endl;
                        File << CPUtotaltime << std::endl;
                        File.close();

                        CPUtimestart = SimTK::cpuTime();

                }       //Here ends tries loop

                std::cout << "\tAchieved test in " << CPUtotaltime << " s.\n" <<
std::endl;

                File.open(filename, std::ios::app);
                File << "StdDev: " << MOTION_STDDEV << "\t " << "\tTest time: " <<
CPUtotaltime << " s"
                        << "\t" << convs << "/" << tries << " success" <<
std::endl;
                File.close();

                convs = 0;
                CPUtotaltime = 0;
                CPUtimestart = SimTK::cpuTime();

        }       // Here ends StdDev values loop

        File.open(savename, std::ios::out);
        File << saved_try << std::endl;
        File << convs << std::endl;
        File << CPUtotaltime << std::endl;
```

```
		File.close();

		std::cout << "\nSIMULATION ENDED. Press Enter to close this window." <<
std::endl;
		getchar();
		return 0;

	}	// Here ends all tests

	catch (const std::exception& e) {
		std::cout << "\nError: " << e.what() << std::endl;
	}
}
```

Grashof_condition.cpp

```
#include <string>
#include <vector>
#include "Fourbar/Grashof_condition.h"

// GrashofCondition Implementation
GrashofCondition::GrashofCondition(std::string description, bool isCrank) :
description(description), isCrank(isCrank) {}
std::string GrashofCondition::get_description() { return description;   }
bool GrashofCondition::get_isCrank()                    { return isCrank;           }

// evaluateGrashof function Implementation
GrashofCondition evaluateGrashof(std::vector<double>& L) {
	const double T1 = L[0] + L[2] - L[3] - L[1];
	const double T2 = L[0] + L[3] - L[2] - L[1];
	const double T3 = L[3] + L[2] - L[0] - L[1];

	const bool T1pos = (T1 > 0 ? 1 : 0);
	const bool T2pos = (T2 > 0 ? 1 : 0);
	const bool T3pos = (T3 > 0 ? 1 : 0);

	GrashofCondition AllConditions[9]{
		{ "Fourbar is a 0-Rocker-0-Rocker.",		false	},
		{ "Fourbar is a Crank-Crank.",				true	},
		{ "Fourbar is a Rocker-Rocker.",		false	},
		{ "Fourbar is a Pi-Rocker-Pi-Rocker.",		false	},
		{ "Fourbar is a Rocker-Crank.",				false	},
		{ "Fourbar is a Pi-Rocker-0-Rocker.",		false	},
		{ "Fourbar is a 0-Rocker-Pi-Rocker.",		false	},
		{ "Fourbar is a Crank-Rocker.",				true	},
		{ "Fourbar is a Crank-Crank and it folds.",	true	}
	};

	std::size_t chosenCondition;

	// Grashof condition evaluation according to terms T1, T2 & T3.
	if (T1*T2*T3 != 0) chosenCondition = T1pos * 4 + T2pos * 2 + T3pos;
	else chosenCondition = 8;

	return AllConditions[chosenCondition];
}
```

Gyroscope.cpp

```
#include "Simbody.h"
#include "SimTKpf/Simbody_Instrument.h"
#include "Fourbar/Gyroscope.h"

Gyroscope::Gyroscope(const SimTK::MultibodySystem& system, const
SimTK::SimbodyMatterSubsystem& matter, SimTK::State& RefState,
	double GroundBarLenght, double StdDev = 0) : Simbody_Instrument(system, matter,
RefState, StdDev),
	m_GroundBarLenght(GroundBarLenght) {}

void Gyroscope::measure(){
	using namespace SimTK;
	m_system.realize(rf_state, Stage::Velocity);	// We can get now cartesian
coordinates and velocities

	// Get first bar mobilized body
	MobilizedBodyIndex index = MobilizedBodyIndex(1);
	MobilizedBody body = m_matter.getMobilizedBody(index);

	// Get positions
	Vec3 Pos2 = body.getBodyOriginLocation(rf_state);
	Vec3 Pos1 = Vec3(-m_GroundBarLenght, 0, 0);

