Trabajo Fin de Grado
Grado en Ingeniería Aeroespacial

Estudio de Simulación del Comportamiento de un Sistema Híbrido de Fabricación y Ensamblado de tipo Jobshop

Autor: Ángel Quiles Oñate

Tutor: Marcos Calle Suárez

Dpto. Organización Industrial y Gestión de Empresas I
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2022

Trabajo Fin de Grado
Grado en Ingeniería Aeroespacial

Estudio de Simulación del Comportamiento de un Sistema Híbrido de Fabricación y Ensamblado de tipo Jobshop

Autor:
Ángel Quiles Oñate

Tutor:
Marcos Calle Suárez
Profesor Titular

Dpto. Organización Industrial y Gestión de Empresas I
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2022

Trabajo Fin de Grado: Estudio de Simulación del Comportamiento de un Sistema Híbrido de Fabricación y Ensamblado de tipo Jobshop

Autor: Ángel Quiles Oñate
Tutor: Marcos Calle Suárez

El tribunal nombrado para juzgar el trabajo arriba indicado, compuesto por los siguientes profesores:

Presidente:

Vocal/es:

Secretario:

acuerdan otorgarle la calificación de:

El Secretario del Tribunal

Fecha:

Agradecimientos

A mis padres, que me mostraron el camino.
A mi hermano, que siempre está.
A mis abuelos, que siempre estarán.
A mis amigos, sin ellos no sería nada.

Resumen

En este documento se pretende estudiar el modelado y simulación de un entorno industrial, cuyo ritmo de fabricación se rige por una serie de *Workload Policies*, utilizando un software público y de código abierto como es el lenguaje de programación de *Python* y la librería *Simpy* de simulación de eventos discretos. Comparando los resultados de los modelos mas relevantes del *paper* original con modelos similares, aplicando otra regla de ordenación de colas, se estudiarán si el modelo es mejorable.

Abstract

This document pretends to study the modelling and simulation of an industrial enviroment, which manufacturing pace is ruled by a series of *Workload policies*, while making use of public and open source software such as *Python* and its discrete events simulation package, *Simpy*. Comparing the results of the most relevant models in the original paper with similar ones, using a different queue sorting criteria, the results will be tested for an upgrade.

-author's translation-

Índice

Notación

APP1	Estrategia de *Workload Control* 1, umbrales constantes
APP2	Estrategia de *Workload Control* 2, umbrales adaptables
ATO	Assemble to order
b_j	Nivel del buffer de tipología de componente j
CR	Regla de despacho del Ratio Crítico
DD_k	Fecha de entrega (*Due Date*) del pedido k
d_k	Retraso del pedido k, $d_k = T_{out} - DD_k$
EDD	Regla de despacho por Fecha de Entrega mas Próxima, del inglés *Earliest Due Date*
ETO	Engineer to Order
$Exp(\lambda)$	Distribución exponencial con parámetro característico λ
FIFO	Regla de despacho por Primero en Entrar,Primero en Salir, del inglés *First In-First Out*
i	Identificador del componente i-ésimo del tipo j
j	Identificador del tipo de componente, $j = 1,2,3,4,5,6$
k	Identificador pedido k-ésimo
K	Parametro de ajuste de umbrales de la estrategia APP2
LST	Regla de despacho Least Slack Time, Menor Margen de Tiempo
MTF	Make to Forecast
MTO	Make to order
MTS	Make to Stock
N_{in}	Número de pedidos recibidos
N_{out}	Número de pedidos completados
n_{max}	Número de iteraciones por configuración de simulación
N_{op}	Numero de operaciones totales
Q_j^k	Vector que indica el número de componentes j en el pedido k
Q_{comp}	Numero de componentes *distintos* presentes en un pedido
$t_{arrival}$	Tiempo de llegada, aleatorio
$\hat{t}_{process_k}$	Tiempo estimado de procesado del pedido k
$T_{arrival}$	Tiempo espera medio entre pedidos consecutivos
$t_{assembly}$	Tiempo de montaje
$Thrb_j$	Máximo número de componentes permitido del tipo j en el buffer j

Thr_{WIP}	Máximo número de trabajos en proceso de forma simultánea permitido en la planta (*Threshold WIP*)
T_k	Instante de llegada del pedido k
$t_{manufacture}$	Tiempo de procesado de una operación
T_{out_k}	Tiempo de salida del pedido k
t_{sim}	Horizonte de simulación
Tt_k	*Throughput time*, tiempo de procesado, $Tt_k = T_{out_k} - T_k$
U	Utilización, fracción del tiempo productivo entre el tiempo total de simulación
$U\,[a,b]$	Distribución uniforme con límite inferior a y superior b
WIP	Trabajo en proceso, del inglés *Work In Progress*

1 Objetivo

En este capitulo se presentarán los motivos que han llevado a la realización de este trabajo, así como su justificación y el alcance que pretende tener.

1.1 Presentación

En el último siglo, los sistemas de producción sufrieron un gran desarrollo y crecimiento, tanto en volumen como en eficiencia. Este salto se debe principalmente al surgimientos de nuevas técnicas de control y gestión de la producción, junto con el auge de la simulación y las herramientas de predicción de las demandas.

Sin embargo, en los pequeños talleres, en su mayoría artesanos, estas herramientas de predicción son excesivamente costosas o ineficientes para el volumen de producción con el que trabajan. Es por eso que para este tipo de sistemas de producción, las reglas de despacho tradicionales resultan una alternativa mucho mas útil. Simplemente siguiendo una estrategia EDD (Fecha de Entrega mas Temprana, del inglés *Earliest Due Date*) los resultados podrían mejorar considerablemente en determinados entornos, frente al clásico FIFO (Primero en Llegar-Primero en Salir, del inglés *First In-First Out*) empleado, por ejemplo, en numerosos talleres del automóvil locales.

El siguiente paso, sería una regla de control del ritmo de trabajo, dinámica pero suficientemente sencilla para poder ser aplicada en el día a día de un pequeño *job-shop* o taller. Como plantean *Renna et al* [1], con unas políticas de control del carga de trabajo *(Workload Control Policies)* se podría mejorar los resultados de un pequeño taller, al reducir los niveles de piezas almacenadas mientras se mantiene un nivel de producción suficientemente bueno.

Como declaran en las conclusiones de su estudio, esta estrategia de control «permite mejorar la utilización media y, por tanto, reducir el consumo de energía [...], además, la reducción del nivel de WIP reduce el coste de almacenamiento de los objetos», y por otro lado «el modelo propuesto utiliza una cantidad de información limitad y se caracteriza por una baja carga computacional, lo que corrobora a usabilidad del modelo propuesto en casos industriales», lo que refuerza la idea planteada anteriormente.

1.2 Objeto de trabajo

Este trabajo pretende continuar con el concepto planteado por *Renna et al* [1], recreando su modelo y explorando su estrategia con otro criterio de ordenación de colas.

Para ello, se recurrirá a la simulación con un lenguaje de programación libre como es *Python* y al paquete de simulación de eventos discretos *Simpy*. Ya que *Python* se trata de uno de los lenguajes mas usados en el entorno científico, el proyecto servirá también como demostración de las posibilidades de la simulación de eventos discretos en un software de uso libre.

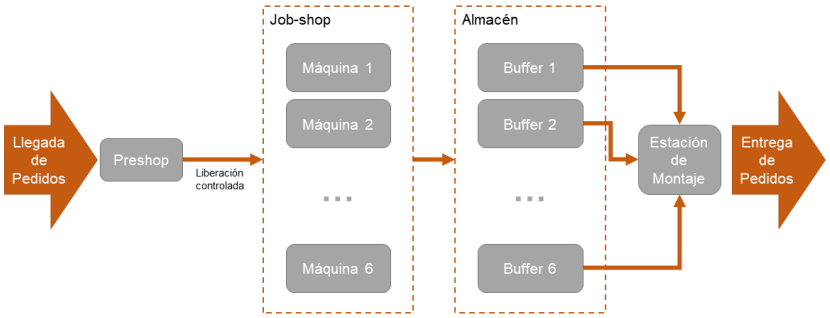

Figura 1.1 Representación del *layout* del taller [Fuente: *Renna et al.*].

Entonces, se tomará como punto de partida el modelo expuesto en *Renna et al*, usando los dos resultados mas favorables de *Workload Control policies* y ampliando el estudio existente mediante el análisis del comportamiento de este entorno productivo bajo la implementación de otro criterio de ordenación ampliamente utilizado, dado su buen comportamiento. Una vez recreado el modelo en *Python* se realizarán las simulaciones pertinentes y el tratamiento de los datos obtenidos para su comparación, con el fin de extraer conclusiones y posibles recomendaciones.

2 Sistemas de Producción

Para poder comprender el resto del proyecto, es necesario entender, al menos de forma superficial, los sistemas productivos y cómo funcionan. Este capítulo se centrará en explicar las bases de estos sistemas, comenzando por una introducción a la teoría y siguiendo con ejemplos de sistemas existentes, incluyendo un estudio de como encaja el modelo estudiado entre ellos.

2.1 Introducción

La producción es, tal y como explica James L. Riggs en su libro *Sistemas de Producción: Planeación, Análisis y Control* [2], el acto intencional de producir algo útil, así como la generación tanto de bienes, como de servicios. Esta definición tiene el valor de que no limita la forma en la que se produce, podríamos hablar de un coche, o del contenido de un libro.

En este proyecto, se tomará un sentido mas industrial, de este modo, otro concepto del que se hablará mucho es el de *sistema de producción*, el cual puede ser definido como el proceso organizado mediante el cual los elementos son transformados en productos útiles. De esta forma, para una empresa que empieza un proyecto, elegir un sistema de producción es una decisión crucial, pues no solo conlleva elegir un proceso de manufactura para el bien deseado, sino que incluye la gestión, organización de todos los recursos de materia, maquinaria, etc. que se engloban dentro de la producción. En definitiva, el sistema contempla tanto los procesos como las actividades necesarias para realizar la transformación de los elementos iniciales a los productos o servicios deseados.

Al comienzo, los sistemas de producción existentes eran mucho mas simples que los actuales y por tanto la necesidad de realizar un estudio de mejora era prácticamente inexistente. Pero con el paso del tiempo, los procesos se volvieron complejos, teniendo utilizando componentes que provienen de otro sistema y proporcionando productos a otros sistemas que lo utilizarán como puntos de partida, creándose un enrevesado sistema de sistemas.

Figura 2.1 Retrato de Friedrick W. Taylor [Fuente: www.biografiasyvidas.com [3]].

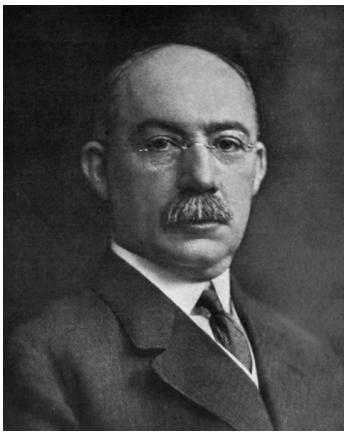

Figura 2.2 Retrato de Henry L. Gantt [Fuente: www.biografiasyvidas.com [4]].

La revolución industrial trajo el desarrollo de la industria, la producción se aceleró, la busqueda de la reducción de costes por la fuerza bruta llevó, en primer lugar, a centrar las mejoras en el objeto pero sin tener en cuenta las condiciones de los trabajadores, aún así aparecieron conceptos como la distribucion en planta por departamentos y registros mejorados de costes.

Ya en el siglo XX, un nuevo enfoque, con tendencia "científica", entró de lleno en la gestión y administración de la producción, de la mano de nombres como Frederick W. Taylor padre de la corriente de pensamiento llamada *administración científica* o *Taylorismo* y Henry L. Gantt, creador del diagrama homónimo que permite planificar y visualizar los distintos procesos de un proyecto. Esta tendencia de pensamiento, prefería contemplar la máquina y el operario como una unidad, donde uno de los factores en la efectividad era el salario de la persona encargada. Mas adelante, se comenzaría a tener en cuenta la fatigua, la monotonía o las reacciones emocionales como factores que afectaban a la productividad, lo que convertiría al conjunto trabajador-máquina en algo poco intuitivo, agravándose la situación ante la llegada de sistemas de producción cada vez mas complejos. Pese a esto, se tendió a la mejora de las condiciones laborales y la automatización, a la producción en serie (conocida a partir de su implantación en las fábricas de *Ford*).

A finales del último siglo, la crisis económica de los 80 incentivó a los empresarios americanos a importar filosofías de producción asiáticas, que habían implantado empresas como Toyota. Actualmente, existen estrategias y filosofias de administración de la producción que buscan eliminar poco a poco cualquier gasto innecesario asociado a la actividad productiva, el *Lean Manufacturing*, que se inspira y aúna diversos principios orientales. La automatización de los procesos parece el camino, y aunque exista el miedo al desempleo que pueda surgir por la sustitución de la mano de obra, lo cierto es que las condiciones para los trabajadores han ido mejorado, eliminando tareas peligrosas y monótonas.

2.2 Sistemas de Producción Básicos

Existen dos estrategias básicas para la organización de la producción, el *Make to Stock* (MTS) y el *Make to Order* (MTO), presentan enfoques totalmente opuestos. En la actualidad se han derivado numerosas filosofías que intentan aunar lo mejor de ambos sistemas, como el *Assemble to Order* (ATO) o el *Make to Forecast* (MTF). También existen algunas estrategias mas extremas como el *Engineer to Order* (ETO).

2.2.1 Make to Stock (MTS)

El sistema *Make to Stock* (en castellano, Fabricar para Almacenar), es utilizado cuando sólo existe una sola variedad de producto y la demanda no presenta grandes variaciones a corto plazo. La idea

Figura 2.3 Una linea de producción de ruedas en 2014 en Jianxing, China [Fuente: www.nytimes.com].

detrás de este sistema es que la producción sea continua, fabricándose el producto sin que el cliente haya realizado un pedido, y almacenándose para cuando se necesite.

Este sistema se incluye en los conocidos como sistemas *Push*. Según son definidos en *Diseño y Gestión de Sistemas Productivos* [5], estos sistemas son en los que la demanda "empuja" la producción con el objetivo de cumplir los plazos. El principio es simple, se produce sin parar para cuando sea necesario. El sistema se comporta como un flujo constante de material, para ello, los productos deben ser lo mas simples posibles, sin variaciones.

Para que los costes por inventario no se hagan inasumibles, es necesario que la demanda sea relativamente estable, al menos a largo plazo, ya que el sistema es inflexible y muy complicado de reajustar. Por tanto, el dimensionamiento inicial de la producción es clave en este sistema, un sistema mas grande generará costes de almacenaje excesivos, y uno pequeño no suplirá la demanda de los clientes a tiempo.

La ventaja de este sistema se da para el cliente, ya que puede disponer del producto terminado de forma instantánea a realizar el pedido. Reincidiendo en el dimensionamiento del sistema, si el sistema es pequeño y no se alcanza la demanda, se pierde esta ventaja de la rapidez de entrega.

En resumen, podemos agrupar las ventajas de este tipo de sistema como las siguientes:

- Los tiempos de producción y de espera por parte del cliente son mínimos. El primero, por la simplicidad requerida al producto final y el segundo, gracias al almacenaje de los productos acabados.
- Altamente automatizable. El que sea un producto simple y que se fabrique en altas cantidades, hace que su la automatización sea la mejor opcion, lo que reduce costes y tiempos.
- Baja intervención humana. Directamente relacionado con el punto anterior, una alta automatización implica una necesidad menor de intervención humana, además de una formación mas reducida de la mano de obra.

Y los principales problemas que presenta:

- Inversión inicial muy alta. Es necesario la compra del sistema de automatización de la producción, la maquinaria y los almacenes.

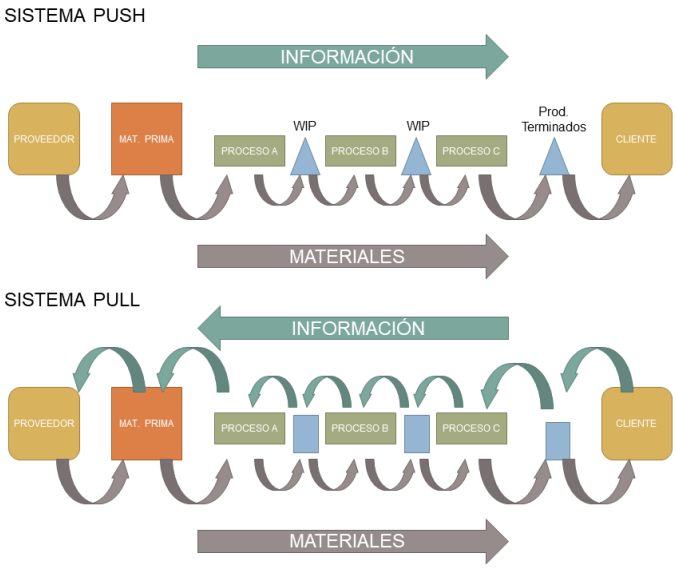

Figura 2.4 Diagrama comparativo entre los sistemas *push* y *pull* [Fuente: Elaboración Propia].

- Sistema muy inflexible. El alto grado de automatización impide cambios y/o personalización de los productos sin una reconfiguración del sistema completo. Incrementar o reducir el ritmo de producción es algo menos costoso.
- Puesta a punto compleja. El diseño y puesta a punto del sistema es cara y compleja por culpa de la automatización, lo que no hace rentable reconfigurarlo a menudo.

Los sistemas MTS siguen siendo usados en la actualidad, los encontramos en productos que pueden ser almacenados largas cantidades de tiempo (no perecederos) y que poseen una demanda estable, además de ser productos poco personalizables. Este es el caso de tornillos, rodamientos, neumáticos y pilas, productos que se encuentran limitados por unos estándares de diseño y que son necesarios constantemente, pero también se usa en el caso de refrescos, la harina o incluso la leche. Para el escenario en el que vamos a trabajar, el de un taller *(job-shop)* no es la estrategia mas acertada, debido al volumen de producción y las diferencias entre ordenes, como se verá mas adelante.

2.2.2 Make to Order (MTO)

Los sistemas *Make to Order* también conocidos como *Build to Order*, en castellano Fabricar bajo Pedido, consisten en esperar a que un cliente haga un encargo para comenzar a fabricar el producto. Para ello, la empresa ha debido con anterioridad ofrecer sus productos y posibilidades al mercado, y es una vez que se firme un contrato, cuando se prepara la producción del encargo.

Frente al MTS, el MTO es un sistema *pull*, en el que es la demanda la que "tira" de la producción, de igual forma de la que dentro de la fabrica de estos sistemas, una estación de trabajo "tira" de la producción de la estación aguas arriba cuando lo necesitan, creándose, como explican *Onieva et al* [5], una demanda interna. Se genera entonces un flujo de información que viaja aguas arriba, desde el cliente, hasta los proveedores, pasando antes por las etapas productivas; al contrario que los sistemas MTS, donde la información viajaba aguas abajo (la información bajaba en forma de *deadline* por la línea de producción), como se ve en la figura 2.4.

En este caso, al ser el cliente el que establece el inicio de la producción, existe la posibilidad de personalizar el producto final a las necesidades del cliente (en el MTS el producto se encontraba ya listo cuando se establecía el pedido). Esta posibilidad da lugar a que productos mas complejos sean producidos, pero tienen que estar definidos al menos en parte. El mejor ejemplo se suele encontrar

Figura 2.5 *Final Assembly Line* del A400M en San Pablo, Sevilla [Fuente: fly-news.es].

en la industria aeronáutica, donde un cliente, ya sea una aerolínea o un ejército, encarga un avión configurando los equipos embarcados, motorizaciones, etc, según sus necesidades y preferencias (diferenciar de los sistemas ETO, que se verán mas adelante).

A diferencia de los sistemas *Make to Stock*, salvando pequeños inventarios intermedios, no hacen falta almacenar el producto final, ya que una vez terminado se entrega al cliente directamente. Si en un sistema de este tipo, se optimizan los tiempos de forma que los componentes lleguen a la vez a las estaciones de trabajo, sin ningún tipo de retraso, los inventarios intermedios son innecesarios, se habla entonces de un sistema *just-in-time* (justo a tiempo en castellano). Sin embargo, el cliente tendrá que esperar a que el producto se termine, ya que se empieza a producir cuando lo establece el pedido.

A modo de resumen, presenta las siguientes ventajas:

- Productos de muy alta calidad y personalizados.
- Producción equiparada a la demanda existente, pues se fabrica bajo pedido.
- Mayor flexibilidad, comparada con sistemas MTS.

Pero también presenta algunas desventajas:

- Mayores costes de producción, sobre todo comparando con MTS.
- La formación necesaria para los trabajadores es mayor, pues los productos son mas complejos y es necesario especialistas.
- Tiempos de espera mas largos para el cliente.

2.2.3 Otros sistemas

Con el desarrollo de la industria, también se desarrollaron los sistemas productivos, apareciendo desarrollos de los dos comentados anteriormente. A continuación, se presenta una lista de algunos, pero no se profundizará mas ya que se escapa de alcance de este proyecto.

- *Engineer to Order, ETO*
  - En este caso, el cliente plantea un problema y sus limitaciones, y la empresa contratada se encarga de cercar el problema y diseñar el producto solución, a demás de fabricarlo. El producto es, por tanto, lo mas personalizado posible.

  - El tiempo de espera del cliente es el mayor de todos, pues incluye la fase de diseño, fabricación e incluso de certificación.

- *Assemble to Order, ATO*
  - Punto medio entre MTO y MTS, en donde se fabrican y almacenan los componentes del producto final con una filosofía similar a la MTS, y cuando se establece un pedido, se montan según los requerimientos del cliente.
  - El tiempo de espera se ve reducido con respecto al MTO, puesto que solo se incluye el de montaje, las piezas están ya listas.
  - Para que exista personalización, las piezas deben tener algún tipo de modularidad. Un ejemplo de este sistema pueden ser coches, donde se monten los extras según la demanda del cliente.
- *Make to Forecast, MTF*
  - Híbrido de los sistemas MTS y MTO, donde se fabrica según se prevé que los clientes vayan a presentar un pedido.
  - Los productos finales deben ser tan simples como en el MTS, pero la ventaja es que si funciona bien, los costes por inventario se reducen drásticamente, manteniendo la velocidad de entrega al cliente.
  - El principal problema de este sistema es la previsión de las variaciones de la demanda. Hace falta en primer lugar un modelo de predicción con un porcentaje de acierto alto, y después, un sistema productivo capaz de absorber las variaciones del ritmo de producción, lo que implica que habrá momentos en los que la planta se encuentre sobredimensionada y ociosa, para no saturar cuando se prevé un pico de demanda.

La realidad es mucho mas compleja, y existen muchos mas sistemas de los comentados en este documento, se podría tener un taller que fabrique componentes con una filosofía MTF, pero que los monte según el cliente lo pida, hibridándolo con un ATO, por ejemplo. De cualquier modo, es mas que suficiente para continuar con el proyecto.

Tabla 2.1 Comparativa de las características esenciales de los distintos sistemas de producción básicos..

Estrategia	Personalización	Limitación	Velocidad de Entrega	Coste Personalización	Complejidad de gestión	Reto
ETO	Plena, única	Ilimitada	Muy lenta	Muy Alto	Alta	Diseño y fabricación
MTO	Amplia	Gamas de fabricación	Lenta	Alto	Moderada	Organización
MTF	Moderada	Oferta	Moderada	Moderado	Muy alta	Adivinar demanda
ATO	Poca	Módulos de montaje	Rápida	Poco	Poca	Planificación de inventarios
MTS	Ninguna	Variedad almacenada	Inmediata	Ninguno	Muy baja	Demanda estable

2.3 Sistemas de Control

Los sistemas de control, se pueden definir como el conjunto de procesos y actividades centradas en vigilar y controlar el progreso de los pedidos por las distintas estaciones, liberar de forma progresiva los pedidos a la fábrica y recabar información del estado de los pedidos y la planta en sí; como explica Mikell P. Groover en su libro [7].

Groover divide el control en tres fases, la *liberación de los pedidos*, la *planificación de los pedidos* y el *progreso de los pedidos*, pero en este proyecto haremos una distinción entre *control en la entrada* y el *control a pie de planta*, donde el primero incorpora a la liberación y el segundo incorpora las dos últimas fases de Groover.

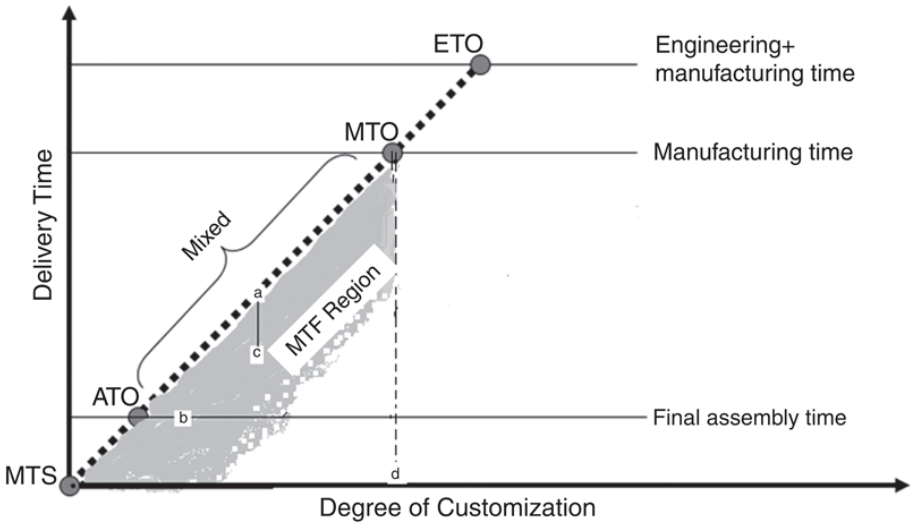

Figura 2.6 Gráfico tiempo de entrega frente a grado de personalización cualitativo de las regiones donde es mas útil aplicar un sistema de los comentados [Fuente: International Journal of Operations & Production Management [6]].

2.3.1 Regulación en la Entrada

Cuando un pedido entra en producción, este llega con la información estrictamente necesaria para ser procesada en la planta. Esta información dependerá del tipo de planta y producto en cuestión, pero puede incluir la hoja de ruta del producto por el taller (que operaciones deben ser realizadas, en qué estación y en que orden), los requisitos de materiales, la lista de piezas necesarias y diversas etiquetas sobre transporte y gestión.

Una vez que el pedido ha entrado en la planta, se cuenta para los datos, se tiene que gestionar su prioridad en el taller y sus estadísticas cuentan para las estadísticas globales del taller, por ello, puede resultar interesante controlar cuando se "libera" un pedido, con el objetivo de no sobrecargar a los trabajadores y las máquinas.

Workload Control

Existen diversos criterios y mecanismos para regular la entrada. En el artículo estudiado, se utiliza un criterio llamado *Workload Control Policy* (Política de Control de Carga de Trabajo), que elige cuando liberar el pedido en función de la carga de trabajo de la planta.

En este caso en concreto, evalúa la carga como el número de pedidos en proceso al mismo tiempo en la planta, el número de pedido *Work-In-Process (WIP)*. Para ello, los autores establecen un umbral máximo de pedidos que se pueden estar trabajando al mismo tiempo, este criterio se debe evaluar antes de liberar cualquier encargo al taller. Se puede escribir como:

$$WIP \leq Thr_{WIP} \tag{2.1}$$

Siendo WIP el número de pedidos en proceso, y Thr_{WIP} el umbral (*threshold*) o número máximo de pedidos en proceso permitido. Siguiendo este criterio, los encargos de clientes que lleguen cuando esta condición no se cumpla, deberán esperar a cuando se cumplan. Los pedidos se almacenan mientras esperan a ser procesados en una lista, el *preshop*, que puede ser ordenada según la prioridad elegida; por lo que podríamos tomar al sistema como una máquina y al preshop como su cola.

Estos criterios pueden incurrir en retrasos si la condición es muy estricta, por lo que habría que jugar con el umbral. Este umbral puede ser arbitrario, o venir dado por el límite de capacidad de la planta, en todo caso, es útil limitar que un pedido entre en producción si sabe que sus componentes van a estar esperando mucho tiempo, lo que implica costes de inventario.

2.3.2 Regulación a nivel de Planta

Por otro lado, es también posible regular el flujo de las ordenes por el taller. En la actualidad, es muy sencillo mantener un control del estado y el progreso de los componentes de un pedido, gracias a la tecnología y los distintos métodos de etiquetado, lo que hace mas sencillo visualizar el punto en el que se encuentra el taller, pudiendo tomar decisiones en cada momento acorde a la situación.

Un método muy sencillo para regular flujo de los pedidos es dar prioridad a unos antes que a otros, de este modo, es posible controlar la carga de de las estaciones de trabajo y el orden de las colas.

Para regular la carga de las estaciones, tiene sentido repartir la carga, de este modo, si se tienen varias máquinas que realizan la misma tarea, resulta absurdo asignarle una nueva operación a la maquina cuya cola es mas larga, pudiendo dejar alguna sin carga alguna. En cuanto al orden de las colas, dependiendo del caso, puede resultar interesante fabricar algunos pedidos antes que otros, es donde entran las reglas de despacho.

Reglas de despacho

Existen varios métodos para asignarle prioridad a una tarea frente a otra, es posible asignarle un valor a cada una mediante una función que pondere diversos criterios, como el tiempo que falta hasta la entrega, el precio de venta, el tiempo estimado de procesado, etc. Sin embargo, si se pretende optimizar algún parámetro, ordenar la cola puede llegar a ser una tarea casi imposible, incluso con potentes ordenadores.

Las reglas de despacho, por su parte, son muy simples, de uso instantáneo y no necesitan apenas cálculo alguno. En algunos casos muy simples, pueden llegar a garantizar la solución óptima en cuanto a costes y tiempos de espera del cliente, lo que en la realidad sugiere buenos resultados si se saben usar. Algunas son:

- *FIFO*. Del inglés *First In-First Out* (Primero en Entrar-Primero en Salir), es una cola tradicional, los encargos, piezas, operaciones, etc. se procesan en el orden de llegada.
- *EDD*. Del inglés *Earliest Due Date* (Fecha de Entrega mas Temprana), se coloca primero en la cola el que vaya a ser entregado antes. Intenta minimizar los retrasos.
- *LST*. Del inglés *Least Slack Time* (Menor Margen de Tiempo), se le da prioridad al que tenga menos margen o *slack time*, que se define como tiempo que queda hasta la fecha de entrega menos el tiempo estimado de fabricación, $S_k = (DD_k - t) - \hat{t}_{process_k}$, siendo t el instante en el que se calcula este parámetro y $\hat{t}_{process_k}$ el tiempo estimado de fabricación.
- *Critical Ratio*. O razón crítica, se define como la fracción entre el tiempo restante hasta la entrega y el tiempo estimado de entrega, $CR_k = (DD_k - t)/\hat{t}_{process_k}$.

Donde se ha tomado para un pedido *k-ésimo* DD_k como la fecha de entrega (Due Date), T_{0_k} como el momento de llegada y $\hat{t}_{process_k}$ como el tiempo estimado de procesado.

2.4 Mediciones Interesantes

A continuación, se exponen una serie de variables que pueden resultar útiles para valorar un sistema productivo.

2.4.1 Pedidos completados

El propio nombre es es auto-explicativo, dado un tiempo determinado, cuantos pedidos puede el sistema fabricar y entregar terminados al cliente. Se denotará como N_{out}.

Presenta un límite superior obvio, el número máximo de pedidos completados será igual al numero de de pedidos recibidos en ese mismo intervalo de tiempo $N_{out} \leq N_{in}$.

Para un mismo intervalo de tiempo, nos da una idea de qué sistema es más productivo, pero no tiene por que implicar que sea mas rentable, habría que estudiar otras variables, como los retrasos, o los niveles de inventario.

2.4.2 Retraso

El retraso se mide como la diferencia entre el tiempo de entrega y la fecha de entrega pactada con el cliente, positiva si llega tarde, negativa si se entrega antes de tiempo. Para un pedido k se calcula como $d_k = T_{out_k} - DD_k$, donde T_{out_k} es el instante en el que se termina el pedido.

Un sistema con un retraso bajo mantendrá al cliente satisfecho, pero si es excesivamente bajo es posible que el sistema se encuentre sobredimensionado.

2.4.3 Tiempo de procesado

En función del criterio, el tiempo de procesado se puede medir considerando o no el tiempo de espera del pedido en el preshop. En este proyecto se considerará el tiempo de espera antes de ser liberado el pedido, ya que desde el punto de vista del cliente es el tiempo que el pedido pasa en proceso. De este modo, la expresión es mucho mas sencilla, $Tt_k = T_{out_k} - T_{0_k}$.

El tiempo de procesado, en inglés *throughput time*, da una idea de lo rápido que es el sistema; junto con el numero de pedidos completados, pone en perspectiva la capacidad productiva del sistema; no es lo mismo un producto que tarde en fabricarse 2 horas que uno que tarda 1 día.

2.4.4 Pedidos en cola

Aquí con cola se hace referencia al *preshop*, por lo que se está hablando de la cantidad de pedidos que se encuentran esperando a ser liberados en cierto momento. Dependerá del criterio de liberación; si es muy estricto, esta cola crecerá.

2.4.5 Pedidos en proceso

Otra variable auto-explicativa, aquí se consideran el número de pedidos en los que se está trabajando en cada momento, sin contar los que esperan a ser liberados en el preshop. Es equivalente al parámetro que se usó antes para establecer un criterio de liberación, el *WIP*.Da una idea de la carga de trabajo en cada momento.

2.4.6 Utilización

También conocida como la ocupación, mide la fracción del tiempo en los que la estación de trabajo o máquina está siendo productiva, a veces resulta mas sencillo medir el tiempo ocioso que el tiempo ocupado $U = t_{prod}/\Delta t = \left(\Delta t - t_{empty}\right)/\Delta t$.

Para un sistema, se puede medir la media de la utilización de cada estación, maquina u operario, con el objetivo de dar una visión global del sistema; sin embargo, el desglose de utilizaciones puede resultar útil para localizar cuellos de botella.

2.4.7 Niveles de los búferes

En ciertos sistemas se utilizan pequeños inventarios para compensar la variabilidad del ritmo de producción y de la demanda. Son los denominados búferes (del inglés *buffer*) y, en el caso de que se produjese un aumento repentino de la demanda, es posible vaciarlos para suplirla, amortiguando el aumento del ritmo de producción que haría falta para compensar. Es posible también que actúen en sentido contrario, en el caso de una bajada del ritmo de producción (por una avería, una baja, etc), garantiza poder suplir la demanda.

Unos niveles de búferes muy altos pueden acarrear unos costes elevados de inventario, por lo que es interesante limitarlo. Además, mantener esos búferes supone mantener el ritmo de producción aunque haya bajado la demanda.

3 Simulación

Una vez se conocen las características principales y como funcionan los sistemas productivos, es útil replicarlos para comprobar las hipótesis supuestas o cómo varían ante variaciones de las condiciones del entorno. En esta situación, la simulación se convierte en una herramienta clave para el desarrollo de nuevas estrategias.

3.1 Introducción a la simulación

La simulación surgió, según *R. McHaney* [8], para reducir el riesgo que supone realizar cambios en sistemas existentes o emplear sistemas nuevos sin ser probados, ya se trate de un riesgo económico o material. Antiguamente se usaban magos, videntes u el horóscopo, pero en el mundo en el que vivimos actualmente, la predicción se basa en la experiencia previa, en modelos establecidos por la ciencia y que poseen una exactitud y precisión que los videntes no podrían ni imaginar, pero que, a día de hoy siguen sin ser cien por cien exactas.

Como herramienta, la simulación, permite experimentar sin afectar al mundo real, se consigue de este modo encontrar errores y comportamientos inadecuados con antelación. Independientemente de la aplicación (física, química, matemáticas, sociología, ingenieril...) la meta de la simulación es conseguir emular la realidad con la mínima desviación posible. Saber que va a pasar es clave para la toma de decisiones estratégicas o de producción, y mientras mas fiable sea la simulación, menos riesgo se corren al tomar estas decisiones.

3.1.1 El modelo

Toda simulación se basa en un *modelo* de la realidad, que permite sacar conclusiones sobre el comportamiento de los elementos estudiados. Un modelo no es más que una representación de lo que se pretende estudiar; el modelo, recoge los aspectos que se consideran mas relevantes para la experiencia y se intentan replicar. En el qué se elige replicar y cómo se replica en el modelo reside el cuanto de fiable y como se corresponderá con la realidad. Dependiendo de la aplicación, se debe valorar si se prefiere un modelo simple, barato y rápido que de resultados aproximados frente a un modelo complejo, caro y con mayor tiempo de simulación, pero que de resultados mas precisos.

Es posible clasificar los modelos en distintos tipos, *Whicker* y *Singleman* [9] proponen una división formal de los modelos:

- *Modelos Físicos*. Son modelos a escala, pretenden replicar el sistema real de forma mas económica y aislada. Se incluyen aquí, desde una maqueta de una estructura, hasta un túnel de viento.

- *Modelos Esquemáticos*. En ellos se pretende mantener ciertas características de un sistema, sin importar que se pierdan otras. Es el caso de los mapas y planos, que pierden información volumétrica, pero que, aún así, permiten simular distribuciones y organización de espacios.
- *Modelos Simbólicos*. Son representados mediante la matemática o alguna codificación informática. Por ejemplo, las ecuaciones que rigen el movimiento de un tiro parabólico no son mas que un modelo, donde se han recogido las características que mas interesan y se han obviado muchas variables (rozamiento, aceleraciones de Coriolis...), aquí entran también los modelos informáticos (CFD, elementos finitos...).

Desde otro punto de vista, según la capacidad de ser replicados, se pueden dividir los modelos en *deterministas* y *estocásticos*. En los primeros, dado un mismo modelo con unas mismas condiciones iniciales, el resultado final tras la simulación va a ser siempre el mismo, aquí se incluyen los problemas de cinemáticos clásicos de física, las ecuaciones son siempre las mismas, los resultados serán siempre iguales. En contra, en los modelos estocásticos interviene el azar, lo que implica que dadas unas condiciones iniciales, el resultado final puede variar, esto complica mucho la simulación, pues hay que modelar el azar, y esto depende del sistema que se intente replicar. Gracias a la estadística y la probabilidad, han surgido formas de modelar el azar que son aplicables a la mayoría de las situaciones de la naturaleza. Este azar puede surgir por diversos motivos, pequeñas imprecisiones en las condiciones iniciales, en un modelo muy sensible pueden hacer que diverjan los resultados aunque en su comienzo fuese imperceptible; o a elementos externos que influyen en el sistema pero que son demasiado complejos para modelarlos con una ecuación determinista, eventos sociales, mecánica cuántica, etc.

3.1.2 El paso del tiempo

Por otro lado, considerando al modelo como algo estático, pues no deja de ser un conjunto de reglas y características; la simulación consiste en coger el modelo en cuestión y observar su evolución a lo largo del tiempo, en donde el modelo va variando y evolucionando sus características iniciales. Es tras el paso de este tiempo, cuando se pueden extraer resultados y conclusiones.

Sin embargo, existen varias estrategias sobre como hacer el tiempo, ya que desde la aparición de la simulación por ordenador, se abrió la puerta a realizar simulaciones con un reloj que avance a distinto ritmo que el real. Antes, las simulaciones se hacían observando la interacción entre personas o mediante reproducciones de los sistemas en condiciones controladas, lo que limitaba el paso del tiempo al real. Pero ahora, se puede variar el ritmo de la simulación en los ordenadores, ya sea mas lento, lo que permite apreciar detalles que se escapan; o mas rápido, lo que ahorra tiempo al poder realizar mas simulaciones y comprobar varias casuísticas en un mismo tiempo. Es por eso que se consideran distintos tipos de simulación en función del tipo de paso de tiempo que tengan, generalmente se distingue entre los siguientes tipos, tal y como explica Roger McHaney [8].

Simulación de Montecarlo

El nombre hace referencia al famoso casino monegasco y a las partidas de azar que allí se juegan, pero lo popularizó *Von Neumann* durante el desarrollo de la bomba atómica, pues llamaba así a los experimentos que realizaba con números aleatorios para resolver problemas de difusión de neutrones. En estos experimentos, el paso del tiempo no tiene ningún peso en el resultado de la simulación, simplemente presenta unos números aleatorios como un input del sistema.