	// Get velocities
	Vec3 Vel2 = body.getBodyOriginVelocity(rf_state);
	Vec3 Vel1 = Vec3(0, 0, 0);

	Vec3 r = Pos2 - Pos1;
	Real distance = r.norm();
	Vec3 w = -((Vel2 - Vel1) % r) / (distance * distance);
	Vec3 u = Vec3(0, 0, 1);

	reading = ~w * u;	// Dot product
}
```

Txt_write.cpp

```
#include <fstream>
#include "Simbody.h"
#include "SimTKpf/PF_utilities.h"
#include "SimTKpf/Particle_Classes.h"
#include "SimTKcommon/internal/Subsystem.h"
#include "Fourbar/Gyroscope.h"
#include "Fourbar/Txt_write.h"

void Angle_write(SimTK::State& ref_State, ParticleList& particles) {

	std::size_t particle_number = particles.getAllParticles().size();
	std::fstream File;
	SimTK::String filename = "Angles.txt";
	char AngleValue[9];

	File.open(filename, std::ios::app);
	sprintf(AngleValue, "%.4f", to2Pi(ref_State.getQ()[0]));
	File << AngleValue;

	for (std::size_t i = 0; i < particle_number; ++i) {
		sprintf(AngleValue, "\t%.4f", to2Pi(particles[i].getState().getQ()[0]));
		File << AngleValue;
```

```
	}

	File << std::endl;
	File.close();
}

void Weight_write(ParticleList& particles) {

	std::size_t particle_number = particles.getAllParticles().size();
	std::fstream File;
	SimTK::String filename = "Weights.txt";
	char WeightValue[9];

	particles.normalizeLinearWeights();

	File.open(filename, std::ios::app);

	for (std::size_t i = 0; i < particle_number; ++i) {
		sprintf(WeightValue, "\t%.6f", exp(particles[i].getWeight()));
		File << WeightValue;
	}

	File << std::endl;
	File.close();
}

void Omega_write(Gyroscope& RefGyr, std::vector<Gyroscope>& PtGyr) {

	std::size_t particle_number = PtGyr.size();
	std::fstream File;
	SimTK::String filename = "Omegas.txt";
	char OmegaValue[8];

	File.open(filename, std::ios::app);
	RefGyr.measure();
	sprintf(OmegaValue, "%.3f", RefGyr.read());
	File << OmegaValue;

	for (std::size_t i = 0; i < particle_number; ++i) {
		PtGyr[i].measure();
		sprintf(OmegaValue, "\t%.3f", PtGyr[i].read());
		File << OmegaValue;
	}

	File << std::endl;
	File.close();
}
```

Measuring_Instrument.cpp

```
#include "SimTKpf/Measuring_Instrument.h"

const double Measuring_Instrument::read(){
	return reading;
}
```

Particle_Classes.cpp

```
#include "Simbody.h"
#include "SimTKpf/Particle_Classes.h"

// ParticleDynState Implementation

ParticleDynState::ParticleDynState() : logw(0) {}

ParticleDynState::ParticleDynState(const SimTK::State& stateset, double weight = 0)
	: state(stateset), logw(weight) {}

void ParticleDynState::operator =(const ParticleDynState& p2) {
	logw = p2.getWeight();
	state = p2.getState();
}

bool ParticleDynState::operator <(const ParticleDynState& p2) {
	return logw < p2.getWeight();
}

void ParticleDynState::setWeight(const double& weight)		{ logw = weight;	}
const double ParticleDynState::getWeight() const		{ return logw;		}
double& ParticleDynState::updWeight()				{ return logw;		}

void ParticleDynState::setState(const SimTK::State& stateset)	{ state = stateset;	}
const SimTK::State& ParticleDynState::getState() const		{ return state;		}
SimTK::State& ParticleDynState::updState()			{ return state;
	}

void ParticleDynState::setStateDefault(const SimTK::MultibodySystem& system) {
	state = system.getDefaultState();
}


// ParticleList Implementation

ParticleList::ParticleList() : ESS(0) {}

ParticleList::ParticleList(std::size_t particle_number) {
	particles.resize(particle_number);
	ESS = 0;
}
ParticleList::ParticleList(std::vector <ParticleDynState>& ParticleVector) {
	particles = ParticleVector;
	ESS = 0;
}

const ParticleDynState ParticleList::getParticle(std::size_t n) const  {
return particles[n]; }
const ParticleDynState ParticleList::operator () (std::size_t n) const  {
return particles[n]; }