Requieren de numerosas repeticiones de los experimentos para poder sacar conclusiones, pues a mayor número de experiencias, mayor seguridad tendrá el resultado. La gran ventaja es que son perfectas para un ordenador, que puede realizar las cuentas de una forma directa y repetir la simulación numerosas veces por minuto.

Este tipo de simulaciones es usada en numerosos campos, desde calcular la probabilidad de fallo de un sistema complejo, a partir de las probabilidades de fallo de componentes individuales; hasta

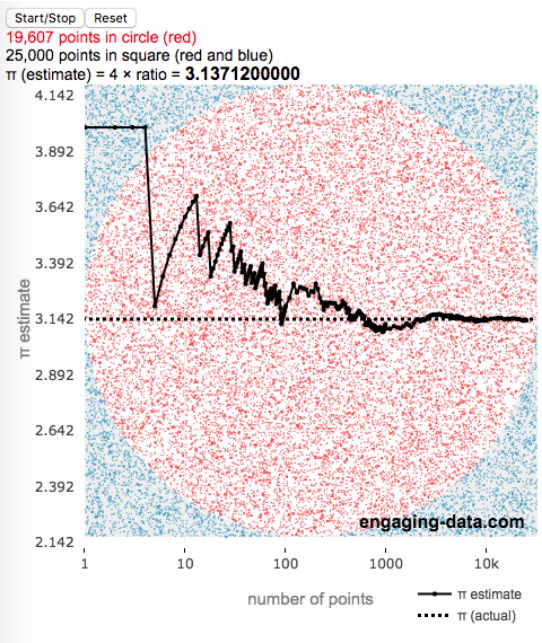

Figura 3.1 Experiencia del calculo de pi, mediante Montecarlo, se ve la tendencia hacia el valor [Fuente: engaging-data.com].

videojuegos, para calcular sombras mediante técnicas de *ray-tracing*. Existe también un método para estimar π a usando Montecarlo, lanzando puntos con coordenadas completamente aleatorias a un círculo inscrito en un cuadrado unidad, basándose en que para un número de lanzamientos muy alto, la fracción de puntos dentro del círculo respecto del total se parece mucho a la relación entre las áreas del círculo y el cuadrado, de donde se despeja π.

$$\frac{n_{in}}{n_{total}} \approx \frac{S_{circle}}{S_{square}} = \frac{\pi}{4}$$

Este método para calcular relaciones de áreas se puede usar par integrar áreas bajo cualquier curva, comparando las coordenadas con el valor de la función, llamado integración de Montecarlo.

Simulación Continua

Ahora si entra en juego el paso del tiempo como una variable de la simulación. En el caso de la simulación continua, se centra en una representación de un sistema mediante una serie de ecuaciones temporales, ya sean ecuaciones algebraicas, diferenciales o modelos estadísticos continuos. *McHaney* [8] ejemplifica este tipo de simulación con el ejemplo de los modelos depredador-presa, donde hay dos poblaciones y una se alimenta de la otra; de este modo el crecimiento de cada una, dependerá de la población del otro.

$$\frac{\partial x}{\partial t} = \alpha x - \beta xy$$
$$\frac{\partial y}{\partial t} = \delta xy - \gamma y$$

Estas simulaciones se usan para infinidad de aplicaciones, cálculo de estructuras, simulaciones hidráulicas, de poblaciones, tiempo atmosférico, expansión de una enfermedad, etc. en general cualquier sistema en el que su comportamiento pueda expresarse como bucles de retroalimentación o ecuaciones continuas con variables interrelacionadas es candidato a ser simulado de esta forma.

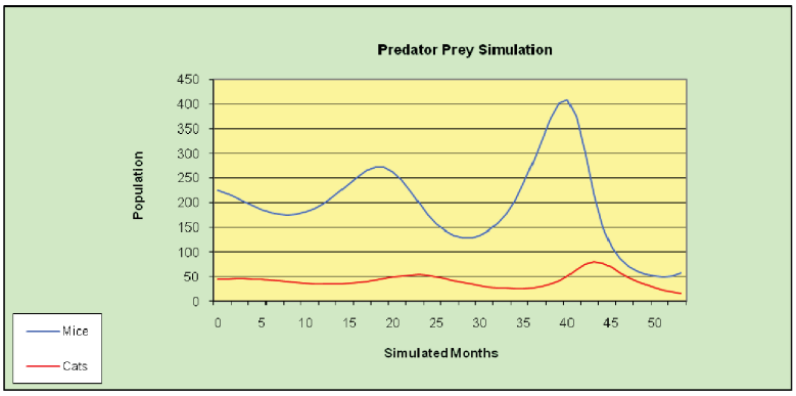

Figura 3.2 Gráfico sacado de una simulación continua de un modelo depredador-presa [Fuente: *McHaney, R.* [8]].

Estas simulaciones funcionan muy bien, pero presentan el problema de encontrar las ecuaciones que rigen el problema, ya que para determinados problemas, como se verá mas adelante, las variables no son continuas, como es el caso de la llegada de pedidos a una fabrica. Además, los ordenadores no son capaces de trabajar con variables continuas, necesitarían una memoria infinita, es por ello que para este tipo de simulaciones se tiende a discretizar el problema, tomar intervalos de integración, etc., mientras mas pequeños mejor.

Simulación de Eventos Discretos

Las simulaciones de eventos discretos se caracterizan por "discretizar" el tiempo en bloques, durante los que no ocurre nada importante, ya que el sistema solo cambia de estado en los *eventos* instantáneos que ocurren entre cada *step* (paso) de tiempo. Todas las variables salvo el tiempo permanece constante entre cada evento. Esto resulta ser una solución de compromiso en muchos casos, pues hay variables con no pueden cambiar instantáneamente de estado, pero resulta muy útil para modelar entradas y salidas de elementos en un depósito o una cola, lo que resulta clave a la hora de simular la dinámica de sistemas productivos.

Ya que en un proceso productivo las operaciones se realizan siguiendo unas instrucciones paso a paso, lo que es fácilmente trasladado a los pasos de tiempo discretos de este tipo de simulación. En un sistema productivo es fácil encontrarse con un tiempo de montaje, un tiempo de fabricación, un tiempo de empaquetado, de transporte, tiempo de espera a la llegada de recursos o a la disponibilidad de una herramienta o máquina.

En cuanto a los eventos, son instantáneos, cambian las variables de estado del sistema y pueden surgir a raíz de otros eventos, o generarlos. Un ejemplo sencillo sería el proceso que sigue una pieza desde su llegada a una maquina, su procesado y el envío de esta hasta que llega a la siguiente estación. Como se puede ver en la figura 3.3 el proceso para una línea es sencillo, pero cuando hay dos en paralelo, el reloj de simulación debe pararse en cada evento; durante el tiempo que la simulación procesa el evento de una línea, el estado de la otra se mantiene constante.

Existen dos enfoques a como pasar el tiempo del reloj de simulación en este tipo de simulaciones, por un lado existen softwares que establecen un *tiempo de progreso fijo*, en los que el tiempo es discretizado en pequeños pasos y en la simulación se avanza uno a uno parándose en cada uno y comprobando si existe algún evento activo en esa porción de tiempo. Por otro lado, existen softwares de simulación que poseen una lista de los eventos presentes en la simulación junto con el instante de tiempo en el que se llevan a cabo, antes de avanzar el tiempo, el software revisa en la lista cual es el siguiente evento, y mueve el reloj hasta ese punto; cabe recordar que los eventos pueden generar otros eventos, que se irán añadiendo a la lista progresivamente.

La estrategia de saltar al siguiente evento, llamada en inglés *next-event advance*, puede ser, generalmente mas rápida; ya que no necesita simular los intervalos entre un evento y otro.

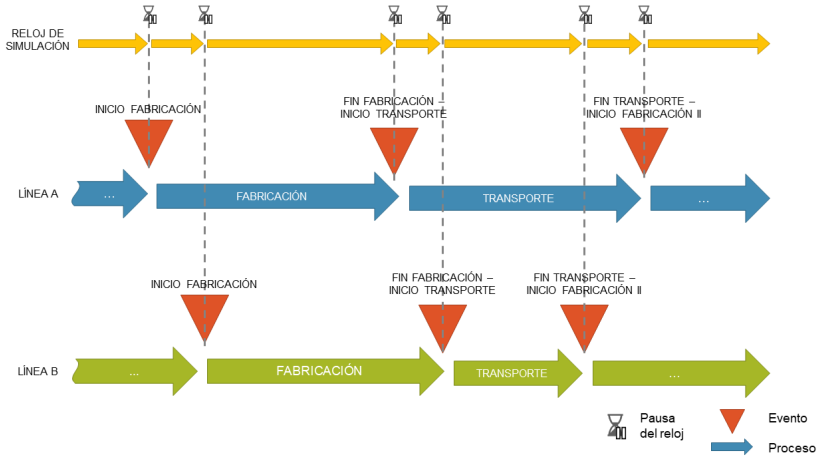

Figura 3.3 Diagrama donde se puede ver como se entrelazan las pausas en una simulación de eventos discretos con dos líneas paralelas, las paradas corresponderían con una simulación *next-event advance* [Fuente: Elaboración propia].

3.2 Modelo en Python

En este proyecto se ha elegido *Python* para como plataforma para realizar la simulación, junto con el complento *SimPy* que ofrece numerosas herramientas para simulación de eventos discretos. Ya que el objetivo del trabajo es el desarrollo no es el desarrollo del código de *Python* en sí, si no la simulación, en esta sección se presenta una descripción sobre como modelar algunos proceso relevantes, que se explorarán adaptados al caso estudiados en el siguiente capítulo.

3.2.1 *Python* y *SimPy*

En primer lugar, *Python* es un lenguaje de programación considerado de alto nivel y multiparadigma, usado ampliamente en distintas industrias, que incluyen desde aplicaciones científicas, simulación, hasta el desarrollo de aplicaciones o videojuegos. El motivo de la elección de este lenguaje de programación, además de por su popularidad y su relativa sencillez de aprendizaje, reside en que el software posee una licencia de código abierto, lo que permite que cualquiera pueda replicar este proyecto sin necesidad de comprar una licencia, como pasaría con *MATLAB* y su extensión de simulación *Simulink* u otros sofwares específicos de simulación de eventos discretos.

En *Python*, se incluyen funciones que ya aparecen en multitud de lenguajes, a continuación se presenta un resumen de funcionalidades básicas y relevantes para este proyecto:

- **Variables**. Las variables guardan información, ya sea texto, un entero, un número decimal, un valor lógico o un conjunto de los anteriores. Existen distintos tipos de variable según el tipo de dato que guarde, los que se usan en este proyecto son:
  - *Entero*, un número entero, se define como `A = 34` .
  - *Float*, numero en formato de coma flotante de doble precisión, permite almacenar decimales, se definen exactamente igual que los enteros pero deben tener presente algún decimal: `b = 3.00` .
  - *Booleano*, solo puede tomar valor verdadero o falso, se definen con las palabras reservadas correspondientes: `Var_verdadero = True; Var_falso = False` .
  - *Strings*, cadenas de caracteres, se definen de de la siguiente forma: `C = 'texto'`. Se usan para guardar texto como una serie de caracteres, sumar dos cadenas es equivalente a encadenarlas.

  - *Lista*, agrupa varios objetos de forma ordenada, que pueden ser de distintos tipos. Se define como `lista = [3.45, 'hola', False]`. Los elementos se indexan con un índice que indica su posición, comenzando con el 0, para acceder al objeto en la última posición se puede usar el índice -1. Los elementos pueden se a su vez otras listas, lo que permite crear una especie de matrices con listas anidadas.
  - *Diccionarios*, almacena elementos por pares, para acceder al valor del objeto se le especifica al diccionario su clave asociada, `dic = {'clave1': 'valor1', 'clave2':[1, 2.1, 'hola'], 'clave3': True}`. Como se ve se pueden introducir listas dentro de estos diccionarios, incluso otro diccionario, lo que lo convierte en una herramienta útil para almacenar cualidades o características de un objeto, o en el caso de este proyecto los tiempos y variables de estado de un pedido.

- **Sentencias condicionales**. Son instrucciones que especifican que parte del código debe ejecutarse cuando o mientras se cumplan ciertas condiciones.
  - Sentencia *if*. La sentencia `if condition: code` establece que el código que se escriba a continuación solo se puede ejecutar si se cumple una condición especificada. Se complementa con las sentencias `elif condition2: code 2` , que permite establecer otra sección ejecutable en el caso que no se cumpla la primera condición sólo si se cumple otra condición expresada, se pueden escribir tantos elif como sea necesario; y `else: code3` , que permite que se ejecute otro fragmento de código si no se cumplen ninguna de las condiciones anteriores. Se pueden anidar condiciones, introduciendo sentencias if dentro de otras:
  - Bucle *for*. Estos bucles permiten ejecutar un mismo bloque de código varias veces, recorriendo un objeto iterable (una lista o una cadena por ejemplo) y asignándole a una variable un valor del iterable cada vez, recorriéndolo de principio a fin. Por ejemplo:

```
>>> primos = [1, 2, 3, 5, 7, 11, 13]
>>> for num in primos: # En cada iteración num tomará un valor
    de primos
...     print(num),

1
2
3
5
7
11
13
```

  - Bucle *while*. Este tipo de bucle ejecuta un bloque de código sin parar, mientras se mantenga una condición especificada. Se puede crear un bucle infinito si la condición es verdadera siempre, o escribiendo `True` en la posición de la condición. Ejemplo en el que se escriben los números primos menores que 10:

```
>>> primos = [1, 2, 3, 5, 7, 11, 13], num_id = 0 # Se empieza
    con el primer elemento de la lista primos
>>> while primos[num_id] <= 10:
...     print(primos[num_id]),
...     num_id += 1 # Actualización del índice
```

```
1
2
3
5
7
```

- **Funciones**. Las funciones son bloques de código con un nombre asignado y se ejecutan solo cuando son llamadas en el código, a las que entran unos inputs y devuelve unos resultados. Se definen como `def nombre_funcion(input1,input2,...):` , dentro de la función se pueden leer las variables definidas fuera de ella, pero trabajará con variables a nivel local. Para poder devolver algún valor, necesita que al final aparezca esta línea `return output` . Un ejemplo que devuelve la suma de dos variables:

```
>>> def suma(x,y):
... return x+y
...
>>> suma(3,4)
7
>>> suma(10,2.4)
12.4
```

- **Métodos**. Los métodos son funciones internas asociadas a objeto. Permiten acceder a funciones propias u atributos no visibles a simple vista. Para acceder a ellos se escribe el nombre del objeto, seguido con un punto, el nombre del método y entre paréntesis los inputs que necesite el método, si es que es necesario. Uno muy usado es método *append* para listas, que permite añadir un elemento al final de la lista.

```
>>> saludos = ['hola', 'adios', 'que tal']
>>> saludos.append('buenas')
>>> print(saludos)
['hola', 'adios', 'que tal', 'buenas']
```

Una gran ventaja de *Python* es que, a pesar de que de base carece de muchas herramientas para aplicaciones mas específicas, la comunidad que posee detrás ha ido desarrollando módulos o paquetes complementarios para cada tareas. La sencillez de instalación y su variedad hacen que se puede usar para casi cualquier propósito, para su instalación, basta con que el paquete de datos se encuentre en la ruta de *Python* y escribir al principio del código `import paquete as paq` , donde paq es el nombre (normalmente abreviado) con el que se llamará al paquete en el código. En este trabajo se usarán varios, resumidos en tabla 3.1, pero el que permite la simulación de eventos discretos en *Python* es *SimPy*.

Esta extensión, establece un marco de trabajo para que *Python* pueda comprender como se relacionan los procesos, como avanza el tiempo y manejar las interrupciones. Con *SimPy* se definen los procesos como una función más en el código para modelar un cliente, una máquina, una cola o lo que se elija , pero se añade antes de correr la simulación un comando nuevo que le dice a *Python* como tratar esa función.

Para una simulación con *SimPy*, primero se deben definir los procesos involucrados mediante funciones, con sus entradas y salidas. En segundo lugar, crear un entorno de simulación de *SimPy* donde correr la simulación. A este entorno se le asignan los procesos, llamando a las funciones y asignando un nombre al proceso; esto permite utilizar una sola función para modelar varios procesos similares y despues crearlos en el entorno de simulación llamando a la misma función con varios

Tabla 3.1 Relación de módulos de Python usados.

Nombre	Código	Descripción
SimPy	`import simpy`	Eventos discretos.
Random	`import random`	Permite generar números aleatorios con numerosas distribuciones.
NumPy	`import numpy as np`	Incorpora unos arrays de datos en forma de matrices y vectores muy útiles además de herramientas para trabajar con ellos.
PyPlot	`import matplotlib.pyplot as plt`	Añade herramientas de representación de gráficos.
Pandas	`import pandas as pd`	Facilita la exportación de datos en formato .dot y Excel.
Time	`import time`	Permite usar contadores de tiempo para indicar el paso del tiempo en el código.

parámetros y nombres distintos. En este paso se definen también los eventos. Por ultimo se da la orden de simular el entorno creado. El ejemplo que presenta la documentación propia de *SimPy* [10], dos relojes simultáneos con ritmos distintos, es bastante esclarecedor:

Código 3.1 Ejemplo de código para simulación en Simpy, dos relojes asíncronos, de [10].

```
>>> import simpy
>>>
>>> def clock(env, name, tick):
...     while True:
...         print(name, env.now)
...         yield env.timeout(tick)
...
>>> env = simpy.Environment()
>>> env.process(clock(env, 'fast', 0.5))
<Process(clock) object at 0x...>
>>> env.process(clock(env, 'slow', 1))
<Process(clock) object at 0x...>
>>> env.run(until=2)
fast 0
slow 0
fast 0.5
slow 1
fast 1.0
fast 1.5
```

Nótese como el ejemplo utiliza un bucle sin fin con un `while True:` para que los relojes corran indefinidamente, sin esto el proceso definido en la función correría una sola vez; en estos casos, es importante darle al comando de inicio de simulación el parámetro del tiempo de la simulación, ya que sin él correría indefinidamente. Además, cabe destacar el uso de los métodos de *Python* para, primero definir el entorno de simulación, y segundo, añadirle características a este entorno. Por último, es sumamente importante que cada proceso (definido como una función en *Python*) tenga una duración, para eso se le incorpora una pausa con el comando `yield env.timeout(duration)`, incluyendo la duracion de este proceso dentro.

3.2.2 Procesos básicos

A continuación se presentan, en rasgos generales, como se van a modelar y codificar los distintos procesos involucrados en un sistema productivo. Se han identificado 4 procesos que pueden ser modelados de forma independiente que sean relevantes en este proyecto, el proceso de llegada y creación de pedidos nuevos, el proceso por el que se liberan los pedidos de forma ordenada desde el *preshop* a la planta, el proceso por el que las máquinas individuales manejan su cola y realizan las operaciones requeridas para dejarlas en el buffer de componentes; y por último el proceso de ensamblaje, que toma las piezas de los buffers y las monta según un pedido y lo entrega al cliente.

Llegada de pedidos

En primer lugar, es necesario modelar la forma en la que llegan los pedidos. Suponiendo que un pedido llega nada mas comenzar la simulación, el software desconoce qué características tiene este pedido, cuantas piezas, de qué tipo, qué fecha de entrega hay que cumplir, etc. El primer paso es generar los datos siguiendo las distribuciones aleatorias correspondientes, es el bloque hexagonal en la figura 3.4.

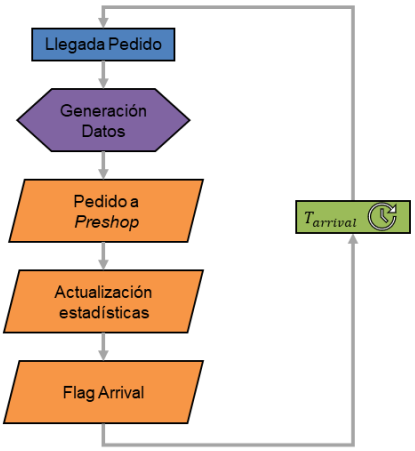

Figura 3.4 Esquema lógico del proceso de llegada de pedidos [Fuente: Elaboración propia].

Una vez generadas los atributos relevantes para la simulación de este pedido, este debe pasar a una lista de *preshop*, donde se registren todos los pedidos que han llegado, pero que aún no han iniciado su fabricación. Los pedidos se modelarán en el código como un diccionario que almacene cada uno de los atributos, esto permite añadir o sobrescribir atributos conforme el pedido avanza en la planta. La lista de preshop, a ojos de *Python*, se tratará como una lista que guarda de forma ordenada los diccionarios de los pedidos.

En este momento, se pueden actualizar las estadísticas que se estén midiendo sobre la planta, en relación a las llegadas. Además, se lanza un aviso (en programación se suele llamar *flag* a la variable que indica este tipo de eventos) a los demás procesos, de que se ha actualizado la lista de preshop; se verá mas adelante como hacen uso los distintos procesos de estas alertas.

Lo único que le falta al proceso es volver a empezar, una vez haya pasado un tiempo. Este tiempo se tendrá que modela con una distribución de probabilidad lo mas parecida a la realidad posible. A continuación se presenta un código que modelaría este proceso a rasgos generales.

Código 3.2 Código para un proceso genérico de llegada de pedidos.

```
def process_newOrder(env):
    # New order arrives, its data is generated and passed to the preshop
    global preshop, flagarrival
    global level_queue, t_level_queue
    while True:
        Tarrival = 4

        yield env.timeout(random.expovariate(1 / Tarrival)) # Time
            between new orders

        # Generation of parameters of the order
        Q_comp = random.randint(2, 6) # Number of different types of
            components required
        components = [1, 2, 3, 4, 5, 6]
        Qjk = np.zeros(6) # List with the quantity of the component j for
             order k
        for t in range(Q_comp):
            ty = random.choice(components)
            components.remove(ty)
            Qjk[ty - 1] = random.randint(1, 6) # Number of component of
                type j

        DDk = env.now + sum(Qjk) * random.randint(3, 5) # Deadline

        orderk = {'Qjk': Qjk, 'DDk': DDk, 'TO': env.now} # Dictionary of
             the order

        # Introduce the order in the preshop
        preshop.append(orderk) # The preshop is a list filled with
            orders waiting to be released

        # Queue performance Measurements
        level_queue.append(len(preshop) - 1) # Before addition
        t_level_queue.append(env.now)

        level_queue.append(len(preshop)) # After addition
        t_level_queue.append(env.now)

        flagarrival.succeed() # Flag event new order has arrived
        flagarrival = env.event() # Reset flag
```

Proceso de liberación a planta

El siguiente paso es manejar la *preshop* e ir liberando los pedidos poco a poco a la planta. Lo primero, como se aprecia en la figura 3.5 es comprobar que la lista de pedidos pendientes no se encuentre vacía, lo que produciría errores si se intenta ejecutar el código que sigue. En el caso de que se encuentre sin ningún pedido, se tendría que esperar a la alerta de que llegue un nuevo pedido (el aviso que se modelo anteriormente). Cuando ya se sabe que existen pedidos que liberar,

es necesario calcular los parámetros de la planta relevantes y compararlos con los umbrales de *Workload Control*.

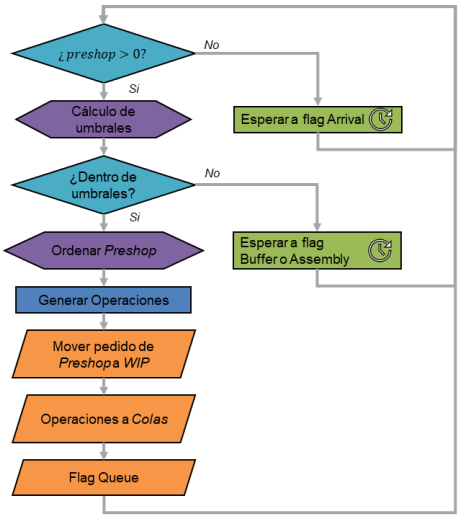

Figura 3.5 Esquema lógico del proceso de liberación de pedidos [Fuente: Elaboración propia].

El proceso vuelve a evaluar su situación, si se cumplen los límites establecidos, sigue adelante; en el caso contrario, debe esperar a que el buffer o la lista de pedidos *WIP* se desatasquen un poco. El proceso normal de liberación continúa ordenando la Preshop siguiendo la norma de despacho elegida, de este modo, se asegura elegir el pedido que se encuentre en lo mas alto de esa lista ordenada.

Con la lista ordenada y el pedido de mayor prioridad elegido, se pasa a generar tantas operaciones como componentes haya en el pedido, teniendo en cuenta el tipo y cantidad. Estas operaciones se modelan de forma similar los pedidos, en forma de diccionario con sus parámetros, la diferencia es que estas operaciones son las que se guardaran en las colas de las máquinas y los buffers, lo pedidos se quedan en un plano superior a la planta, pues no representan objetos en sí. Con esta filosofía, es necesario eliminar el pedido elegido del *preshop* y pasarlo a la lista de *Work In Process* junto con el resto de pedidos activos en la planta, las operaciones pasan a repartirse por las distintas máquinas disponibles, de forma indiferente , pues se esta considerando que cualquier máquina puede realizar cualquier tipo de operación.

El último paso necesario, es avisar a las máquinas que se han actualizado sus colas de espera. Un codigo de ejemplo se presenta a continuación:

Código 3.3 Código para un proceso genérico de liberación de pedidos.

```
def process_release(env):
    global preshop, WIP
    global flagqueue, flagbuffer, flagassembled
    while True:
        if len(preshop) > 0:
            A = len(WIP)
            B = max(max(Thrb-b),0)
            if A < Thr_wip and B: # First & Second workload policy
                respectively
                queuetorelease = [] # Generation of a list of operations
                    to release to the machines
```

```
                # Firstly, organise the Preshop
                preshop = sorted(preshop, key=lambda d: d['DDk'])
                order = preshop[0] # Selected order (higher priority
                    order)
                Qjk = order['Qjk']
                DDk = order['DDk']
                for component in range(len(Qjk)):
                    if Qjk[component] != 0:
                        for jj in range(int(Qjk[component])):
                            queuetorelease.append({'DDk': DDk, 'Comp':
                                component + 1}) # An operation is assigned
                                as a dict with properties

                # Moving orders from preshop to Work-In-Progress list
                WIP.append(preshop[0])
                del preshop[0]

                # Release of the generated operations
                for operation in queuetorelease:
                    mach = sorted(queue_mach, key=lambda d: len(queue_mach
                        [d]))[
                        0]  # Extract which machine has a shorter queue
                    queue_mach[mach].append(operation) # Operation is
                        introduced in shorter queue
                    flagqueue.succeed() # Queue update event flag
                    flagqueue = env.event() # Event Reset
            else:
                yield flagbuffer or flagassembled # Wait for either the
                    WIP reduction or the buffers update
        else:
            yield flagarrival
```

Proceso de fabricación en una máquina

Este proceso solo es necesario definirlo una vez, y después llamarlo tantas veces como máquinas existan en la planta. Desde el punto de vista de una máquina individual i, lo primero que necesita para comenzar a producir las operaciones, es saber si hay alguna operación pendiente en su cola. En caso negativo, debe esperar a la que se actualizen las colas (*flag queue*).

Cuando haya alguna operación, es necesario ordenar la cola según el criterio elegido previamente, se escoge la operación con mayor prioridad y se saca de la cola. En este momento, se comienza procesar la pieza, poniendo el proceso en pausa el tiempo correspondiente (generado aleatoriamente generalmente).

Tras terminar el procesado de la operación, se actualizan los atributos de la operación, se le añade la marca de tiempo en el que termina su procesado. Las operaciones terminadas se van mandando la lista de buffer correspondiente al tipo de componente, lanzando el aviso correspondiente de que se ha actualizado el buffer. Véase el código de ejemplo, donde se puede apreciar que los buffer se han modelado como listas que van almacenando los diccionarios de las operaciones, estos buffers no se ordenarán, pues cada tipo de componente es igual, independientemente del pedido para el que se haya fabricado.

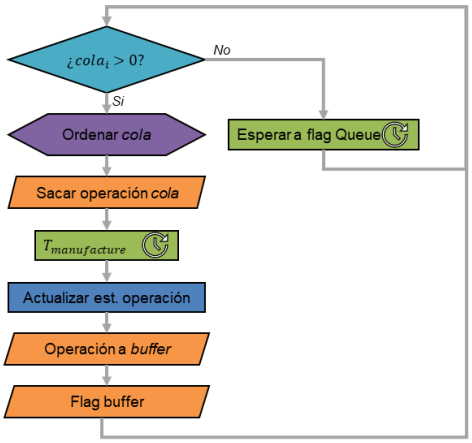

Figura 3.6 Esquema lógico del proceso de procesamiento de operaciones en una máquina genérica [Fuente: Elaboración propia].

Código 3.4 Código para un proceso genérico de procesamiento en una máquina.

```
def process_manufacture(env, mach):
    # "mach" acts as an identifier of the machine
    global queue_mach, buffers, WIP
    global flagqueue, flagbuffer
    while True:
        if len(queue_mach['machine ' + str(mach)]) > 0:
            # First the machine queue is ordered
            queue_mach['machine ' + str(mach)] = sorted(queue_mach['
                machine ' + str(mach)], key=lambda d: d['DDk'])
            inprocess = queue_mach['machine ' + str(mach)].pop(0) # The
                first operation in queue is picked

            # Manufacturing start
            inprocess['T_mach'] = env.now
            yield env.timeout(random.expovariate(1)) # Manufacturing time

            # Send order to buffer
            comp = inprocess['Comp']
            inprocess['T_buff'] = env.now
            buffers['Buffer ' + str(comp)].append(inprocess)

            flagbuffer.succeed() # Buffer update flag
            flagbuffer = env.event() # Flag reset
        else:
            yield flagqueue # Wait for the machine queue to update
```

Proceso de montaje

El proceso final comienza comprobando que existen pedidos pendiente por ser montados, esto es equivalente a evaluar si algún pedido en la lista de *Work In Process*, en caso contrario sería necesario

esperar a que entre un nuevo pedido en proceso, esto es que sus operaciones hayan sido incluidas en las colas de las máquinas (o esperar a *flag qeue*).

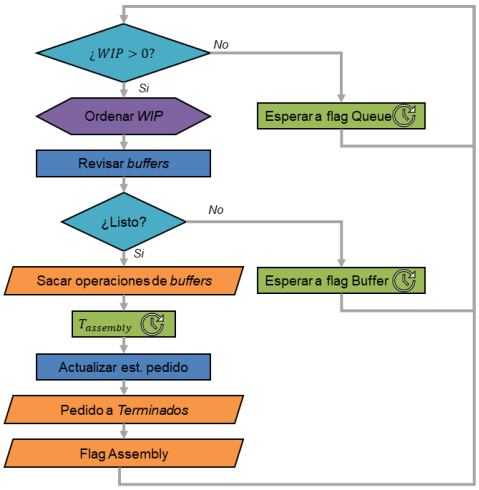

Figura 3.7 Esquema lógico del proceso de montaje de pedidos [Fuente: Elaboración propia].

Al igual que en los procesos anteriores, es necesario ordenar la lista de *WIP* y elegir el pedido de mayor prioridad siguiendo la regla elegida. Con ese pedido en la mano, es el momento de comprobar que los buffers contienen las cantidades de componentes necesarias para el montaje final del pedido. Si este no fuese el caso, habría esperar a que los buffers se actualizen (*flag buffer*).

Suponiendo que se dan las condiciones necesarias para el montaje del pedido, es necesario sacar las operaciones seleccionadas de los buffers; se escojen independientemente del pedido para el que habían sido generadas originalmente, pues todos los componentes de un mismo tipo son exactamente iguales, de este modo, si llega un pedido con una prioridad muy alta, saldrá en cuanto se tengan los componentes suficientes, no hay que esperar a que se terminen los componentes de su pedido en concreto (estos se usarán para ensamblar otro pedido).

Con todos los componentes preparados, el proceso se pone en pausa el tiempo que tarde en ensamblarse el pedido, de manera similar al proceso de la máquina. Por último, antes de de dar la señal de pedido terminado y mandar el pedido a la lista de pedidos acabados, es necesario actualziar las estadísticas del pedido, con los tiempo de montaje y fin.

Código 3.5 Código para un proceso genérico de una estación de montaje con buffers de los componentes.

```
def process_assembly(env):
    global buffers, WIP, finished
    global flagbuffer, flagassembled, flagqueue
    while True:
        if len(WIP) > 0:
            # Firstly, organise the WIP
            WIP = sorted(WIP, key=lambda d: d['DDk'])
            inprocess = WIP[0] # Pedido seleccionado
            ready = True
            for j in range(len(buffers)):
                # Check if buffer j has enough components for the order k
                    (bj > Qjk)
```

```
                if len(buffers['Buffer ' + str(j + 1)]) < inprocess['Qjk'
                    ][j]:
                    ready = False

            if ready:
                inprocess['T_ass'] = env.now # The moment the order
                    enters assembly
                operations = []
                for j in range(len(buffers)):
                    # Extract the operations needed from buffer j
                    operationj = buffers['Buffer ' + str(j + 1)][:int(
                        inprocess['Qjk'][j])]
                    del buffers['Buffer ' + str(j + 1)][:int(inprocess['
                        Qjk'][j])]
                    # This operations are added to a list with the used
                        operations
                    operations.append(operationj)

                # Add the used operations to the finished order
                inprocess['operations'] = operations
                yield env.timeout(random.expovariate(1)) # Assembly time
                inprocess['T_out'] = env.now

                # Write the order in the finished list
                finished.append(inprocess)
                del WIP[0]

                flagassembled.succeed() # Finished order event flag
                flagassembled = env.event() # Reset flag

            else:
                # Wait for buffer update
                yield flagbuffer
        else:
            yield flagqueue # Wait for new order in the WIP
```

4 Modelo de Simulación

En este capitulo se entrará en una explicación mas detallada del modelo del artículo que se ha tomado como punto de partida y se concretarán las condiciones de la simulación, así como las particularidades de los procesos y su codificación.

4.1 Contexto inicial

En este proyecto, se toma el mismo modelo de taller que plantean *Renna et al.* [1] en su estudio de principios de año. Su objetivo inicial consiste en buscar un criterio de toma de decisiones simple y sencillo, que permita mejorar tanto el rendimiento como la sostenibilidad de un taller *(job-shop)*, reduciendo los niveles medios de los almacenes y manteniendo la productividad. Sin embargo, el escenario simulado no es una réplica exacta del modelo que ellos proponen en su artículo, ya que se han realizado distintas suposiciones cambiando algunos parámetros, que no se aclaran en el artículo, o diferencias que han surgido tras la adaptación del modelo a *Python*.

Era importante para los autores, que el criterio elegido sea aplicable a la logística diaria del taller, esto es, que sea sencillo pero efectivo. Existen complejas funciones y reglas de despacho que optimizan con resultados muy positivos, pero vienen ligados a licencias de programas muy caras o con un coste temporal elevado que hace que sea imposible llevarlo a la practica en el día a día.

En su trabajo, establecen que la razones por las que un taller escogería el uso de las políticas de control de carga de trabajo, evitar trabajar en pedidos cuya fecha de entrega es muy lejana, reducir la carga de trabajo y los costes de inventario, además de limitar de alguna forma el inventario de componentes en proceso.

4.1.1 Modelo de *job-shop*

Los autores definen un *job-shop*, o un taller, al que llegan pedidos con diferentes componentes, que pueden ser fabricados por cualquiera de las 6 máquinas presentes. Antes de que los pedidos se "lancen" a las estaciones de fabricación, estos se quedan almacenados en una lista de ordenes llamada *preshop*, de forma que cuando se cumplan ciertas condiciones de trabajo se vayan liberando los pedidos y se comiencen a fabricar sus respectivos componentes.

Es decir, nos encontramos ante un sistema que podría parecer *Make to Order*. Dado que los clientes eligen el numero de cada tipo de componentes del producto final, se trata de un producto muy personalizado, lo que se ve beneficiado por este tipo de estrategia frente a una estrategia *Make to Stock*, donde se acumularía un *stock* de productos acabados de diversas configuraciones absurdo hasta alcanzar el que pide el cliente.

Una vez se han fabricado todos los componentes del pedido, se montan en una estación de ensamblaje independiente, véase la figura 4.1.

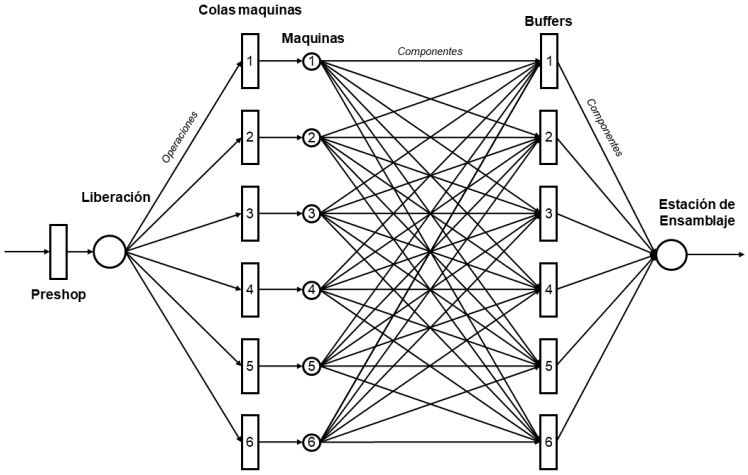

Figura 4.1 Esquemático de todas las entidades que intervienen en el proceso [Fuente: Elaboración propia].

La llegada de los pedidos, sigue una distribución exponencial con media $\mu_{llegada}$ unidades de tiempo, es decir con parámetro $\lambda = 1/\mu_{llegada}$:

$$t_{arrival} = Exp\left(\lambda = 1/\mu_{llegada}\right)$$

Estos pedidos, son generados de forma aleatoria (en *Python* se usará el paquete *random*). El proceso pasa por, en primer lugar definir el número de componentes distintos que va a tener, después cuáles serán esos componentes, y finalmente, cuantos componentes de cada tipo seleccionado. Se sigue la siguiente lógica:

1. Número de tipos de componentes distintos: $Q_{comp} = U\,[2,6]$
2. Selección aleatoria de los tipos de componentes de los que se necesitan piezas. Por ejemplo: $Q_{comp} = 3 \rightarrow j = \{1,2,3,4,5,6\} \rightarrow j = \{1,3,4\}$
3. Generación aleatoria del vector Q_j^k con el número de componentes de cada tipo seleccionado, conforme a la distribución de cada tipo de componente[1]. Por ejemplo: $j = \{1,3,4\} \rightarrow Q_1^k = U[1,5];\ Q_3^k = U[4,6];\ Q_4^k = U[8,10] \rightarrow Q_j^k = \{2,0,6,8,0,0\}$

Cuando el vector se ha generado, se establece la fecha de entrega del pedido DD_k, siguiendo la siguiente ley:

$$DD_k = T_k + M \cdot N_{op}; \quad M = U[3,5]; \quad N_{op} = \sum Q_j^k$$

Donde T_k es el instante en el que el pedido llega al taller, M una variable aleatoria y N_{op} el total de los componentes (u operaciones) a fabricar en el pedido. La fecha de entrega, junto con el vector Q_j^k son suficientes para definir el pedido, que pasa a la lista de *preshop*, antes de ser liberada a las estaciones de fabricación cuando se cumplan las condiciones que se explican en el siguiente apartado. Los componentes del pedido, pasarían como *operaciones* que van definidas por su fecha de entrega y su tipo de componente, no haría falta indicar a que pedido pertenecen, pues todos los componentes del mismo tipo son considerados idénticos por [1].

En el caso que las operaciones (cada uno de los componentes de un pedido) pasen de la *preshop* a la planta de fabricación, es necesario asignar cada una a la cola de cada maquina. Este reparto se

[1] El número de componentes de cada tipo sigue una distribución uniforme con límites preestablecidos

realizará de forma secuencial, buscando la máquina con la cola de espera mas corta y asignándole una operación, después se repite el proceso con la siguiente operación, hasta que se vuelvan a incumplir las reglas de control del *Workload*.