ParticleDynState& ParticleList::updParticle(std::size_t n)	{ return particles[n]; }
ParticleDynState& ParticleList::operator [] (std::size_t n)	{ return particles[n]; }

void ParticleList::addParticle(const ParticleDynState& pt) {
	particles.emplace_back(pt);
}
void ParticleList::operator << (const ParticleDynState& pt) {
	particles.emplace_back(pt);
}
```

```
void ParticleList::deleteParticle() {
	particles.pop_back();
}
void ParticleList::deleteParticle(std::size_t index) {
	particles.erase(particles.begin() + index);
}
void ParticleList::deleteParticle(std::size_t first_index, std::size_t last_index) {
	particles.erase(particles.begin() + first_index, particles.begin() + last_index);
}

const std::vector <ParticleDynState>& ParticleList::getAllParticles() const {
	return particles;
}
std::vector <ParticleDynState>& ParticleList::updAllParticles() {
	return particles;
}
std::size_t ParticleList::size() const {
	return particles.size();
}

void ParticleList::advanceStates(SimTK::TimeStepper& ts, const double dt) {
	for (auto it = particles.begin(); it != particles.end(); ++it) {
		ts.initialize(it->getState());
		ts.stepTo(ts.getTime() + dt);
		it->setState(ts.getState());
	}
}

void ParticleList::setEqualWeights() {
	for (auto it = particles.begin(); it != particles.end(); ++it)
		it->setWeight(0);
}
void ParticleList::setEqualLinearWeights() {
	const double newWeight = std::log(1. / particles.size());

	for (auto it = particles.begin(); it != particles.end(); ++it)
		it->setWeight(newWeight);
}

void ParticleList::normalizeWeights() {
	SimTK::Real maxLogW = particles[0].getWeight();

	for (auto it = particles.begin(); it != particles.end(); ++it)
		maxLogW = std::max<SimTK::Real>(maxLogW, it->getWeight());	// Find the
maximum logarithmic weight

	for (auto it = particles.begin(); it != particles.end(); ++it)  // Now normalize
weights
		it->updWeight() -= maxLogW;
}
void ParticleList::normalizeLinearWeights() {
	double sumLinearWs = 0;

	for (auto it = particles.begin(); it != particles.end(); ++it)
		sumLinearWs += std::exp(it->getWeight());

	for (auto it = particles.begin(); it != particles.end(); ++it)
		it->updWeight() -= std::log(sumLinearWs);
}

void ParticleList::calculateESS() {
	double sumLinearWeights = 0, LinearW, accum = 0;

	for (auto it = particles.begin(); it != particles.end(); it++) {
		LinearW = std::exp(it->getWeight());
		sumLinearWeights += LinearW;
```

```
                accum += LinearW * LinearW;
        }

        if (accum == 0)       ESS = 0;       // ESS == 0 when all linear weights are zero
        else                  ESS = (sumLinearWeights*sumLinearWeights) / (particles.size() *
accum);// ESS == 1 when equal weights
}

const double ParticleList::getESS() const { return ESS; }

void ParticleList::resample() {

        std::size_t i, j, particle_number = particles.size();
        std::vector <ParticleDynState> ResamplingPtcls;
        std::vector <double> LinearWeights, accumLinearWs, equalWeights;
        double sumLinearWeights = 0, value = 0, maxLogW = 0;

        // Get maximum logarithmic weight:

        for (auto it = particles.begin(); it != particles.end(); ++it)
                if (maxLogW < it->getWeight()) maxLogW = it->getWeight();

        // Get particles linear weights, calculate their sum and normalize them:

        for (auto it = particles.begin(); it != particles.end(); ++it) {
                value = std::exp(it->getWeight() - maxLogW);      // value here contains
linear weight
                LinearWeights.push_back(value);
                sumLinearWeights += value;
        }

        for (i = 0; i < particle_number; ++i)
                LinearWeights[i] /= sumLinearWeights;