Las distintas maquinas o estaciones, irán completando las operaciones progresivamente, el tiempo de fabricación sigue una ley exponencial similar a la del tiempo entre llegadas de pedidos, pero con media fija e igual a la unidad:

$$t_{manufacturing} = Exp(\lambda = 1)$$

La pieza terminada se pasa a su *buffer* correspondiente y se almacena hasta que haya suficientes piezas para completar un pedido en la estación de ensamblaje. Es importante repetir que cualquier máquina puede realizar la operación correspondiente a cualquier tipo de componente y, por tanto, cada máquina puede mandar el pedido a cualquier *buffer*, como se ve en el esquema de relaciones de la figura 4.1.

Por último, la estación de ensamblaje, monitoriza el estado de los almacenes, comprobando la lista de pedidos ordenadas según alguno de los criterios que se verán mas adelante, para que cuando se tengan los componentes necesarios para completar el pedido que se encuentre en primer lugar en la cola, ensamblarlo y entregarlo al cliente. El tiempo de ensamblaje, viene dado por la misma expresión que el tiempo de fabricación de un componente, y es independiente del número de componentes a ensamblar:

$$t_{assembly} = Exp(\lambda = 1)$$

4.1.2 Estrategias del estudio original

Los autores del estudio en el que se basa este proyecto decidieron comparar dos estrategias de *Workload Control* con dos mecanismos simples de ordenacion de pedidos sin control de liberación alguno.

- *Referencia 1*. La primera estrategia sirve como primer punto de referencia, para las estrategias de control de carga de trabajo. Esta no presenta ningún control sobre la liberación, las ordenes se van liberando conforme llegan y además, las colas se ordenan siguiendo la regla de despacho FIFO.
- *Referencia 2*. Otro punto de referencia, exactamente igual que el anterior, pero ordenando las colas con la regla EDD.
- *Estrategia 1, umbrales fijos*. Se fijan dos condiciones de control del *Workload* para que se puedan liberar los pedidos. La primera ya se ha visto anteriormente, limitar directamente respecto al número de pedidos en los que se está trabajando al mismo tiempo.

  $$WIP <= Thr_{WIP} \tag{4.1}$$

  La segunda condición, hace referencia al nivel de los buffers, con el objetivo de aliviar los costes por inventario y que estos no crezcan, limita a que no se liberen operaciones si en todos los buffers hay mas de 6 componentes esperando, pues es el máximo número de operaciones de un componente que puede haber en un pedido (según el primer escenario, véase el apartado siguiente). De este modo para que se liberen debe cumplirse lo siguiente además de la condición 4.1:

  $$f(b_j) = \text{máx}\left\{ \max_j \left(Thrb - buffer_j\right); 0 \right\} > 0, \quad Thrb = 6 \tag{4.2}$$

Donde b_j indica el nivel del buffer del componente j. En el caso que sea mayor que cero se cumple que aún hay hueco en algún buffer.

- *Estrategia 2, umbrales adaptables*. Una variación con respecto al caso anterior, se mantiene la primera condición de liberación de 4.1. Sin embargo, la segunda se modifica, y se incluye que el umbral de los buffers sea adaptable a cada tipo de componente conforme avanza la simulación. La expresión de control es la misma, pero con el nuevo umbral que se calcula en función de la media y la varianza del número de componentes de ese tipo que han ido llegando.

$$f(b_j) = \text{máx}\left\{\max_j \left(Thrb_j - buffer_j\right);\, 0\right\} > 0, \quad Thrb_j = 6 \tag{4.3}$$

$$Thrb_J = average_j + K * variance_j \tag{4.4}$$

Donde K es un parámetro de ajuste, que ajusta la probabilidad de poder almacenar todos los componentes de un pedido. Un valor alto de K aumenta la probabilidad de almacenar todos los componentes de un pedido, pero aumentaría los costes por inventario.

4.1.3 Escenarios de simulación

En el artículo original, para estudiar las distintas estrategias, se plantean cuatro escenarios para ver como reacciona cada una de ellas bajo distintas condiciones de demanda y configuración de pedidos.

Estos cuatro escenarios se van a mantener en este estudio, son los siguientes:

- *Escenario base*. Es el caso mas sencillo, los pedidos llegan con la distribución exponencial antes vista y con media $\mu_{llegada} = 4$. El número de componentes distintos mantiene la distribución uniforme U [2,6] y el número de operaciones por componente una uniforme U [1,6]. El tiempo de ensamblaje sigue una distribución exponencial de media uno y la fecha de entrega se calcula mediante la expresión antes vista.
- *Demanda variable*. En este escenario se introduce una serie de fluctuaciones en la demanda de pedidos, parámetro λ (y por tanto la media) de la distribución exponencial deja de ser constante, y va variando cada 200 unidades de tiempo, siguiendo la siguiente tabla:

Tabla 4.1 Fluctuación de la demanda.

Tiempo	Media, $\mu_{llegada}$	Parámetro λ
0	4	0.25
200	6	2/3
400	2	0.5
600	5	0.2
800	4	0.25

- *Componentes desiguales*. Ahora, varía la distribución del número de operaciones de un tipo concreto dependiendo del tipo de componentes que sea, la distribución no es la misma para cada componente, pero se mantiene la distribución que da el número distintos de componentes. Para cada componente la probabilidad sigue las siguientes distribuciones:
- *Combinación de escenarios 2 y 3*. Este último caso es el mas complejo de los cuatro, pues combina la demanda variable del segundo caso y las distribuciones de operaciones específicas para cada tipo de componente.

Tabla 4.2 Demanda de operaciones específica de cada componente.

Componente	Distribución
1	$U\,[1,5]$
2	$U\,[1,3]$
3	$U\,[4,6]$
4	$U\,[8,10]$
5	$U\,[1,10]$
6	$U\,[7,10]$

En los casos con demanda variable, basta con introducir una serie de "bucles if" que definan el valor del tiempo medio de espera dependiendo del tramo de la simulación. Para implementar la distinta probabilidad del numero de operaciones por componente, es mas complejo, pero es posible crear una lista que en cada elemento albergue los dos extremos de la distribución uniforme de cada componente. De este modo, la definición de un pedido nuevo, junto con la espera a su llegada y la definición del número de operaciones para componentes vendría dado por el siguiente código:

Código 4.1 Codificación de la definición de un pedido y el tiempo de espera genérica para cualquiera de los escenarios estudiados.

```
Qprob = [[1, 5], [1, 3], [4, 6], [8, 10], [1, 10], [7, 10]] # Uniform
    distribution limits for each component

Tarrival = 4 # 'Type_Demand == 'Constant'
if Type_Demand == 'Variable':
    instant = env.now
    if 200 <= instant < 400:
        Tarrival = 6
    elif instant < 600:
        Tarrival = 2
    elif instant < 800:
        Tarrival = 5

yield env.timeout(random.expovariate(1 / Tarrival)) # Time between new
    orders

Q_comp = random.randint(2, 6) # Number of different types of components
    required
components = [1, 2, 3, 4, 5, 6]
Qjk = np.zeros(6) # List with the quantity of the component j for order
    k
for t in range(Q_comp):
    ty = random.choice(components)
    components.remove(ty)
    if Type_Comp == 'Different':
        Qjk[ty - 1] = random.randint(Qprob[ty - 1][0], Qprob[ty - 1][1])
            # Number of component of type j
    else: # Type_Comp == 'Equal'
        Qjk[ty - 1] = random.randint(1, 6) # Number of component of type
            j
```

```
DDk = env.now + sum(Qjk) * random.randint(3, 5) # Deadline

orderk = {'Qjk': Qjk, 'DDk': DDk, 'TO': env.now}
```

4.2 Diseño Experimental

El siguiente apartado define el experimento que se va a realizar en este estudio, se establecerán los parámetros fijados de simulación y que criterio se ha elegido para comparar con el planteado en el estudio original.

4.2.1 Ratio crítico

En el trabajo de *Renna et al.* [1], se comparan varias estrategias de gestión de carga de trabajo. Las conclusiones del estudio determinan que ambas estrategias de *Workload Control* mejoraban en cuanto a sostenibilidad al tener menor nivel de utilización y reducían los inventarios de ensamblaje, con la contraprestación de ser algo menos eficientes en cuanto al ritmo de producción.

En todo momento en su estudio se impone la ordenación de las colas internas mediante la regla de *Earliest Due Date* (fecha de entrega mas cercana), en este proyecto se comparará su modelo frente al mismo con ordenación de todas las colas bajo el criterio de del *Ratio Crítico*. Esta comparación entre EDD y Ratio Crítico, para un modelo de *jobshop* con estas características, es novedosa, no se ha realizado antes; y se puede entender como una ampliación del artículo original. El interés surge de intentar encontrar una mejora al modelo existente, variando una pequeña parte del mismo.

Como se vio anteriormente, el criterio del Ratio Crítico compara el tiempo restante hasta la entrega con el tiempo estimado de fabricación. El cálculo es el siguiente, como se vio en la sección 2.3, incluyendo las variables propias de este modelo:

$$CR_k = \frac{DD_k - t}{\hat{t}_{process_k}} = \frac{DD_k - t}{\left(\sum_j Q_j^k\right) * \bar{t}_{manufacture} + \bar{t}_{assembly}} \tag{4.5}$$

El tiempo estimado $\hat{t}_{process_k}$ se puede calcular a partir de las medias de las distribuciones de fabricación de cada componente y ensamblaje ($\bar{t}_{manufacture}$ y $\bar{t}_{assembly}$), lo que finalmente dependería del número total de componentes, es decir la suma de Q_j^k. En el caso de las colas de las máquinas, para ordenar las operaciones, el tiempo estimado es la media de tiempo de fabricación de un componente. En los buffers no hace falta ordenar las operaciones.

Lo que se pretende al introducir esta regla de despacho es intentar darle prioridad a los pedidos y operaciones que llevan mas prisa con respecto al tiempo de fabricación asociado, y no solo porque su fecha de entrega esté cercana.

4.2.2 Características del estudio

En el estudio se replicarán los 4 escenarios planteados en el artículo original, tanto para la estrategia de umbrales constantes como la de umbrales adaptables. Primero se tomará como criterio de ordenación de colas el *EDD* y se repetirá para el Ratio Crítico, teniendo en total 16 configuraciones distintas que simular (4 escenarios por 2 estrategias por 2 reglas de despacho). Cada una de estas configuraciones se simulará 1000 veces a lo largo de un horizonte de simulación $t_{sim} = 1000$ unidades de tiempo.

Las variables que se medirán, son las propuestas por *Renna et al.*[1], explicadas la sección 2.4, los resultados y conclusiones se presentan en el siguiente capítulo.

El trabajo original deja algunos parámetros sin aclarar, para este estudio se han tomado valores considerados razonables, entre los que se encuentran:

- *Número de simulaciones,* n_{max}. Es el número de simulaciones por caso, esto es escenario, estrategia y regla de despacho, distinto. Se tomará igual mil iteraciones por caso, para asegurar un mínimo de convergencia de resultados.
- *Horizonte de simulación* ,t_{sim}. Las unidades de tiempo que durará cada simulación individual, igual a mil unidades de tiempo, tal y como se establece en el artículo estudiado.
- *Umbral de* WIP*,* Thr_{WIP} Se toma de forma arbitraria igual a 4, ya que si la frecuencia de llegada es de un pedido cada 4 unidades de tiempo y el numero de operaciones por pedido medio es 14 unidades de tiempo[2], contando el tiempo medio de fabricación de cada operación mas el tiempo de ensamblaje, sale una media de tiempo de procesado de 15 unidades de tiempo. Dividiendo la frecuencia de llegada entre el tiempo de procesado medio, da que de media, habrá 3.75 pedidos en proceso al mismo tiempo en la planta, con un umbral de 4 la planta es capaz de trabajar bien en un funcionamiento normal.
- *Parámetro de ajuste,* K. En el documento estudiado, se hacen simulaciones para tres valores de este parámetro $K = 1, 1.5, 3$; siendo el mejor resultado para $K = 3$ ya que, como se explica en el propio estudio, se llega a cubrir el almacenaje de los componentes en el 99.73 % de los casos, suponiendo una distribución normal. Se tomará este valor en la simulación de este estudio.

Tabla 4.3 Parámetros de simulación.

Parámetro	Símbolo	Valor
Número de simulaciones	n_{max}	1 000
Horizonte de simulación	t_{sim}	1 000 $u.d.$
Umbral de *WIP*	Thr_{WIP}	4
Parámetro de ajuste de umbral	K	3

[2] Teniendo como distribución de distintos componentes U [2,6] y la de numero de componentes de cada tipo U [1,6], teniendo en cuenta que son distribuciones independientes, la media de componentes totales es el producto de las medias, es decir, 14 unidades de tiempo

5 Resultados

A continuación, se van a comentar los resultados de las variables interés descritas en el apartado de Mediciones Interesantes (2.4). Se presentan los resultados para cada configuración del experimento. Se presentarán gráficos de barras para cada uno de los cuatro escenarios estudiados, para cada uno de ellos se mostrarán los resultados de cada estrategia (Trh_b constante frente a Thr_b adaptable).

Para cada estrategia se representará el nivel medio de la variable estudiada, representando en azul los resultados obtenidos usando la regla EDD como criterio de ordenación de colas y en naranja los correspondientes al ratio crítico.

5.1 Pedidos completados

En la figura 5.1 se puede observar como se obtienen resultados similares para para cada escenario independientemente del criterio y estrategia. En primer lugar, se observa que el criterio de ordenación del ratio crítico no funciona tan bien como el EDD desde el punto de vista de la producción bruta.

La diferencia máxima entre los dos criterios se produce en el escenario 4, el único caso donde el Ratio Crítico supone una ventaja, en concreto, usar el ratio crítico bajo estas condiciones supone completar un 1,79 % mas de media (un 1,83 % si sólo se tiene en cuenta la estrategia 1) que usando EDD. La diferencia mínima se encuentra en el segundo caso, donde las diferencias no son tan grandes, siendo el criterio del ratio un 0,20 % peor de media (0.11 % peor para el *APP1* y un 0.28 % en el *APP2*).

En términos generales de producción, elegir el ratio crítico supone producir 0,19 % menos si se escoge la estrategia de umbrales constantes, y un 0,17 % menos con la estrategia de umbrales variables; que si se usasen las mismas estrategias con el criterio de fecha de entrega mas temprana.

5.2 Retraso

A diferencia del caso anterior, observando los resultados brutos del retraso medio en la figura 5.2, es fácil darse cuenta que en los tres primeros casos, el retraso es negativo, lo que indica que las entregas se realizan antes de tiempo. Es en estos casos donde el criterio EDD parece superior al del Ratio Crítico, siendo la diferencia mayor en el caso de demanda variable (caso 2). En el caso base la diferencia es mínima (menos de un 2,30 % con respecto al EDD para ambas estrategias).

Sin embargo, en el caso 4, el retraso medio se vuelve positivo, es decir los pedidos comienzan a llegar tarde. Pero es mas interesante notar que es aquí donde el ratio crítico ha funcionado mejor en términos de retrasos, pues presenta un retraso medio menor que las simulaciones con EDD. De hecho es el caso donde la diferencia absoluta entre ambos criterios es mayor, las simulaciones en las que se ha usado el ratio crítico muestran un retraso de entorno al 12 % menor que en las EDD,

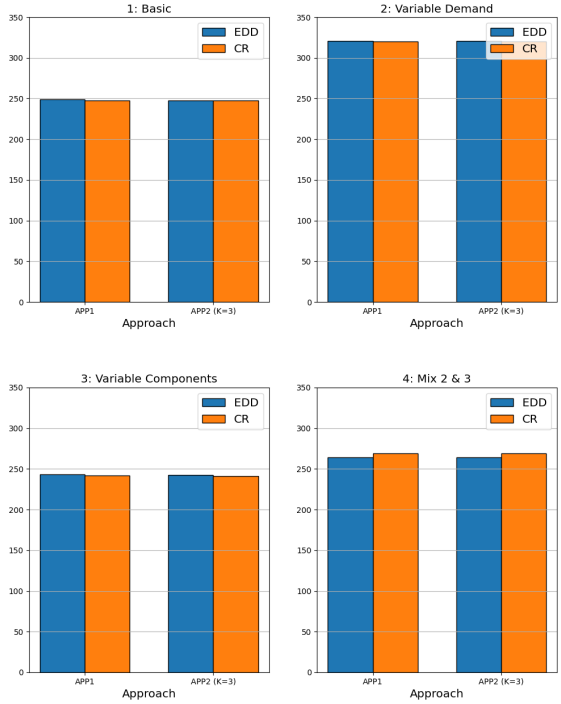

Figura 5.1 Comparativa de los pedidos medios completados en cada configuración [Fuente: Elaboración propia].

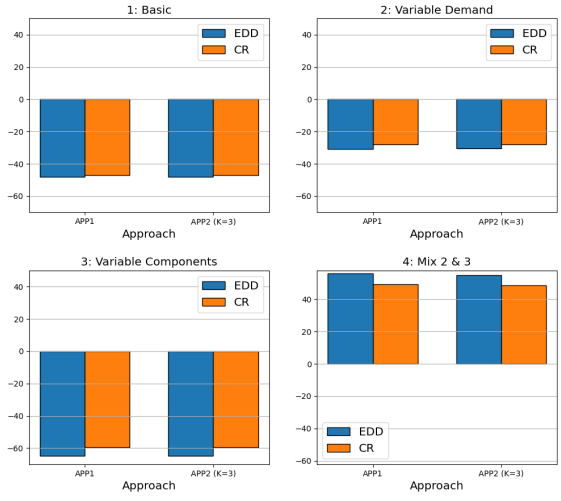

Figura 5.2 Comparativa del retraso medio de los pedidos en cada configuración [Fuente: Elaboración propia].

siendo una pizca mejor aquellas que además presentaban la estrategia 2 (11,74 % frente al 11,69 % de la estrategia 1).

Esto podría deberse a que el sistema se satura en términos de capacidad productiva en esta

configuración, la mezcla de alta demanda y que con los componentes diferenciados se tiene un número medio de componentes por pedido mas alto, consigue que el sistema no sea capaz de producir y terminar los encargos a tiempo. Encontrar la verdadera razón a este comportamiento se escapa al alcance de este proyecto y se deja pendiente de estudio su exploración.

5.3 Tiempo de procesado

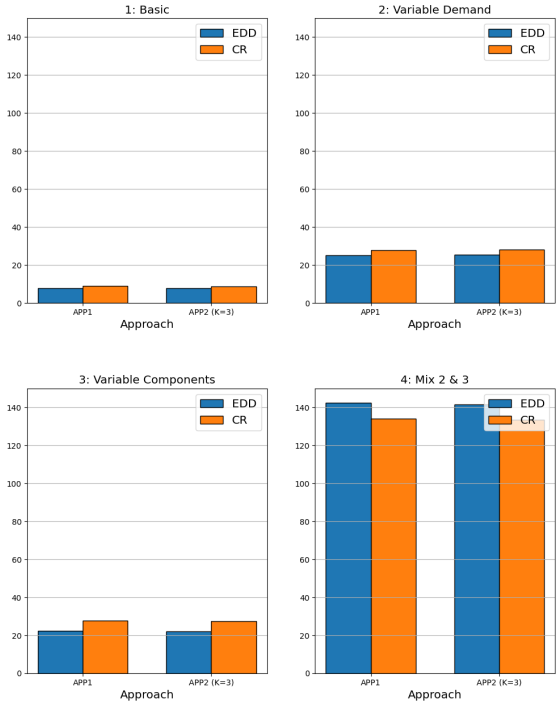

Figura 5.3 Comparativa del tiempo medio de procesado en cada configuración [Fuente: Elaboración propia].

Al igual que con el retraso medio, con el tiempo de procesado medio el criterio EDD es superior al del Ratio Crítico en todos los escenarios salvo en el último (ver figura 5.3). Se aprecia también que ambas estrategias son similares en términos de resultados, independientemente del criterio elegido, dicho esto, en los tres primeros casos, en promedio usar el ratio critico supone tiempos de procesado entorno a un 16 % mas largos que usando EDD, sin embargo, en el último escenario es un 5,59 % mejor que el EDD.

Volviendo a relacionarlo con el retraso, se ve como en el caso 4 el tiempo de procesado aumenta mucho, lo que puede ser la causa de que el retraso se vuelva positivo en este escenario. El aumento del tiempo de procesado parece estar directamente ligado a la complejidad del escenario, pues en el caso 2 y 3 presenta niveles similares, pero es en la combinación de ambos donde mas aumenta este tiempo.

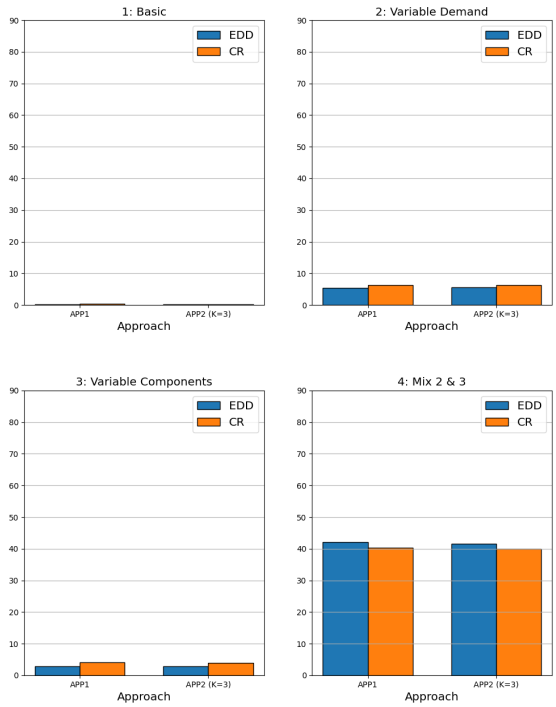

Figura 5.4 Comparativa de los pedidos en cola medios en cada configuración [Fuente: Elaboración propia].

5.4 Pedidos en cola

Para esta medida, existe una correlación similar al tiempo de procesado con la complejidad del escenario. Tener la escala misma escala en la figura 5.4 no ayuda a la visualización de los primeros escenarios, por ello se presenta la siguiente tabla de resultados, que también puede verse en el Apéndice B.

Tabla 5.1 Resultados numéricos del nivel medio de los pedidos en cola, para las distintas configuraciones.

Estrategia	Criterio	Escenario			
		1	2	3	4
APP1	EDD	0,17	5,48	2,91	42,14
	CR	0,30	6,27	4,07	40,37
APP2	EDD	0,17	5,60	2,86	41,67
	CR	0,29	6,36	3,99	39,88

Como se ve las colas medias se hacen mas y mas largas cada vez. Se puede comprobar que para el primer escenario no se suelen acumular mas de 3 pedidos y que la cola vuelve a caer a cero con frecuencia. Sin embargo, para los casos dos y tres, las colas comienzan a crecer, apareciendo "montañas" de pedidos en cola pero que vuelven a bajar a niveles mas bajos. En el último escenario se da un caso extremo, y es que la cola no para de crecer, indefinidamente, salvo en contadas ocasiones; esto sucede porque el sistema no es capaz de producir al ritmo de la demanda, es por

esto por lo que el retraso se vuelve positivo en este caso.

Comentando los resultados, se puede ver como ocurre algo similar al las variables comentadas con anterioridad; en todos los casos salvo el último el criterio EDD es algo superior (menores colas) que el Ratio Crítico. En concreto, la mayor diferencia se da en el primer caso, llegando a colas un 75,25 % mas largas de media en usando el CR frente al EDD (se da para el caso del escenario 1 con estrategia APP1). Para el último escenario, ambas estrategias dan resultados similares, pero aquellas en la que se utiliza el criterio de la razón crítica suponen unas colas un 4,25 % menores de media entre la estrategia 1 y 2.

5.5 Pedidos en proceso

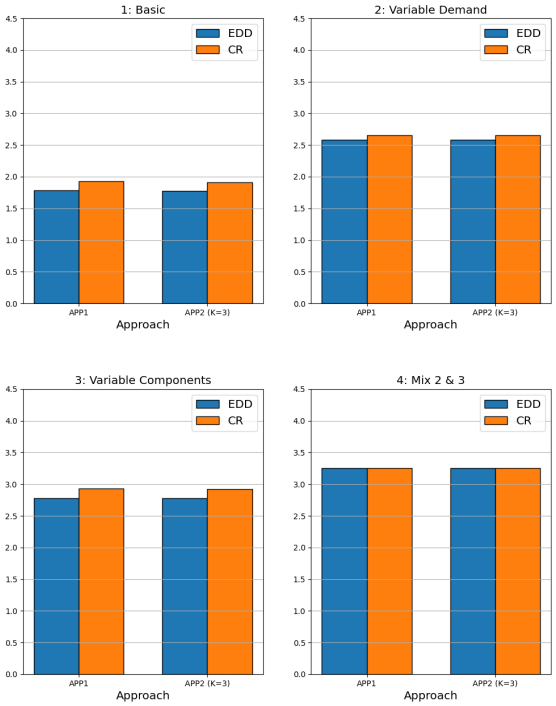

Figura 5.5 Comparativa de los niveles de WIP medios en cada configuración [Fuente: Elaboración propia].

Se presentan unos resultados muy similares para cualquier caso y configuración del taller, las diferencias son mínimas entre las distintas combinaciones de criterios y estrategias. Se puede ver claramente como nunca se sobre pasa el límite impuesto por el criterio de *Workload Control* de $Thr_{WIP} = 4$, que ocurriera esto significaría que algo no funciona en el modelo.

Es interesante destacar que los mayores niveles de WIP se dan en el caso 4, independientemente del criterio y la estrategia, estando todos muy cercanos entre ellos (todos se aproximan a la centésima a 2,25). Los escenarios 2 y 3 son muy similares, entorno a 2,5 y 3 pedidos en proceso de media, respectivamente; mientras que en el primer caso, el numero medio de pedidos en proceso es significativamente menor que en el resto, no llega a 2 pedidos de media.

No existen diferencias significativas entre una u otra estrategia, sin embargo, con los criterios se mantiene la tendencia que se ha visto con otras mediciones; la carga de trabajo es algo mayor para el caso en los que se usa la razón crítica (entorno a un 5 % mayor), pero en el cuarto caso se invierten, aunque por muy poco, los resultados consiguiendo un menor nivel de WIP (o número de pedidos en proceso) con el ratio crítico (la diferencia media entre ambas estrategias no llega a 0,2 %).

5.6 Utilización

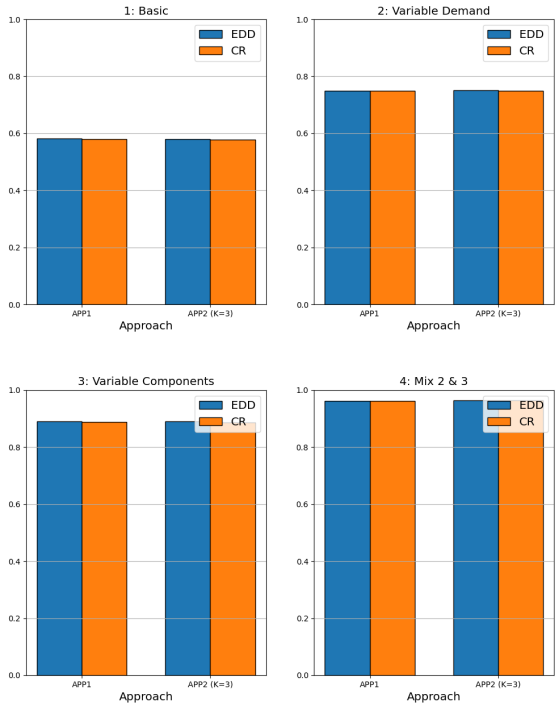

Figura 5.6 Comparativa de la utilización media en cada configuración [Fuente: Elaboración propia].

Viendo estos resultados, la utilización parece independiente del criterio y estrategia utilizada. componentes, haciendo que la planta tenga que procesar mas operaciones para cada pedido. Lo que si se puede apreciar es la progresión ascendente de la utilización conforme avanza la complejidad del escenario, se pasa de un 0,58 de media total en el primer escenario a un 0,96 en la final. Los resultados son tan parejos que habría que mirar la milésima en numerosos casos, lo cual se escapa de la precisión establecida para este trabajo.

En ninguno de los casos la utilización supera la unidad, lo que supondría trabajar por encima de la capacidad del taller, el máximo se queda cerca, 0,96 para el caso 4 con el criterio CR (ambas estrategias han conseguido el mismo valor junto con el EDD con estrategia APP2); el mínimo aparece en el escenario 1 con el criterio del ratio crítico (0,58 para ambas estrategias).

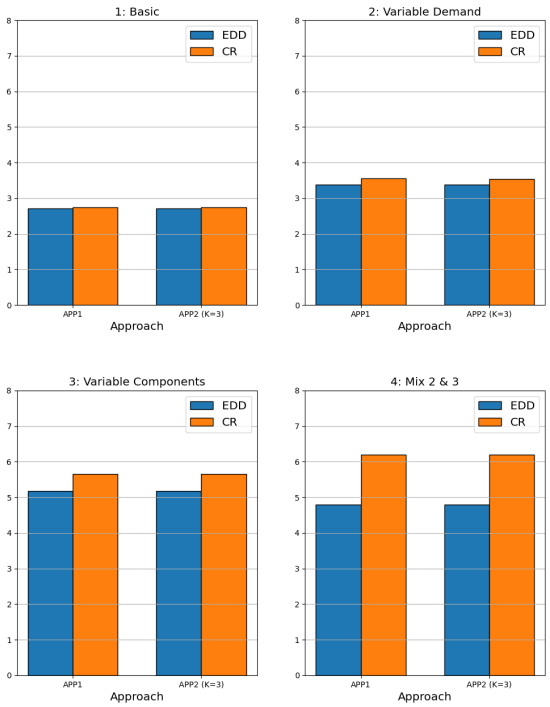

Figura 5.7 Comparativa de los niveles medios de los búferes en cada configuración [Fuente: Elaboración propia].

5.7 Niveles de los búferes

Comenzando por la media, son muy parejas en el primero caso, pero conforme se complica el escenario, el criterio EDD supone una reducción del nivel medio de los buffers, llegando al caso de que el criterio del ratio crítico supone un 29,13 % más de componentes almacenados en el caso 4, frente al 0,96 % del primer caso. La diferencia entre estrategias es inapreciables para cualquiera da los dos criterios estudiados.

En cuanto a la desviación estándar, da una idea de la variabilidad del nivel del búffer a lo largo de la simulación. Al igual que con la media, la desviación es mayor cuando se ha empleado el criterio de la razón crítica (salvo en el primer caso para la estrategia APP1, pero la diferencia en ese caso no llega al 1 %). Esto supone que no solo el tamaño del almacén necesita ser mayor si se utiliza el ratio crítico como criterio, si no que además la variabilidad de este implica que puede pasar mas tiempo vacío y que la probabilidad de llenarlo es mayor que en el caso del EDD.

Se puede apreciar también en la figura 5.8, que la desviación aumenta mucho cuando la demanda varia y cuando los componentes son distintos. Paradójicamente, disminuye un poco (no a niveles del caso 1) en el escenario 4 cuando se mezclan ambos fenómenos, sobre todo para los casos con criterio EDD).

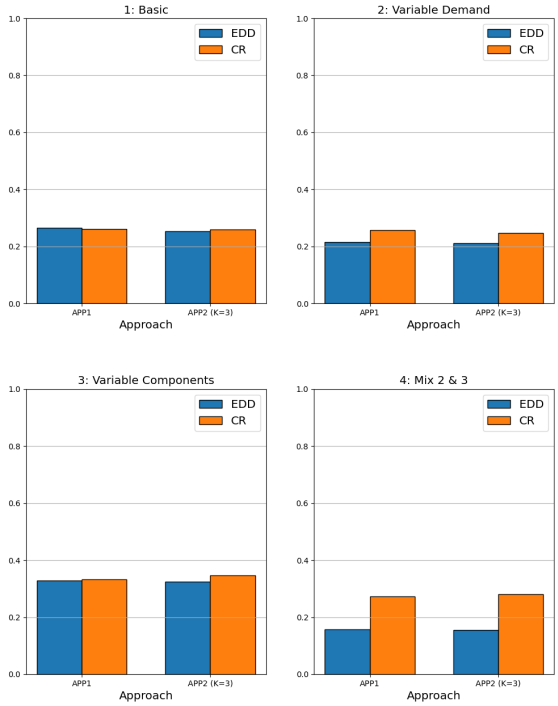

Figura 5.8 Comparativa de la desviación estándar media de los niveles del búferes en cada configuración [Fuente: Elaboración propia].

6 Conclusiones y posibles ampliaciones futuras

Una vez se han comentado los resultados, la superioridad de la norma de despacho el ratio crítico no es tan clara.

El taller que utiliza el criterio del ratio crítico para ordenar las colas ha resultado ser mas lento fabricando, con una mayor carga de trabajo y mayores retrasos, lo que se ha traducido en menor número de pedidos completados. Además utilizar este criterio implica mayor acaparamiento de stock en los búferes antes de acabar un pedido y mayor variabilidad de este stock, lo que se traduce en un almacén mas grande y mayores costes de almacenamiento.

La utilización ha resultado ser prácticamente la misma, pero esto también significa que la planta se encuentra más tiempo desaprovechada, hay máquinas que no se encuentran funcionando. El problema se agrava cuando las colas de pedidos en espera resultan ser mayores que usando EDD, esto plantea la duda sobre la viabilidad de las estrategias estudiadas en este proyecto y el estudio de *Renna et al.*[1]

Sin embargo, estos resultados reafirman la idea planteada al comienzo, a veces lo mas sencillo es mejor. El criterio EDD, presenta mejores resultados siendo una de las reglas de despacho mas sencillas.

Por otro lado, el proyecto cumple con el objetivo de estudiar una nueva aproximación al problema planteado por *Renna et al.* [1], explora otras vías de organización de la planta y ha comparado los resultados con el modelo del estudio original. Este trabajo, pese a que los resultados no han encontrado una alternativa mejor al modelo en el que se basa, cumple con los anteriores objetivos y lo hace usando un lenguaje de programación de código abierto, accesible a todo el mundo.

Sería interesante, estudiar otras reglas de despacho, como la de *Least Slack Time* o introducir nuevas estrategias del control de la producción, alternativas al *Workload Control*. También sería de interés estudiar el comportamiento en situaciones de alta carga, como es el escenario 4, donde se han visto situaciones en las que los resultados de un criterio frente a otro se invierten, con el objetivo de encontrar el punto de inflexión del cambio.

Apéndice A
Código general

A continuación se presenta el código donde se definen y simulan los modelos vistos, y que presenta los resultados analizados y presentes en el Apéndice B