        // Arrange a vector with accumulative linear weights:

        value = 0;
        for (auto it = LinearWeights.begin(); it != LinearWeights.end(); ++it) {
                value += *it;         // value here contains accumulative normalized linear
weights
                accumLinearWs.push_back(value);
        }
        accumLinearWs[particle_number - 1] = 1.1;

        // Arrange a vector with accumulative equal weights:

        i = 1;
        value = 0.99999 / (particle_number - 1);   // value here contains equal weights step
        for (auto it = LinearWeights.begin(); it != LinearWeights.end() - 1; ++it, ++i) {
                equalWeights.push_back(value*i);
        }
        equalWeights.push_back(1.0);



        // Select particles to later replacement:

        i = 0, j = 0;
        while (i < particle_number) {
                if (equalWeights[i] <= accumLinearWs[j]) {
                        ResamplingPtcls.emplace_back(particles[j]);
                        i++;
                }
                else
                        j++;
        }
```

```
        // Replace particles:
        particles = ResamplingPtcls;

        // Make all particles weights equal
        setEqualWeights();
}
```

Particle_Filter.cpp

```
#include "Simbody.h"
#include "SimTKpf/PF_Options.h"
#include "SimTKpf/Particle_Classes.h"
#include "SimTKpf/Particle_Filter.h"
#include "SimTKpf/PF_utilities.h"
#include "SimTKpf/Simbody_Instrument.h"

// ParticleFilter Implementation

ParticleFilter::ParticleFilter(std::size_t particle_number, PF_Options& Options) :
        Filter_Options(Options),
        Particles(ParticleList(particle_number)) {}

void ParticleFilter::setOptions(PF_Options newOptions)   { Filter_Options = newOptions;
        }
const PF_Options ParticleFilter::getOptions() const             { return Filter_Options;
                }
PF_Options& ParticleFilter::updOptions()                        { return Filter_Options;
                }

void ParticleFilter::setOneOption(PF_Options_index index, double newValue)     {
Filter_Options.setOneOption(index, newValue); }
const double ParticleFilter::getOneOption(PF_Options_index index) const        { return
Filter_Options.getOneOption(index); }
double& ParticleFilter::updOneOption(PF_Options_index index)                            {
return Filter_Options.updOneOption(index); }

const ParticleList ParticleFilter::getParticleList() const      { return Particles; }
ParticleList& ParticleFilter::updParticleList()                         { return Particles;
}

const std::vector<ParticleDynState> ParticleFilter::getParticleVec() const     { return
Particles.getAllParticles(); }
std::vector<ParticleDynState>& ParticleFilter::updParticleVec()                         {
return Particles.updAllParticles(); }

std::size_t ParticleFilter::getParticleNumber() const    { return Particles.size();
        }

void ParticleFilter::setEqualWeights()                   { Particles.setEqualWeights();
                        }
void ParticleFilter::setEqualLinearWeights()      { Particles.setEqualLinearWeights();
        }

void ParticleFilter::normalizeWeights()                  { Particles.normalizeWeights();
                        }
void ParticleFilter::normalizeLinearWeights()     { Particles.normalizeLinearWeights();
        }

void ParticleFilter::calculateESS()        { Particles.calculateESS();         }
double ParticleFilter::getESS() const      { return Particles.getESS(); }
```

```
void ParticleFilter::updateStates(SimTK::TimeStepper& ts, SimTK::Assembler& assembler) {

	double const timestep = getOneOption(PF_Options_index::SIMULATION_TIME_STEP);
	double const Motion_StdDev = getOneOption(PF_Options_index::MOTION_STDDEV);
	SimTK::Random::Gaussian motion_noise (0, Motion_StdDev);
	motion_noise.setSeed(getSeed());

	Particles.advanceStates(ts, timestep);	// Advance particles state

	for (std::size_t i = 0; i < Particles.size(); i++) {	// Add some noise after
advancing particles states
		Particles[i].updState().updQ()[0] += motion_noise.getValue();
		assembler.assemble(Particles[i].updState());
	}
}

void ParticleFilter::resample() {
	Particles.resample();
}
```

PF_Options.cpp

```
#include <exception>
#include "SimTKpf/PF_Options.h"