Código A.1 Codigo global empleado.

```
""" MAIN CODE """
import simpy
import random
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import time

start_time = time.time()

print('===== SET-UP =====')
# Parameters
nmaxiter = 1000 # Max number of iterations
tsim = 1000 # Simulation time
Thr_wip = 4 # Max simultaneos WIP orders allowed
Thrb = 6 # Max allowed components in a buffer
Qprob = [[1, 5], [1, 3], [4, 6], [8, 10], [1, 10], [7, 10]] # Uniform
    distribution limits for each component
Case_config = [['Constant', 'Equal'], ['Variable', 'Equal'], ['Constant'
    , 'Different'], ['Variable', 'Different']]
# Configuration of the cases, type of [Demand, Components]

# Performance Results definitions
Mean_Delay_APP1_EDD = np.zeros(4)
Mean_Queue_APP1_EDD = np.zeros(4)
Mean_Throughput_APP1_EDD = np.zeros(4)
Mean_WIP_APP1_EDD = np.zeros(4)
Mean_Utilization_APP1_EDD = np.zeros(4)
Mean_Buffer_APP1_EDD = np.zeros(4)
Dev_Buffer_APP1_EDD = np.zeros(4)
Mean_Completed_APP1_EDD = np.zeros(4)
```

```
Mean_Delay_APP2_EDD = np.zeros(4)
Mean_Queue_APP2_EDD = np.zeros(4)
Mean_Throughput_APP2_EDD = np.zeros(4)
Mean_WIP_APP2_EDD = np.zeros(4)
Mean_Utilization_APP2_EDD = np.zeros(4)
Mean_Buffer_APP2_EDD = np.zeros(4)
Dev_Buffer_APP2_EDD = np.zeros(4)
Mean_Completed_APP2_EDD = np.zeros(4)

Mean_Delay_APP1_CR = np.zeros(4)
Mean_Queue_APP1_CR = np.zeros(4)
Mean_Throughput_APP1_CR = np.zeros(4)
Mean_WIP_APP1_CR = np.zeros(4)
Mean_Utilization_APP1_CR = np.zeros(4)
Mean_Buffer_APP1_CR = np.zeros(4)
Dev_Buffer_APP1_CR = np.zeros(4)
Mean_Completed_APP1_CR = np.zeros(4)

Mean_Delay_APP2_CR = np.zeros(4)
Mean_Queue_APP2_CR = np.zeros(4)
Mean_Throughput_APP2_CR = np.zeros(4)
Mean_WIP_APP2_CR = np.zeros(4)
Mean_Utilization_APP2_CR = np.zeros(4)
Mean_Buffer_APP2_CR = np.zeros(4)
Dev_Buffer_APP2_CR = np.zeros(4)
Mean_Completed_APP2_CR = np.zeros(4)


# Process definitions
# GENERAL
def process_newOrder(env):
    # New order arrives, its data is generated and passed to the preshop
    global orders, preshop, flagarrival
    global level_queue, t_level_queue
    while True:
        Tarrival = 4 # 'Type_Demand == 'Constant'
        if Type_Demand == 'Variable':
            instant = env.now
            if 200 <= instant < 400:
                Tarrival = 6
            elif instant < 600:
                Tarrival = 2
            elif instant < 800:
                Tarrival = 5

        yield env.timeout(random.expovariate(1 / Tarrival)) # Time
            between new orders
```

```
        Q_comp = random.randint(2, 6) # Number of different types of
            components required
        components = [1, 2, 3, 4, 5, 6]
        Qjk = np.zeros(6) # List with the quantity of the component j for
             order k
        for t in range(Q_comp):
            ty = random.choice(components)
            components.remove(ty)
            if Type_Comp == 'Different':
                Qjk[ty - 1] = random.randint(Qprob[ty - 1][0], Qprob[ty -
                     1][1]) # Number of component of type j
            else: # Type_Comp == 'Equal'
                Qjk[ty - 1] = random.randint(1, 6) # Number of component
                    of type j

        DDk = env.now + sum(Qjk) * random.randint(3, 5) # Deadline

        orderk = {'Qjk': Qjk, 'DDk': DDk, 'TO': env.now}

        # Save the new order
        orders.append(orderk)

        # Introduce the order in the preshop
        preshop.append(orderk) # The preshop is a list filled with
            orders waiting to be released

        # Queue performance Measurements
        level_queue.append(len(preshop) - 1) # Before addition
        t_level_queue.append(env.now)

        level_queue.append(len(preshop)) # After addition
        t_level_queue.append(env.now)

        flagarrival.succeed() # Flag event new order has arrived
        flagarrival = env.event() # Reset flag


def process_manufacture(env, mach):
    global queue_mach, buffers, t_level_buffer, level_buffer, WIP,
        t_utilization
    global flagqueue, flagbuffer
    while True:
        if len(queue_mach['machine ' + str(mach)]) > 0:
            # First the machine queue is ordered
            queue_mach['machine ' + str(mach)] = sorted(queue_mach['
                machine ' + str(mach)], key=lambda d: d['DDk'])
            inprocess = queue_mach['machine ' + str(mach)].pop(0) # The
                first operation in queue is picked

            # Manufacturing start
```

```
            inprocess['T_mach'] = env.now
            tstart = env.now # For the utilization calculation
            yield env.timeout(random.expovariate(1)) # Manufacturing time

            # Performance measurements
            t_utilization[mach - 1] += env.now - tstart

            # Send order to buffer
            comp = inprocess['Comp']
            inprocess['T_buff'] = env.now
            buffers['Buffer ' + str(comp)].append(inprocess)

            # Performance measurements
            t_level_buffer.append(env.now)
            instant_level = 0
            for j in range(len(buffers)):
                instant_level += len(buffers['Buffer ' + str(j + 1)]) /
                    len(buffers) # Mean between all the buffers
            level_buffer.append(instant_level)

            flagbuffer.succeed() # Buffer update flag
            flagbuffer = env.event() # Flag reset
        else:
            # Wait for the machine queue to update
            yield flagqueue


def process_assembly(env):
    global buffers, WIP, finished, t_level_buffer, level_buffer
    global flagbuffer, flagassembled, flagqueue, flagbuffout
    while True:
        if len(WIP) > 0:
            # Firstly, organise the WIP
            if sortmode == 'EDD': # If sorting EDD sort the queue by
                Earliest Due Date
                WIP = sorted(WIP, key=lambda d: d['DDk'])
            elif sortmode == 'CritRat': # If sorting with Critical Ratio
                mode
                instant = env.now
                WIP = sorted(WIP, key=lambda d: (d['DDk'] - instant) / (
                    sum(d['Qjk']) * 1 + 1))
            inprocess = WIP[0]
            ready = True
            for j in range(len(buffers)):
                # Check if buffer j has enough components for the order k
                    (bj > Qjk)
                if len(buffers['Buffer ' + str(j + 1)]) < inprocess['Qjk'
                    ][j]:
                    ready = False
```

```
            if ready:
                inprocess['T_ass'] = env.now # The moment the order
                    enters assembly
                operations = []
                for j in range(len(buffers)):
                    # Extract the operations needed from buffer j
                    operationj = buffers['Buffer ' + str(j + 1)][:int(
                        inprocess['Qjk'][j])]
                    del buffers['Buffer ' + str(j + 1)][:int(inprocess['
                        Qjk'][j])]
                    # This operations are added to a list with the used
                        operations
                    operations += operationj

                flagbuffout.succeed() # Extraction from buffer completed
                flagbuffout = env.event() # Reset flag

                # Performance measurements
                t_level_buffer.append(env.now)
                instant_level = 0
                for j in range(len(buffers)):
                    instant_level += len(buffers['Buffer ' + str(j + 1)])
                        / len(buffers) # Mean of all the buffers
                level_buffer.append(instant_level)

                # Add the used operations to the finished order
                inprocess['operations'] = operations
                yield env.timeout(random.expovariate(1)) # Assembly time
                inprocess['T_out'] = env.now

                # Performance measurements (before step)
                level_wip.append(len(WIP))
                t_level_wip.append(env.now)

                # Write the order in the finished list
                finished.append(inprocess)
                del WIP[0]

                flagassembled.succeed() # Finished order event flag
                flagassembled = env.event() # Reset flag

                # Performance measurements (after step)
                level_wip.append(len(WIP))
                t_level_wip.append(env.now)

            else:
                # Wait for buffer update
                yield flagbuffer
        else:
            yield flagqueue # Wait for new order in the WIP
```

```
# APPROACH 1
def process_releaseAPP1(env):
    global preshop, WIP
    global flagqueue, flagbuffout, flagassembled
    while True:
        if len(preshop) > 0:
            b = np.zeros(6) # Temporary list with the lenghts of the
                buffers
            for j in range(6):
                b[j] = len(buffers['Buffer ' + str(j + 1)])
            if len(WIP) < Thr_wip and max(max(Thrb - b), 0): # First &
                Second workload policy respectively
                queuetorelease = [] # Generation of a list of operations
                    to release to the machines
                # Firstly, organise the Preshop
                if sortmode == 'EDD': # If sorting EDD sort the queue by
                    Earliest Due Date
                    preshop = sorted(preshop, key=lambda d: d['DDk'])
                elif sortmode == 'CritRat': # If sorting with Critical
                    Ratio mode
                    instant = env.now
                    preshop = sorted(preshop, key=lambda d: (d['DDk'] -
                        instant) / (sum(d['Qjk']) * 1 + 1))
                order = preshop[0]
                Qjk = order['Qjk']
                DDk = order['DDk']
                for component in range(len(Qjk)):
                    if Qjk[component] != 0:
                        for jj in range(int(Qjk[component])):
                            queuetorelease.append({'DDk': DDk, 'Comp':
                                component + 1}) # An operation is
                            # assigned as a dict with properties

                # Performance Measurements before step
                level_queue.append(len(preshop))
                t_level_queue.append(env.now)

                level_wip.append(len(WIP))
                t_level_wip.append(env.now)

                # Moving orders from preshop to Work-In-Progress list
                WIP.append(preshop[0])
                del preshop[0]

                # Performance Measurements after step
                level_queue.append(len(preshop))
                t_level_queue.append(env.now)
```

```
                level_wip.append(len(WIP))
                t_level_wip.append(env.now)

                # Release of the generated operations
                for operation in queuetorelease:
                    mach = sorted(queue_mach, key=lambda d: len(queue_mach
                        [d]))[
                        0]  # Extract which machine has a shorter queue
                    queue_mach[mach].append(operation) # Operation is
                        introduced in shorter queue
                    flagqueue.succeed() # Queue update event flag
                    flagqueue = env.event() # Event Reset
            else:
                yield flagbuffout or flagassembled # Wait for either the
                    WIP reduction or the buffers update
        else:
            yield flagarrival


# APPROACH 2
def process_releaseAPP2(env):
    global preshop, WIP
    global flagqueue, flagbuffout, flagassembled
    while True:
        if len(preshop) > 0:
            b = np.zeros(6) # Temporary list with the lengths of the
                buffers
            for j in range(6):
                b[j] = len(buffers['Buffer ' + str(j + 1)])
            Q = np.zeros([6, len(orders)]) # Array with the number of
                components needed for each order
            for k in range(len(orders)):
                for j in range(6):
                    Q[j, k] = orders[k]['Qjk'][j]

            aveq = np.array(list(map(lambda x: np.average(Q[x, :]), range
                (6)))) # Average number of components of
            # each type needed
            varq = np.array(list(map(lambda x: np.var(Q[x, :]), range(6))
                )) # Variance of each of the components

            Thrbj = aveq + K * varq # Updatable buffer threshold for each
                 component
            if len(WIP) < Thr_wip and max(max(Thrbj - b), 0): # First &
                Second workload policy respectively
                # Generation of a list of operations to release to the
                    machines
                queuetorelease = []
                # Firstly, organise the Preshop
```

```
            if sortmode == 'EDD': # If sorting EDD sort the queue by
                Earliest Due Date
                preshop = sorted(preshop, key=lambda d: d['DDk'])
            elif sortmode == 'CritRat': # If sorting with Critical
                Ratio mode
                instant = env.now
                preshop = sorted(preshop, key=lambda d: (d['DDk'] -
                    instant) / (sum(d['Qjk']) * 1 + 1))
            order = preshop[0]
            Qjk = order['Qjk']
            DDk = order['DDk']
            for component in range(len(Qjk)):
                if Qjk[component] != 0:
                    for jj in range(int(Qjk[component])):
                        queuetorelease.append({'DDk': DDk, 'Comp':
                            component + 1}) # An operation is
                        # assigned as a dict with properties

            # Performance Measurements before step
            level_queue.append(len(preshop))
            t_level_queue.append(env.now)

            level_wip.append(len(WIP))
            t_level_wip.append(env.now)

            # Moving orders from preshop to Work-In-Progress list
            WIP.append(preshop[0])
            del preshop[0]

            # Performance Measurements after step
            level_queue.append(len(preshop))
            t_level_queue.append(env.now)

            level_wip.append(len(WIP))
            t_level_wip.append(env.now)

            # Release of the generated operations
            for operation in queuetorelease:
                mach = sorted(queue_mach, key=lambda d: len(queue_mach
                    [d]))[0] # Extract which machine has a
                # shorter queue
                queue_mach[mach].append(operation) # Operation is
                    introduced in shorter queue
                flagqueue.succeed() # Queue update event flag
                flagqueue = env.event() # Event Reset
        else:
            yield flagbuffout or flagassembled # Wait for either the
                WIP reduction or the buffers update
    else:
        yield flagarrival
```

```
print('--- Processes defined at %s seconds ---' % (time.time() -
    start_time))
print('===== SIMULATION START =====')

# Define and Run each of the cases
case_counter = 0 # Index of the case being studied

for case in Case_config:
    Type_Demand, Type_Comp = case
    # Performance measuerements
    # Approach 1
    Delay_APP1_EDD = np.zeros(nmaxiter) # Mean delay of each run
    Level_Queue_APP1_EDD = np.zeros(nmaxiter) # Mean number of orders in
        preshop
    Throughput_APP1_EDD = np.zeros(nmaxiter) # Mean order throughput
       time (Tout-T0) for each run
    Level_WIP_APP1_EDD = np.zeros(nmaxiter) # Mean number of orders
       being worked on at the same time for each run
    Utilization_APP1_EDD = np.zeros(nmaxiter) # Mean fraction of the
       time a machine is working
    Level_Buffer_APP1_EDD = np.zeros(nmaxiter) # Mean of the mean buffer
        level for each buffer
    Completed_APP1_EDD = np.zeros(nmaxiter) # Number of orders completed
        each run

    Delay_APP1_CR = np.zeros(nmaxiter) # Mean delay of each run
    Level_Queue_APP1_CR = np.zeros(nmaxiter) # Mean number of orders in
       preshop
    Throughput_APP1_CR = np.zeros(nmaxiter) # Mean order throughput time
        (Tout-T0) for each run
    Level_WIP_APP1_CR = np.zeros(nmaxiter) # Mean number of orders being
        worked on at the same time for each run
    Utilization_APP1_CR = np.zeros(nmaxiter) # Mean fraction of the time
        a machine is working
    Level_Buffer_APP1_CR = np.zeros(nmaxiter) # Mean of the mean buffer
       level for each buffer
    Completed_APP1_CR = np.zeros(nmaxiter) # Number of orders completed
       each run

    # Approach 2 (K = 3)
    Delay_APP2_EDD = np.zeros(nmaxiter) # Mean delay of each run
    Level_Queue_APP2_EDD = np.zeros(nmaxiter) # Mean number of orders in
        preshop
    Throughput_APP2_EDD = np.zeros(nmaxiter) # Mean order throughput
       time (Tout-T0) for each run
    Level_WIP_APP2_EDD = np.zeros(nmaxiter) # Mean number of orders
       being worked on at the same time for each run
```

```
Utilization_APP2_EDD = np.zeros(nmaxiter) # Mean fraction of the
    time a machine is working
Level_Buffer_APP2_EDD = np.zeros(nmaxiter) # Mean of the mean buffer
     level for each buffer
Completed_APP2_EDD = np.zeros(nmaxiter) # Number of orders completed
     each run

Delay_APP2_CR = np.zeros(nmaxiter) # Mean delay of each run
Level_Queue_APP2_CR = np.zeros(nmaxiter) # Mean number of orders in
    preshop
Throughput_APP2_CR = np.zeros(nmaxiter) # Mean order throughput time
     (Tout-T0) for each run
Level_WIP_APP2_CR = np.zeros(nmaxiter) # Mean number of orders being
     worked on at the same time for each run
Utilization_APP2_CR = np.zeros(nmaxiter) # Mean fraction of the time
     a machine is working
Level_Buffer_APP2_CR = np.zeros(nmaxiter) # Mean of the mean buffer
    level for each buffer
Completed_APP2_CR = np.zeros(nmaxiter) # Number of orders completed
    each run

# Consecutive simulations
# APP1: EDD
sortmode = 'EDD'
for niter in range(nmaxiter): # The process is simulated nmaxiter
    times
    orders = [] # List with every order and its data
    preshop = [] # List with the orders queuing for enter production
         (not operations)
    WIP = [] # List with the orders currently being worked on
    finished = [] # List with the finished orders

    # Medidores
    level_arrivals = [0]
    level_dismissed = [0]
    t_level_orders = [0]

    ## Performance measurements
    # Queue
    level_queue = [0]
    t_level_queue = [0]
    # WIP
    level_wip = [0]
    t_level_wip = [0]
    # Utilization
    t_utilization = np.zeros(6)
    # Buffers
    t_level_buffer = [0] # List with the sample times of the buffers
         levels
```

```
level_buffer = [0] # List with the samples of the average buffer
    level (sum of all 6 levels divided by 6)

# Simpy Enviroment
enviroment = simpy.Environment()
arrival = enviroment.process(process_newOrder(enviroment))
release = enviroment.process(process_releaseAPP1(enviroment))

# Definition of the machines and their queues
queue_mach = {}
for i in range(6):
    queue_mach['machine ' + str(i + 1)] = [] # Queues are defines
        as a dictionary with lists of dictionaries

machine1 = enviroment.process(process_manufacture(enviroment, 1))

machine2 = enviroment.process(process_manufacture(enviroment, 2))

machine3 = enviroment.process(process_manufacture(enviroment, 3))

machine4 = enviroment.process(process_manufacture(enviroment, 4))

machine5 = enviroment.process(process_manufacture(enviroment, 5))

machine6 = enviroment.process(process_manufacture(enviroment, 6))


# Buffers definition
buffers = {}
for i in range(6):
    buffers['Buffer ' + str(i + 1)] = [] # Buffers are defined as
        a dictionary of lists

# Definicion assembly
assembly = enviroment.process(process_assembly(enviroment))

# Definition of update flags
flagarrival = enviroment.event() # Flag that warns a new order
    has arrived
flagbuffer = enviroment.event() # Flag that warns a new
    component has entered the buffer
flagbuffout = enviroment.event() # Flag that warns components
    have left the buffers
flagqueue = enviroment.event() # Flag that warns a new component
    has entered the machines queues
flagassembled = enviroment.event() # Flag that warns an order
    has been finished

enviroment.run(until=tsim)
```

```
        # Run performance results
        delayk = []
        Tprocessk = []
        Completed_APP1_EDD[niter] = len(finished)
        for order in finished:
            delayk.append(order['T_out'] - order['DDk']) # Delay
            Tprocessk.append(order['T_out'] - order['T0']) # Process time

        Delay_APP1_EDD[niter] = sum(delayk) / len(delayk)
        Throughput_APP1_EDD[niter] = sum(Tprocessk) / len(Tprocessk)

        # Queue
        Level_Queue_APP1_EDD[niter] = np.average(level_queue[1:],
            weights=np.diff(t_level_queue))
        # WIP
        Level_WIP_APP1_EDD[niter] = np.average(level_wip[1:], weights=np.
            diff(t_level_wip))
        # Utilization
        Utilization_APP1_EDD[niter] = sum(t_utilization) / 6 / tsim #
            Total utilization/ number of machines/ sim time
        # Buffer
        Level_Buffer_APP1_EDD[niter] = np.average(level_buffer[1:],
            weights=np.diff(t_level_buffer))

    # Final Results
    Mean_Delay_APP1_EDD[case_counter] = sum(Delay_APP1_EDD *
        Completed_APP1_EDD) / sum(Completed_APP1_EDD)
    Mean_Queue_APP1_EDD[case_counter] = sum(Level_Queue_APP1_EDD) /
        nmaxiter
    Mean_Throughput_APP1_EDD[case_counter] = sum(Throughput_APP1_EDD *
        Completed_APP1_EDD) / sum(Completed_APP1_EDD)
    Mean_WIP_APP1_EDD[case_counter] = sum(Level_WIP_APP1_EDD) / nmaxiter
    Mean_Utilization_APP1_EDD[case_counter] = sum(Utilization_APP1_EDD)
        / nmaxiter
    Mean_Buffer_APP1_EDD[case_counter] = sum(Level_Buffer_APP1_EDD) /
        nmaxiter
    Dev_Buffer_APP1_EDD[case_counter] = np.std(Level_Buffer_APP1_EDD)
    Mean_Completed_APP1_EDD[case_counter] = sum(Completed_APP1_EDD) /
        nmaxiter

    print('--- FINISHED: CASE ', case, ', APP1, EDD, at %s seconds ---'
        % (time.time() - start_time))

    # APP1: Critical Ratio
    sortmode = 'CritRat'
    for niter in range(nmaxiter): # The process is simulated nmaxiter
        times
        orders = [] # List with every order and its data
        preshop = [] # List with the orders queuing for enter production
            (not operations)
```

```
        WIP = [] # List with the orders currently being worked on
        finished = [] # List with the finished orders

        # Medidores
        level_arrivals = [0]
        level_dismissed = [0]
        t_level_orders = [0]

        ## Performance measurements
        # Queue
        level_queue = [0]
        t_level_queue = [0]
        # WIP
        level_wip = [0]
        t_level_wip = [0]
        # Utilization
        t_utilization = np.zeros(6)
        # Buffers
        t_level_buffer = [0] # List with the sample times of the buffers
            levels
        level_buffer = [0] # List with the samples of the average buffer
            level (sum of all 6 levels divided by 6)

        # Simpy Enviroment
        enviroment = simpy.Environment()
        arrival = enviroment.process(process_newOrder(enviroment))
        release = enviroment.process(process_releaseAPP1(enviroment))

        # Definition of the machines and their queues
        queue_mach = {}
        for i in range(6):
            queue_mach['machine ' + str(i + 1)] = [] # Queues are defines
                as a dictionary with lists of dictionaries

        machine1 = enviroment.process(process_manufacture(enviroment, 1))

        machine2 = enviroment.process(process_manufacture(enviroment, 2))

        machine3 = enviroment.process(process_manufacture(enviroment, 3))

        machine4 = enviroment.process(process_manufacture(enviroment, 4))

        machine5 = enviroment.process(process_manufacture(enviroment, 5))

        machine6 = enviroment.process(process_manufacture(enviroment, 6))


        # Buffers definition
        buffers = {}
        for i in range(6):
```

```
            buffers['Buffer ' + str(i + 1)] = [] # Buffers are defined as
                a dictionary of lists

        # Definicion assembly
        assembly = enviroment.process(process_assembly(enviroment))

        # Definition of update flags
        flagarrival = enviroment.event() # Flag that warns a new order
            has arrived
        flagbuffer = enviroment.event() # Flag that warns a new
            component has entered the buffer
        flagbuffout = enviroment.event() # Flag that warns components
            have left the buffers
        flagqueue = enviroment.event() # Flag that warns a new component
             has entered the machines queues
        flagassembled = enviroment.event() # Flag that warns an order
            has been finished

        enviroment.run(until=tsim)

        # Run performance results
        delayk = []
        Tprocessk = []
        Completed_APP1_CR[niter] = len(finished)
        for order in finished:
            delayk.append(order['T_out'] - order['DDk']) # Delay
            Tprocessk.append(order['T_out'] - order['T0']) # Process time

        Delay_APP1_CR[niter] = sum(delayk) / len(delayk)
        Throughput_APP1_CR[niter] = sum(Tprocessk) / len(Tprocessk)

        # Queue
        Level_Queue_APP1_CR[niter] = np.average(level_queue[1:], weights
            =np.diff(t_level_queue))
        # WIP
        Level_WIP_APP1_CR[niter] = np.average(level_wip[1:], weights=np.
            diff(t_level_wip))
        # Utilization
        Utilization_APP1_CR[niter] = sum(t_utilization) / 6 / tsim #
            Total utilization/ number of machines/ sim time
        # Buffer
        Level_Buffer_APP1_CR[niter] = np.average(level_buffer[1:],
            weights=np.diff(t_level_buffer))

    # Final Results
    Mean_Delay_APP1_CR[case_counter] = sum(Delay_APP1_CR *
        Completed_APP1_CR) / sum(Completed_APP1_CR)
    Mean_Queue_APP1_CR[case_counter] = sum(Level_Queue_APP1_CR) /
        nmaxiter
```

```
    Mean_Throughput_APP1_CR[case_counter] = sum(Throughput_APP1_CR *
        Completed_APP1_CR) / sum(Completed_APP1_CR)
    Mean_WIP_APP1_CR[case_counter] = sum(Level_WIP_APP1_CR) / nmaxiter
    Mean_Utilization_APP1_CR[case_counter] = sum(Utilization_APP1_CR) /
        nmaxiter
    Mean_Buffer_APP1_CR[case_counter] = sum(Level_Buffer_APP1_CR) /
        nmaxiter
    Dev_Buffer_APP1_CR[case_counter] = np.std(Level_Buffer_APP1_CR)
    Mean_Completed_APP1_CR[case_counter] = sum(Completed_APP1_CR) /
        nmaxiter

    print('--- FINISHED: CASE ', case, ', APP1, Critical Ratio, at %s
        seconds ---' % (time.time() - start_time))

    # APP2: EDD
    sortmode = 'EDD'
    K = 3
    for niter in range(nmaxiter): # The process is simulated nmaxiter
        times
        orders = [] # List with every order and its data
        preshop = [] # List with the orders queuing for enter production
            (not operations)
        WIP = [] # List with the orders currently being worked on
        finished = [] # List with the finished orders

        # Medidores
        level_arrivals = [0]
        level_dismissed = [0]
        t_level_orders = [0]

        ## Performance measurements
        # Queue
        level_queue = [0]
        t_level_queue = [0]
        # WIP
        level_wip = [0]
        t_level_wip = [0]
        # Utilization
        t_utilization = np.zeros(6)
        # Buffers
        t_level_buffer = [0] # List with the sample times of the buffers
            levels
        level_buffer = [0] # List with the samples of the average buffer
            level (sum of all 6 levels divided by 6)

        # Simpy Enviroment
        enviroment = simpy.Environment()
        arrival = enviroment.process(process_newOrder(enviroment))
        release = enviroment.process(process_releaseAPP2(enviroment))
```

```
# Definition of the machines and their queues
queue_mach = {}
for i in range(6):
    queue_mach['machine ' + str(i + 1)] = [] # Queues are defines
        as a dictionary with lists of dictionaries

machine1 = enviroment.process(process_manufacture(enviroment, 1))

machine2 = enviroment.process(process_manufacture(enviroment, 2))

machine3 = enviroment.process(process_manufacture(enviroment, 3))

machine4 = enviroment.process(process_manufacture(enviroment, 4))

machine5 = enviroment.process(process_manufacture(enviroment, 5))

machine6 = enviroment.process(process_manufacture(enviroment, 6))


# Buffers definition
buffers = {}
for i in range(6):
    buffers['Buffer ' + str(i + 1)] = [] # Buffers are defined as
        a dictionary of lists

# Definicion assembly
assembly = enviroment.process(process_assembly(enviroment))

# Definition of update flags
flagarrival = enviroment.event() # Flag that warns a new order
    has arrived
flagbuffer = enviroment.event() # Flag that warns a new
    component has entered the buffer
flagbuffout = enviroment.event() # Flag that warns components
    have left the buffers
flagqueue = enviroment.event() # Flag that warns a new component
     has entered the machines queues
flagassembled = enviroment.event() # Flag that warns an order
    has been finished

enviroment.run(until=tsim)

# Run performance results
delayk = []
Tprocessk = []
Completed_APP2_EDD[niter] = len(finished)
for order in finished:
    delayk.append(order['T_out'] - order['DDk']) # Delay
    Tprocessk.append(order['T_out'] - order['T0']) # Process time
```

```
        Delay_APP2_EDD[niter] = sum(delayk) / len(delayk)
        Throughput_APP2_EDD[niter] = sum(Tprocessk) / len(Tprocessk)

        # Queue
        Level_Queue_APP2_EDD[niter] = np.average(level_queue[1:],
            weights=np.diff(t_level_queue))
        # WIP
        Level_WIP_APP2_EDD[niter] = np.average(level_wip[1:], weights=np.
            diff(t_level_wip))
        # Utilization
        Utilization_APP2_EDD[niter] = sum(t_utilization) / 6 / tsim #
            Total utilization/ number of machines/ sim time
        # Buffer
        Level_Buffer_APP2_EDD[niter] = np.average(level_buffer[1:],
            weights=np.diff(t_level_buffer))

    # Final Results
    Mean_Delay_APP2_EDD[case_counter] = sum(Delay_APP2_EDD *
        Completed_APP2_EDD) / sum(Completed_APP2_EDD)
    Mean_Queue_APP2_EDD[case_counter] = sum(Level_Queue_APP2_EDD) /
        nmaxiter
    Mean_Throughput_APP2_EDD[case_counter] = sum(Throughput_APP2_EDD *
        Completed_APP2_EDD) / sum(Completed_APP2_EDD)
    Mean_WIP_APP2_EDD[case_counter] = sum(Level_WIP_APP2_EDD) / nmaxiter
    Mean_Utilization_APP2_EDD[case_counter] = sum(Utilization_APP2_EDD)
        / nmaxiter
    Mean_Buffer_APP2_EDD[case_counter] = sum(Level_Buffer_APP2_EDD) /
        nmaxiter
    Dev_Buffer_APP2_EDD[case_counter] = np.std(Level_Buffer_APP2_EDD)
    Mean_Completed_APP2_EDD[case_counter] = sum(Completed_APP2_EDD) /
        nmaxiter

    print('--- FINISHED: CASE ', case, ', APP2, EDD, at %s seconds ---'
        % (time.time() - start_time))

    # APP2: Critical Ratio
    sortmode = 'CritRat'
    K = 3
    for niter in range(nmaxiter): # The process is simulated nmaxiter
        times
        orders = [] # List with every order and its data
        preshop = [] # List with the orders queuing for enter production
             (not operations)
        WIP = [] # List with the orders currently being worked on
        finished = [] # List with the finished orders

        # Medidores
        level_arrivals = [0]
        level_dismissed = [0]
        t_level_orders = [0]
```

```
## Performance measurements
# Queue
level_queue = [0]
t_level_queue = [0]
# WIP
level_wip = [0]
t_level_wip = [0]
# Utilization
t_utilization = np.zeros(6)
# Buffers
t_level_buffer = [0] # List with the sample times of the buffers
    levels
level_buffer = [0] # List with the samples of the average buffer
    level (sum of all 6 levels divided by 6)

# Simpy Enviroment
enviroment = simpy.Environment()
arrival = enviroment.process(process_newOrder(enviroment))
release = enviroment.process(process_releaseAPP2(enviroment))

# Definition of the machines and their queues
queue_mach = {}
for i in range(6):
    queue_mach['machine ' + str(i + 1)] = [] # Queues are defines
        as a dictionary with lists of dictionaries

machine1 = enviroment.process(process_manufacture(enviroment, 1))

machine2 = enviroment.process(process_manufacture(enviroment, 2))

machine3 = enviroment.process(process_manufacture(enviroment, 3))

machine4 = enviroment.process(process_manufacture(enviroment, 4))

machine5 = enviroment.process(process_manufacture(enviroment, 5))

machine6 = enviroment.process(process_manufacture(enviroment, 6))


# Buffers definition
buffers = {}
for i in range(6):
    buffers['Buffer ' + str(i + 1)] = [] # Buffers are defined as
        a dictionary of lists

# Definicion assembly
assembly = enviroment.process(process_assembly(enviroment))

# Definition of update flags
```

```
        flagarrival = enviroment.event() # Flag that warns a new order
            has arrived
        flagbuffer = enviroment.event() # Flag that warns a new
            component has entered the buffer
        flagbuffout = enviroment.event() # Flag that warns components
            have left the buffers
        flagqueue = enviroment.event() # Flag that warns a new component
             has entered the machines queues
        flagassembled = enviroment.event() # Flag that warns an order
            has been finished

        enviroment.run(until=tsim)

        # Run performance results
        delayk = []
        Tprocessk = []
        Completed_APP2_CR[niter] = len(finished)
        for order in finished:
            delayk.append(order['T_out'] - order['DDk']) # Delay
            Tprocessk.append(order['T_out'] - order['T0']) # Process time

        Delay_APP2_CR[niter] = sum(delayk) / len(delayk)
        Throughput_APP2_CR[niter] = sum(Tprocessk) / len(Tprocessk)

        # Queue
        Level_Queue_APP2_CR[niter] = np.average(level_queue[1:], weights
            =np.diff(t_level_queue))
        # WIP
        Level_WIP_APP2_CR[niter] = np.average(level_wip[1:], weights=np.
            diff(t_level_wip))
        # Utilization
        Utilization_APP2_CR[niter] = sum(t_utilization) / 6 / tsim #
            Total utilization/ number of machines/ sim time
        # Buffer
        Level_Buffer_APP2_CR[niter] = np.average(level_buffer[1:],
            weights=np.diff(t_level_buffer))

    # Final Results
    Mean_Delay_APP2_CR[case_counter] = sum(Delay_APP2_CR *
        Completed_APP2_CR) / sum(Completed_APP2_CR)
    Mean_Queue_APP2_CR[case_counter] = sum(Level_Queue_APP2_CR) /
        nmaxiter
    Mean_Throughput_APP2_CR[case_counter] = sum(Throughput_APP2_CR *
        Completed_APP2_CR) / sum(Completed_APP2_CR)
    Mean_WIP_APP2_CR[case_counter] = sum(Level_WIP_APP2_CR) / nmaxiter
    Mean_Utilization_APP2_CR[case_counter] = sum(Utilization_APP2_CR) /
        nmaxiter
    Mean_Buffer_APP2_CR[case_counter] = sum(Level_Buffer_APP2_CR) /
        nmaxiter
    Dev_Buffer_APP2_CR[case_counter] = np.std(Level_Buffer_APP2_CR)
```

```
    Mean_Completed_APP2_CR[case_counter] = sum(Completed_APP2_CR) /
        nmaxiter

    print('--- FINISHED: CASE ', case, ', APP2, Critical Ratio, at %s
        seconds ---' % (time.time() - start_time))

    # Update Case counter
    case_counter += 1

# Global Results graphs

print('--- FINISHED SIMULATIONS at %s seconds ---' % (time.time() -
    start_time))
print('===== START PLOTTING =====')

Titles = ['Mean Delay', 'Mean Queue Level', 'Mean Throughput Time', '
    Mean WIP Level',
          'Mean Utilization Level', 'Mean Buffer Level', 'Buffers Std.
             Dev.', 'Mean Completed Orders']
SubTitles = ['1: Basic', '2: Variable Demand',
             '3: Variable Components', '4: Mix 2 & 3']
y_lim = [[-70, 55], [0, 90], [0, 150], [0, 4.5], [0, 1], [0, 8], [0, 1],
     [0, 350]]

Means_APP1_EDD = [Mean_Delay_APP1_EDD, Mean_Queue_APP1_EDD,
    Mean_Throughput_APP1_EDD, Mean_WIP_APP1_EDD,
                  Mean_Utilization_APP1_EDD, Mean_Buffer_APP1_EDD,
                     Dev_Buffer_APP1_EDD, Mean_Completed_APP1_EDD]
Means_APP2_EDD = [Mean_Delay_APP2_EDD, Mean_Queue_APP2_EDD,
    Mean_Throughput_APP2_EDD, Mean_WIP_APP2_EDD,
                  Mean_Utilization_APP2_EDD, Mean_Buffer_APP2_EDD,
                     Dev_Buffer_APP2_EDD, Mean_Completed_APP2_EDD]
Means_APP1_CR = [Mean_Delay_APP1_CR, Mean_Queue_APP1_CR,
    Mean_Throughput_APP1_CR, Mean_WIP_APP1_CR,
                 Mean_Utilization_APP1_CR, Mean_Buffer_APP1_CR,
                    Dev_Buffer_APP1_CR, Mean_Completed_APP1_CR]
Means_APP2_CR = [Mean_Delay_APP2_CR, Mean_Queue_APP2_CR,
    Mean_Throughput_APP2_CR, Mean_WIP_APP2_CR,
                 Mean_Utilization_APP2_CR, Mean_Buffer_APP2_CR,
                    Dev_Buffer_APP2_CR, Mean_Completed_APP2_CR]

Approaches = ['APP1', 'APP2 (K=3)']
Criteria = ['EDD', 'CR']
pos = np.arange(len(Approaches))
bar_width = 0.35

for fig_id in range(len(Titles)):
    fig = plt.figure(Titles[fig_id])

    for subfig_id in range(len(SubTitles)):
```

```
        plt.subplot(2, 2, subfig_id + 1)
        plt.bar(pos, [Means_APP1_EDD[fig_id][subfig_id], Means_APP2_EDD[
            fig_id][subfig_id]],
                bar_width, edgecolor='black')
        plt.bar(pos + bar_width, [Means_APP1_CR[fig_id][subfig_id],
            Means_APP2_CR[fig_id][subfig_id]],
                bar_width, edgecolor='black')
        plt.xticks(pos+bar_width/2, Approaches)
        plt.xlabel('Approach', fontsize='x-large')
        plt.legend(Criteria, loc=0, fontsize='x-large')
        plt.grid(True, axis='y')
        plt.title(SubTitles[subfig_id], fontsize='x-large')
        plt.ylim(y_lim[fig_id])

    plt.subplots_adjust(hspace=0.3)
    fig.set_size_inches(12, 15)
    plt.savefig(Titles[fig_id] + '.png', dpi=100)

print('--- FINISHED PLOTTING AT %s seconds ---' % (time.time() -
    start_time))

print('===== START DATA SAVING =====')
# Save the data to a file
Columns = ['Case', 'Approach', 'Sorting Criteria', 'Demand Type', '
    Components type'] + Titles
Cases = [1, 2, 3, 4]
Data = [] # Where is going to be stored

for case_id in range(len(Cases)):
    a = [Cases[case_id], Approaches[0], Criteria[0], Case_config[case_id
        ][0], Case_config[case_id][1]]
    b = [Cases[case_id], Approaches[1], Criteria[0], Case_config[case_id
        ][0], Case_config[case_id][1]]
    c = [Cases[case_id], Approaches[0], Criteria[1], Case_config[case_id
        ][0], Case_config[case_id][1]]
    d = [Cases[case_id], Approaches[1], Criteria[1], Case_config[case_id
        ][0], Case_config[case_id][1]]
    for column_id in range(len(Titles)):
        a.append(Means_APP1_EDD[column_id][case_id])
        b.append(Means_APP2_EDD[column_id][case_id])
        c.append(Means_APP1_CR[column_id][case_id])
        d.append(Means_APP2_CR[column_id][case_id])

    Data.append(a)
    Data.append(b)
    Data.append(c)
    Data.append(d)

Data = pd.DataFrame(Data, columns=Columns)
Data.to_csv('Global_Data.csv', index=False)
```

```
print('--- FINISHED: DATA SAVING at %s seconds ---' % (time.time() -
    start_time))

plt.show()
```

Apéndice B
Tablas de Resultados

Tabla B.1 Tabla de Resultados I.

Case	Approach	Sorting Criteria	Demand Type	Components type	Mean Delay
1	APP1	EDD	Constant	Equal	-48,031
1	APP2 (K=3)	EDD	Constant	Equal	-48,097
1	APP1	CR	Constant	Equal	-46,939
1	APP2 (K=3)	CR	Constant	Equal	-47,026
2	APP1	EDD	Variable	Equal	-30,871
2	APP2 (K=3)	EDD	Variable	Equal	-30,569
2	APP1	CR	Variable	Equal	-28,104
2	APP2 (K=3)	CR	Variable	Equal	-27,875
3	APP1	EDD	Constant	Different	-64,704
3	APP2 (K=3)	EDD	Constant	Different	-64,904
3	APP1	CR	Constant	Different	-59,371
3	APP2 (K=3)	CR	Constant	Different	-59,643
4	APP1	EDD	Variable	Different	55,820
4	APP2 (K=3)	EDD	Variable	Different	54,859
4	APP1	CR	Variable	Different	49,296
4	APP2 (K=3)	CR	Variable	Different	48,419

Tabla B.2 Tabla de Resultados II.

Case	Approach	Sorting Criteria	Mean Queue Level	Mean Throughput Time	Mean WIP Level
1	APP1	EDD	0,172	7,801	1,784
1	APP2 (K=3)	EDD	0,169	7,795	1,780
1	APP1	CR	0,302	8,916	1,927
1	APP2 (K=3)	CR	0,292	8,846	1,914
2	APP1	EDD	5,481	25,043	2,580
2	APP2 (K=3)	EDD	5,596	25,391	2,586
2	APP1	CR	6,269	27,752	2,654
2	APP2 (K=3)	CR	6,361	28,042	2,651
3	APP1	EDD	2,908	22,352	2,782
3	APP2 (K=3)	EDD	2,858	22,195	2,780
3	APP1	CR	4,075	27,780	2,930
3	APP2 (K=3)	CR	3,990	27,537	2,922
4	APP1	EDD	42,139	142,502	3,249
4	APP2 (K=3)	EDD	41,667	141,599	3,252
4	APP1	CR	40,366	134,266	3,256
4	APP2 (K=3)	CR	39,882	133,385	3,257

Tabla B.3 Tabla de Resultados III.

Case	Approach	Sorting Criteria	Mean Utilization Level	Mean Buffer Level	Buffers Std. Dev.	Mean Completed Orders
1	APP1	EDD	0,582	2,719	0,265	248,676
1	APP2 (K=3)	EDD	0,580	2,712	0,254	247,848
1	APP1	CR	0,580	2,747	0,262	247,782
1	APP2 (K=3)	CR	0,578	2,736	0,258	247,271
2	APP1	EDD	0,750	3,383	0,216	320,401
2	APP2 (K=3)	EDD	0,751	3,387	0,211	320,792
2	APP1	CR	0,748	3,548	0,257	320,043
2	APP2 (K=3)	CR	0,749	3,541	0,247	319,896
3	APP1	EDD	0,891	5,171	0,329	243,414
3	APP2 (K=3)	EDD	0,890	5,185	0,324	242,874
3	APP1	CR	0,887	5,654	0,332	241,839
3	APP2 (K=3)	CR	0,885	5,650	0,346	241,405
4	APP1	EDD	0,962	4,801	0,157	264,152
4	APP2 (K=3)	EDD	0,963	4,794	0,155	264,274
4	APP1	CR	0,963	6,196	0,273	268,976
4	APP2 (K=3)	CR	0,963	6,193	0,281	268,904

Índice de Figuras

Índice de Tablas

Índice de Códigos

Bibliografía

[1] P. Renna, D. Carlucci y S. Materi, «A Decision Making Model for Order Release in an Assembly Job-Shop to Improve Business Performance and Sustainability», en *Multiple Criteria Decision Making for Sustainable Development: Pursuing Economic Growth, Environmental Protection and Social Cohesion*, M. Doumpos, F. A. F. Ferreira y C. Zopounidis, eds. Cham: Springer International Publishing, 2021, págs. 193-211, ISBN: 978-3-030-89277-7. DOI: *10.1007/978-3-030-89277-7_9*. dirección: *https://doi.org/10.1007/978-3-030-89277-7_9*.

[2] J. L. Riggs, *Sistemas de producción : planeación, análisis y control*, 3rd ed. Limusa, 1998, ISBN: 968-18-4878-0.

[3] T. Fernandez y E. Tamaro. «Biografías y Vidas: Biografia de Frederick Winslow Taylor». (2004), dirección: *https://www.biografiasyvidas.com/biografia/t/taylor_frederick.htm* (visitado 05-07-2022).

[4] ——, «Biografías y Vidas: Biografia de Henry Gantt». (2004), dirección: *https://www.biografiasyvidas.com/biografia/g/gantt.htm* (visitado 05-07-2022).

[5] L. O. Giménez, *Diseño y gestión de sistemas productivos*, A. E. Santana, P. C. Achedad, J. M. Sanz y J. G. Martín, eds. Dextra, 2017, ISBN: 9788416898343.

[6] U. Akinc y J. R. Meredith, «Make-to-forecast: customization with fast delivery», *International Journal of Operations & Production Management*, vol. 35, págs. 728-750, 5 ene. de 2015, ISSN: 0144-3577. DOI: *10.1108/IJOPM-12-2012-0567*. dirección: *https://doi.org/10.1108/IJOPM-12-2012-0567*.

[7] M. P. Groover, *Automation, production systems, and computer integrated manufacturing*. Prentice-Hall International, 1987, ISBN: 0-13-054610-0.

[8] R. McHaney, *Understanding Computer Simulation*. Ventus Publishing, 2009, ISBN: 9788776815059. dirección: *https://books.google.es/books?id=WVjc9Iz8PUYC*.

[9] M. L. Whicker, *Computer simulation applications : an introduction*, L. Sigelman, ed. Sage, 1991, ISBN: 0-8039-3245-6.

[10] A. Meurer, C. P. Smith, M. Paprocki y col., *SymPy: symbolic computing in Python*, ene. de 2017, e103. DOI: *10.7717/peerj-cs.103*. dirección: *https://doi.org/10.7717/peerj-cs.103*.