// PF_Options Implementation

PF_Options::PF_Options() {};
PF_Options::PF_Options(
	double PF_SIMULATION_TIME_STEP,		// Time step between two filter
iterations
	double PF_SENSOR_STDDEV,		// Standard deviation of Ground Truth
sensor's noise
	double PF_SENSOR_STDDEV_MOD,	// Decimal who modifies the sensor standard
deviation during updating stage
	double PF_MOTION_STDDEV,		// Standard deviation of noise added in
prediction stage
	double PF_RESAMPLE_STDDEV		// Standard deviation of noise added
after the resampling stage
	) :	SIMULATION_TIME_STEP(PF_SIMULATION_TIME_STEP),
		SENSOR_STDDEV(PF_SENSOR_STDDEV),
		SENSOR_STDDEV_MOD(PF_SENSOR_STDDEV_MOD),
		MOTION_STDDEV(PF_MOTION_STDDEV),
		RESAMPLE_STDDEV(PF_RESAMPLE_STDDEV) {}

void PF_Options::setOptions(double PF_SIMULATION_TIME_STEP,double PF_SENSOR_STDDEV, double
PF_SENSOR_STDDEV_MOD,
	double PF_MOTION_STDDEV, double PF_RESAMPLE_STDDEV){
		SIMULATION_TIME_STEP = PF_SIMULATION_TIME_STEP;
		SENSOR_STDDEV = PF_SENSOR_STDDEV;
		SENSOR_STDDEV_MOD = PF_SENSOR_STDDEV_MOD;
		MOTION_STDDEV = PF_MOTION_STDDEV;
		RESAMPLE_STDDEV = PF_RESAMPLE_STDDEV;
}

void PF_Options::operator =(PF_Options PF_Options2) {
	SIMULATION_TIME_STEP = PF_Options2.SIMULATION_TIME_STEP;
	SENSOR_STDDEV = PF_Options2.SENSOR_STDDEV;
	SENSOR_STDDEV_MOD = PF_Options2.SENSOR_STDDEV_MOD;
	MOTION_STDDEV = PF_Options2.MOTION_STDDEV;
	RESAMPLE_STDDEV = PF_Options2.RESAMPLE_STDDEV;
}
```

```
void PF_Options::setOneOption(PF_Options_index index, double newValue) {
	switch (index) {
		case PF_Options_index::SIMULATION_TIME_STEP:	SIMULATION_TIME_STEP =
newValue;		break;
		case PF_Options_index::SENSOR_STDDEV:			SENSOR_STDDEV
		= newValue;		break;
		case PF_Options_index::SENSOR_STDDEV_MOD:		SENSOR_STDDEV_MOD
	= newValue;		break;
		case PF_Options_index::MOTION_STDDEV:			MOTION_STDDEV
		= newValue;		break;
		case PF_Options_index::RESAMPLE_STDDEV:			RESAMPLE_STDDEV
			= newValue;		break;
		default: throw PF_Options_exception();
	}
}

const double PF_Options::getOneOption(PF_Options_index index) const {
	switch (index) {
		case PF_Options_index::SIMULATION_TIME_STEP:	return
SIMULATION_TIME_STEP; break;
		case PF_Options_index::SENSOR_STDDEV:			return
SENSOR_STDDEV;		break;
		case PF_Options_index::SENSOR_STDDEV_MOD:		return SENSOR_STDDEV_MOD;
		break;
		case PF_Options_index::MOTION_STDDEV:			return
MOTION_STDDEV;		break;
		case PF_Options_index::RESAMPLE_STDDEV:			return
RESAMPLE_STDDEV;		break;
		default: throw PF_Options_exception();
	}
}

double& PF_Options::updOneOption(PF_Options_index index) {
	switch (index) {
		case PF_Options_index::SIMULATION_TIME_STEP:	return
SIMULATION_TIME_STEP; break;
		case PF_Options_index::SENSOR_STDDEV:			return
SENSOR_STDDEV;		break;
		case PF_Options_index::SENSOR_STDDEV_MOD:		return SENSOR_STDDEV_MOD;
		break;
		case PF_Options_index::MOTION_STDDEV:			return
MOTION_STDDEV;		break;
		case PF_Options_index::RESAMPLE_STDDEV:			return
RESAMPLE_STDDEV;		break;
		default: throw PF_Options_exception();
	}
}
```

PF_utilities.cpp

```
#include "Simbody.h"
#include "SimTKpf/PF_utilities.h"

void advance(SimTK::State& state, SimTK::TimeStepper& ts, const double dt) {
	ts.initialize(state);
	ts.stepTo(ts.getTime() + dt);
	state = ts.getState();
}

double LogNormalProb(const double x, const double mean, const double s) {
	double y = ((mean - x) / s);
	return -0.5*y*y;
}
```

Simbody_Instrument.cpp

```
#include "Simbody.h"
#include "SimTKpf/Measuring_Instrument.h"
#include "SimTKpf/Simbody_Instrument.h"

Simbody_Instrument::Simbody_Instrument(const SimTK::MultibodySystem& system, const
SimTK::SimbodyMatterSubsystem& matter,
              SimTK::State& RefState, double StdDev = 0) : m_system(system),
m_matter(matter), rf_state(RefState), m_StdDev(StdDev) {}

void Simbody_Instrument::setGlobalSeed(int seed) { RNG.setSeed(seed); }

const double Simbody_Instrument::read(){
       RNG.setStdDev(m_StdDev);
       return reading + RNG.getValue();
}

SimTK::Random::Gaussian Simbody_Instrument::RNG;
```

Stopwatch.cpp

```
#include "Simbody.h"
#include "SimTKpf/Measuring_Instrument.h"
#include "SimTKpf/Stopwatch.h"

// Stopwatch Implementation

Stopwatch::Stopwatch(StopwatchMode SWmode) : mode(SWmode) { reading = 0; }

void Stopwatch::start() {
       if (mode == StopwatchMode::Real_Time)            ref_time = SimTK::realTime();
       else if (mode == StopwatchMode::CPU_Time)  ref_time = SimTK::cpuTime();
}
void Stopwatch::restart() { reading = 0; }

void Stopwatch::stop() {
       measure();
}

void Stopwatch::measure() {
       if (mode == StopwatchMode::Real_Time)            reading += SimTK::realTime() -
ref_time;
       else if (mode == StopwatchMode::CPU_Time)  reading += SimTK::cpuTime() - ref_time;
}
```

TXTreadAngles.m

```
% Remove variables, clear Command Window and close any figures
clear;
clc;
close all;

% Change these to properly locate the txt file you want to plot
filename = 'Angles.txt';
config_type = 'Release';
build_dir = 'build';
path = sprintf('%s/%s/%s', build_dir, config_type, filename);
```

```
% Read txt
data = readmatrix(path);

% Get matrix sizes
datasize = size(data);
iterations = datasize(1);
statenumber = datasize(2);

% Define axis variables
Xaxis = []; XaxisRef = [];
Yaxis = []; YaxisRef = [];

% Assigning values
for i = 1 : iterations
    XaxisRef = [XaxisRef i];
    YaxisRef = [YaxisRef data(i,1)];
    Xaxis = [Xaxis,linspace(i,i,statenumber-1)];
    Yaxis = [Yaxis,data(i,2:statenumber)];
end

% Plot the values
hold on;
axes = gca;
axes.XLim = [0 iterations];
axes.YLim = [0 2*pi];
xlabel('Iteration')
ylabel('Angle [rad]')

scatter(Xaxis,Yaxis,'b','.')
scatter(XaxisRef,YaxisRef,'r','.')
```

TXTreadAngles_with_pause.m

```
% Remove variables, clear Command Window and close any figures
clear;
clc;
close all;

% Change these to properly locate the txt file you want to plot
filename = 'Angles.txt';
config_type = 'Release';
build_dir = 'build';
path = sprintf('%s/%s/%s', build_dir, config_type, filename);

% Read txt
data = readmatrix(path);

% Get matrix sizes
datasize = size(data);
iterations = datasize(1);
statenumber = datasize(2);

% Define axis variables
Xaxis = 1:iterations;
YaxisRef = data(:,1);

% Save figure axes handle
axes = gca;

% Plot each particle
for i = 2 : statenumber
    YaxisPt = data(:,i);
    scatter(Xaxis,YaxisRef,1,'r','.')
    hold on;
    axes.XLim = [0 iterations];
```

```
    axes.YLim = [0 2*pi];
    xlabel('Iteration')
    ylabel('Angle [rad]')
    scatter(Xaxis,YaxisPt,'b','.')
    hold off;
    pause(0.01);
end
```

TXTreadOmegas.m

```
% Remove variables, clear Command Window and close any figures
clear;
clc;
close all;

% Change these to properly locate the txt file you want to plot
filename = 'Omegas.txt';
config_type = 'Release';
build_dir = 'build';
path = sprintf('%s/%s/%s', build_dir, config_type, filename);

% Read txt
data = readmatrix(path);

% Get matrix sizes
datasize = size(data);
iterations = datasize(1);
statenumber = datasize(2);

% Define axis variables
Xaxis = []; XaxisRef = [];
Yaxis = []; YaxisRef = [];

% Assigning values
for i = 1 : iterations
    XaxisRef = [XaxisRef i];
    YaxisRef = [YaxisRef data(i,1)];
    Xaxis = [Xaxis,linspace(i,i,statenumber-1)];
    Yaxis = [Yaxis,data(i,2:statenumber)];
end

% Plot the values
hold on;
axes = gca;
Ymax = max(data(:));
Ymin = min(data(:));
axes.XLim = [0 iterations];
axes.YLim = [Ymin-0.5 Ymax+0.5];
xlabel('Iteration')
ylabel('Angular velocity [rad/s]')

scatter(Xaxis,Yaxis,'b','.')
scatter(XaxisRef,YaxisRef,'r','.')
```

TXTreadWeightedAngles.m

```
% Remove variables, clear Command Window and close any figures
clear;
clc;
close all;
```

```
% Change these to properly locate the txt files you want to plot
Angle_filename = 'Angles.txt';
Weight_filename = 'Weights.txt';
config_type = 'Release';
build_dir = 'build';
Angle_path = sprintf('%s/%s/%s', build_dir, config_type, Angle_filename);
Weight_path = sprintf('%s/%s/%s', build_dir, config_type, Weight_filename);

% Read txt
Angle_data = readmatrix(Angle_path);
Weight_data = readmatrix(Weight_path);

% Get matrix sizes
datasize = size(Angle_data);
iterations = datasize(1);
statenumber = datasize(2);

maxW = max(max(Weight_data));

hold on;
axes = gca;
axes.XLim = [0 iterations];
axes.YLim = [0 2*pi];
xlabel('Iteration')
ylabel('Angle [rad]')

for i = 1: iterations
    angles = Angle_data(i,:);
    weights = Weight_data(i,2:statenumber);

   for j = 1 : statenumber -1
       scatter(i, angles(1+j), '.', 'b', 'MarkerEdgeAlpha', weights(j)/maxW)
   end

    scatter(i, angles(1), '.','r')
end

% Remove variables, clear Command Window and close any figures
clear;
clc;
close all;

% Change these to properly locate the txt files you want to plot
Angle_filename = 'Angles.txt';
Weight_filename = 'Weights.txt';
config_type = 'Release';
build_dir = 'build';
Angle_path = sprintf('%s/%s/%s', build_dir, config_type, Angle_filename);
Weight_path = sprintf('%s/%s/%s', build_dir, config_type, Weight_filename);

% Read txt
Angle_data = readmatrix(Angle_path);
Weight_data = readmatrix(Weight_path);

% Get matrix sizes
datasize = size(Angle_data);
iterations = datasize(1);
statenumber = datasize(2);

maxW = max(max(Weight_data));

hold on;
axes = gca;
axes.XLim = [0 iterations];
axes.YLim = [0 2*pi];
xlabel('Iteration')
ylabel('Angle [rad]')
```

```
for i = 1: iterations
    angles = Angle_data(i,:);
    weights = Weight_data(i,2:statenumber);

   for j = 1 : statenumber -1
       scatter(i, angles(1+j), '.', 'b', 'MarkerEdgeAlpha', weights(j)/maxW)
   end

    scatter(i, angles(1), '.','r')
end
```